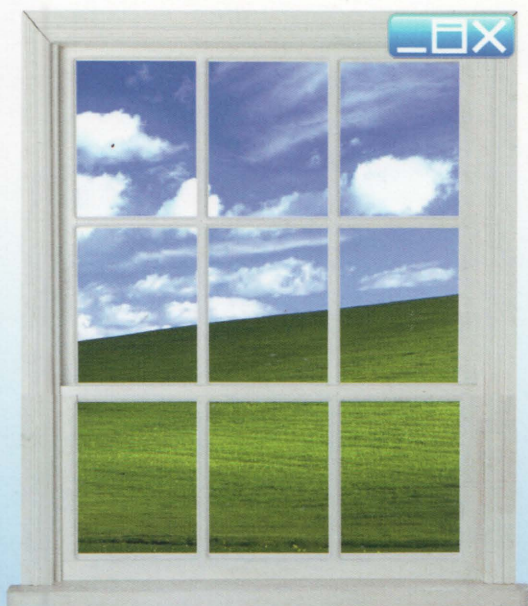


JavaFX



- Управление окнами приложения
- Размещение компонентов внутри окна
- Обработка событий
- Работа с графикой и изображениями
- Трансформации и эффекты
- Создание анимации
- Графики и диаграммы
- Воспроизведение аудио и видео
- JavaFX CSS
- Программа Scene Builder



Материалы
на www.bhv.ru

**Наиболее
полное
руководство**

В ПОДЛИННИКЕ®

Николай Прохоренок

JavaFX

Санкт-Петербург
«БХВ-Петербург»
2020

УДК 004.438Java
ББК 32.973.26-018.1
П84

Прохоренок Н. А.

П84 JavaFX. — СПб.: БХВ-Петербург, 2020. — 768 с.: ил. — (В подлиннике)
ISBN 978-5-9775-4072-8

Описываются базовые возможности библиотеки JavaFX, позволяющей создавать приложения с графическим интерфейсом на языке Java. Рассматриваются способы обработки событий, управление свойствами окна, создание формы с помощью программы Scene Builder, а также все основные компоненты (кнопки, текстовые поля, списки, таблицы, меню и др.) и варианты их размещения внутри окна. Описаны трансформации и эффекты, графики и диаграммы, аудио и видео, стили JavaFX CSS. Книга ориентирована на тех, кто уже знаком с языком программирования Java и хотел бы научиться разрабатывать оконные приложения, насыщенные графикой, анимацией и интерактивными элементами. Большое количество практических примеров помогает начать разработку самостоятельно. Весь материал тщательно подобран, хорошо структурирован и компактно изложен, что позволяет использовать книгу как удобный справочник. Электронный архив с примерами находится на сайте издательства.

Для программистов

УДК 004.438Java
ББК 32.973.26-018.1

Группа подготовки издания:

Руководитель проекта	<i>Евгений Рыбаков</i>
Зав. редакцией	<i>Екатерина Сависте</i>
Компьютерная верстка	<i>Ольги Сергиенко</i>
Дизайн серии	<i>Марины Дамбиевой</i>
Оформление обложки	<i>Карины Соловьевой</i>

Подписано в печать 01.10.19.
Формат 70×100^{1/16}. Печать офсетная. Усл. печ. л. 61,92.
Тираж 1000 экз. Заказ № 10019.
"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.
Отпечатано с готового оригинал-макета
ООО "Принт-М", 142300, М.О., г. Чехов, ул. Полиграфистов, д. 1

ISBN 978-5-9775-4072-8

© ООО "БХВ", 2020
© Оформление. ООО "БХВ-Петербург",

Оглавление

Предисловие	15
Глава 1. Первые шаги	19
1.1. Установка OpenJDK	19
1.2. Установка библиотеки JavaFX.....	20
1.3. Создание и запуск JavaFX-приложения из командной строки	24
1.4. Установка и настройка редактора Eclipse	27
1.5. Установка модуля <i>e(fx)clipse</i>	34
1.6. Установка программы Scene Builder.....	36
1.7. Создание JavaFX-приложения в Eclipse. Способ 1	38
1.8. Создание JavaFX-приложения в Eclipse. Способ 2	49
1.9. Указание идентификаторов компонентов	56
1.10. Доступ к главному окну приложения внутри класса контроллера.....	58
1.11. Метод <i>initialize()</i> и интерфейс <i>Initializable</i>	62
1.12. Методы <i>init()</i> и <i>stop()</i>	64
1.13. Получение параметров из командной строки	66
1.14. Завершение работы JavaFX-приложения.....	68
1.15. Выполнение длительных операций.....	69
Глава 2. Управление окном приложения	71
2.1. Создание и отображение окна	71
2.2. Указание стиля окна	75
2.3. Изменение и получение размеров окна	77
2.4. Местоположение окна на экране.....	79
2.4.1. Класс <i>Screen</i> : размеры экрана.....	80
2.5. Разворачивание и сворачивание окна	81
2.6. Управление фокусом ввода и положением окна по оси <i>Z</i>	83
2.7. Управление прозрачностью окна	84
2.8. Модальные окна.....	85
2.9. Смена значка в заголовке окна	87
2.10. Изменение цвета фона.....	88
2.11. Использование изображения в качестве фона.....	89
2.12. Создание окна произвольной формы	90
2.13. Закрытие окна из программы	91

Глава 3. Размещение компонентов в окне	93
3.1. Класс <i>Scene</i>	93
3.1.1. Создание объекта.....	93
3.1.2. Размеры сцены.....	95
3.1.3. Местоположение сцены	95
3.1.4. Фон сцены	96
3.2. Размеры контейнеров и компонентов.....	97
3.3. Внутренние отступы.....	100
3.3.1. Класс <i>Insets</i>	100
3.4. Местоположение узлов внутри контейнера.....	102
3.5. Фон контейнеров и компонентов	105
3.5.1. Класс <i>Background</i>	105
3.5.2. Класс <i>BackgroundFill</i> : заливка сплошным цветом, градиентом или текстурой	107
3.5.3. Класс <i>CornerRadii</i> : радиус скругления углов	108
3.5.4. Класс <i>BackgroundImage</i> : фоновое изображение	109
3.6. Рамка.....	111
3.7. Изменение прозрачности узла	113
3.8. Управление видимостью узла.....	114
3.9. Управление доступностью узла.....	115
3.10. Абсолютное позиционирование	116
3.10.1. Класс <i>Group</i>	116
3.10.2. Класс <i>Pane</i>	118
3.11. Горизонтальное и вертикальное выравнивание	119
3.11.1. Класс <i>HBox</i> : выравнивание по горизонтали	119
3.11.2. Класс <i>VBox</i> : выравнивание по вертикали	122
3.12. Класс <i>GridPane</i> : размещение узлов внутри ячеек таблицы	125
3.12.1. Класс <i>ColumnConstraints</i> : описание столбцов.....	127
3.12.2. Класс <i>RowConstraints</i> : описание строк	129
3.12.3. Добавление узла в ячейку таблицы	131
3.12.4. Объединение нескольких ячеек.....	134
3.12.5. Изменение свойств ячейки.....	135
3.12.6. Изменение свойств таблицы	136
3.13. Класс <i>FlowPane</i>	138
3.14. Класс <i>BorderPane</i>	140
3.15. Класс <i>StackPane</i>	143
3.16. Класс <i>TilePane</i>	144
3.17. Класс <i>AnchorPane</i> : привязка узла к сторонам панели	147
3.18. Класс <i>TabPane</i> : панель с вкладками.....	150
3.18.1. Класс <i>Tab</i> : вкладка.....	151
3.18.2. Свойства класса <i>TabPane</i>	153
3.19. Класс <i>TitledPane</i> : панель с заголовком	154
3.20. Класс <i>Accordion</i> : панель «аккордеон».....	156
3.21. Класс <i>SplitPane</i> : панель с изменяемыми размерами областей.....	157
Глава 4. Обработка событий.....	161
4.1. Назначение обработчиков событий	161
4.2. Блокировка и удаление обработчика	165
4.3. Последовательность вызова обработчиков	166

4.4. Генерация события из программы	167
4.5. Класс <i>Event</i>	168
4.6. Использование таймеров.....	170
4.6.1. Класс <i>Timer</i>	171
4.6.2. Класс <i>TimerTask</i>	172
4.6.3. Изменение свойств узлов из разных потоков.....	172
4.7. События окна	174
4.7.1. Отображение, сокрытие и закрытие окна.....	174
4.7.2. Изменение положения окна и его размеров	177
4.7.3. Разворачивание и сворачивание окна	178
4.8. События клавиатуры	178
4.8.1. Типы событий	179
4.8.2. Получение информации о событии.....	180
4.8.3. Коды клавиш	181
4.8.4. Управление фокусом ввода.....	183
4.8.5. Назначение клавиш быстрого доступа	185
4.9. События мыши	187
4.9.1. Типы событий	188
4.9.2. Получение информации о событии.....	192
4.9.3. Прокрутка колесика мыши	194
4.9.4. Изменение внешнего вида указателя мыши	196
4.10. Работа с буфером обмена.....	197
4.10.1. Класс <i>Clipboard</i>	197
4.10.2. Класс <i>ClipboardContent</i>	199
4.11. Технология drag & drop.....	202
4.11.1. Типы событий	202
4.11.2. Запуск перетаскивания	205
4.11.3. Обработка сброса.....	206
4.11.4. Получение информации о событии.....	207
4.11.5. Класс <i>Dragboard</i>	208
4.11.6. Класс <i>MouseDragEvent</i>	209
4.12. Класс <i>Robot</i> : генерация событий мыши и клавиатуры	211
Глава 5. 2D-графика	215
5.1. Указание координат и размеров.....	215
5.1.1. Класс <i>Point2D</i> : координаты точки	215
5.1.2. Класс <i>Dimension2D</i> : размеры прямоугольной области	219
5.1.3. Класс <i>Rectangle2D</i> : координаты и размеры прямоугольной области	219
5.1.4. Классы <i>Bounds</i> и <i>BoundingBox</i>	221
5.2. Класс <i>Paint</i>	225
5.3. Класс <i>Color</i> : цвет	226
5.4. Класс <i>LinearGradient</i> : линейный градиент	230
5.5. Класс <i>RadialGradient</i> : радиальный градиент	233
5.6. Режимы наложения.....	236
5.7. Маски слоя	238
5.8. Класс <i>Shape</i>	238
5.8.1. Изменение характеристик фона фигуры.....	239
5.8.2. Изменение характеристик обводки	240
5.8.3. Управление режимом сглаживания.....	243

5.9. Класс <i>Line</i> : прямая линия	244
5.10. Класс <i>Arc</i> : дуга или сектор	245
5.11. Класс <i>Polyline</i> : ломаная линия	248
5.12. Класс <i>CubicCurve</i> : кубическая кривая Безье	249
5.13. Класс <i>QuadCurve</i> : квадратичная кривая	251
5.14. Класс <i>Rectangle</i> : прямоугольник	252
5.15. Класс <i>Ellipse</i> : эллипс	254
5.16. Класс <i>Circle</i> : круг	255
5.17. Класс <i>Polygon</i> : многоугольник	257
5.18. Класс <i>Text</i> : текст	258
5.18.1. Создание объекта	258
5.18.2. Подчеркивание и зачеркивание текста	260
5.18.3. Класс <i>Font</i> : изменение характеристик шрифта	261
5.18.4. Выравнивание текста	264
5.18.5. Выделение фрагмента текста	265
5.18.6. Управление положением текстового курсора	266
5.18.7. Класс <i>HitInfo</i>	267
5.18.8. Класс <i>TextFlow</i>	267
5.19. Класс <i>Path</i> : траектория	269
5.19.1. Класс <i>PathElement</i>	271
5.19.2. Классы <i>MoveTo</i> и <i>ClosePath</i>	272
5.19.3. Класс <i>HLineTo</i> : горизонтальная линия	273
5.19.4. Класс <i>VLineTo</i> : вертикальная линия	273
5.19.5. Класс <i>LineTo</i> : линия	274
5.19.6. Класс <i>ArcTo</i> : дуга	275
5.19.7. Класс <i>CubicCurveTo</i> : кубическая кривая Безье	276
5.19.8. Класс <i>QuadCurveTo</i> : квадратичная кривая	277
5.20. Класс <i>SVGPath</i>	278

Глава 6. Работа с изображениями 283

6.1. Класс <i>Image</i> : загрузка изображения из файла	283
6.2. Класс <i>ImageView</i> : отображение изображения в окне	286
6.3. Интерфейс <i>PixelReader</i> : получение цвета пикселей	288
6.4. Класс <i>WritableImage</i> : создание нового изображения	290
6.5. Интерфейс <i>PixelWriter</i> : изменение цвета пикселей	292
6.6. Класс <i>SwingFXUtils</i>	294
6.7. Класс <i>ImagePattern</i>	296
6.8. Создание снимков сцены, узла и экрана	297

Глава 7. Canvas API 301

7.1. Класс <i>Canvas</i>	301
7.2. Класс <i>GraphicsContext</i>	302
7.2.1. Изменение характеристик заливки	303
7.2.2. Изменение характеристик обводки	304
7.2.3. Изменение цвета отдельных пикселей	306
7.2.4. Рисование линий	307
7.2.5. Рисование траектории	308
7.2.6. Рисование фигур	311
7.2.7. Вывод текста	313

7.2.8. Вывод изображения.....	315
7.2.9. Очистка прямоугольной области или всего холста	316
7.2.10. Сохранение и восстановление состояния	317
7.2.11. Применение эффектов и трансформаций	317
Глава 8. Трансформации и эффекты.....	321
8.1. Трансформации.....	321
8.1.1. Методы класса <i>Node</i>	321
8.1.2. Классы <i>Transform</i> и <i>Affine</i>	323
8.1.3. Класс <i>Translate</i> : смещение	326
8.1.4. Класс <i>Scale</i> : масштабирование	328
8.1.5. Класс <i>Rotate</i> : вращение	329
8.1.6. Класс <i>Shear</i> : сдвиг	331
8.2. Эффекты	333
8.2.1. Методы класса <i>Node</i>	333
8.2.2. Класс <i>DropShadow</i> : внешняя тень	333
8.2.3. Класс <i>InnerShadow</i> : внутренняя тень	335
8.2.4. Класс <i>Shadow</i> : тень.....	337
8.2.5. Класс <i>Reflection</i> : зеркальное отражение	338
8.2.6. Класс <i>GaussianBlur</i> : размытие по Гауссу	339
8.2.7. Класс <i>MotionBlur</i> : размытие в движении.....	340
8.2.8. Класс <i>BoxBlur</i> : размытие.....	341
8.2.9. Класс <i>Bloom</i> : свечение	342
8.2.10. Класс <i>Glow</i> : свечение	343
8.2.11. Класс <i>PerspectiveTransform</i> : трансформация перспективы	344
8.2.12. Класс <i>ColorAdjust</i> : изменение цветового тона, насыщенности, яркости и контраста	346
8.2.13. Класс <i>SepiaTone</i> : эффект состаривания изображения	347
8.2.14. Класс <i>Lighting</i> : эффект освещения источником света.....	348
8.2.15. Класс <i>Light</i> : источник освещения.....	349
8.2.16. Класс <i>Blend</i> : смешивание	352
8.2.17. Класс <i>ColorInput</i>	354
8.2.18. Класс <i>ImageInput</i>	355
8.2.19. Класс <i>DisplacementMap</i> : искажение.....	356
Глава 9. Создание анимации.....	359
9.1. Класс <i>Duration</i> : продолжительность выполнения анимации	359
9.2. Класс <i>Animation</i>	362
9.2.1. Настройка параметров.....	362
9.2.2. Использование меток	364
9.2.3. Запуск и остановка анимации	365
9.3. Классы <i>Transition</i> и <i>Interpolator</i>	366
9.4. Класс <i>FadeTransition</i> : изменение прозрачности	368
9.5. Класс <i>TranslateTransition</i> : изменение местоположения	369
9.6. Класс <i>PathTransition</i> : движение вдоль траектории.....	372
9.7. Класс <i>ScaleTransition</i> : изменение масштаба.....	373
9.8. Класс <i>RotateTransition</i> : вращение.....	375
9.9. Класс <i>FillTransition</i> : изменение цвета заливки.....	377
9.10. Класс <i>StrokeTransition</i> : изменение цвета обводки	378

9.11. Класс <i>ParallelTransition</i> : параллельное выполнение нескольких анимаций	380
9.12. Класс <i>SequentialTransition</i> : последовательное выполнение нескольких анимаций	381
9.13. Класс <i>PauseTransition</i> : пауза во время выполнения анимации	382
9.14. Класс <i>Timeline</i> : шкала времени	383
9.14.1. Класс <i>KeyFrame</i>	384
9.14.2. Класс <i>KeyValue</i>	386
9.15. Класс <i>AnimationTimer</i>	387
Глава 10. 3D-графика	389
10.1. Класс <i>Point3D</i>	389
10.2. Класс <i>Shape3D</i>	393
10.3. Класс <i>Box</i> : куб	394
10.4. Класс <i>Cylinder</i> : цилиндр	396
10.5. Класс <i>Sphere</i> : сфера	397
10.6. Класс <i>MeshView</i> : 3D-фигура произвольной формы	398
10.7. Класс <i>SubScene</i> : субсцена	401
10.8. Управление камерой	403
10.9. Управление источником света	405
Глава 11. Основные компоненты	409
11.1. Поиск узла	409
11.2. Использование JavaFX-свойств	414
11.2.1. Создание JavaFX-свойства	415
11.2.2. JavaFX-свойства только для чтения	416
11.2.3. Назначение и удаление обработчиков	417
Интерфейсы <i>InvalidationListener</i> и <i>Observable</i>	417
Интерфейсы <i>ChangeListener<T></i> и <i>ObservableValue<T></i>	418
11.2.4. Двухнаправленное связывание	419
11.2.5. Однонаправленное связывание	421
11.2.6. Выражения	422
11.3. Классы <i>Labeled</i> и <i>Label</i>	426
11.4. Класс <i>ButtonBase</i>	433
11.5. Класс <i>Hyperlink</i> : гиперссылка	434
11.6. Класс <i>Button</i> : командная кнопка	435
11.7. Класс <i>ButtonBar</i>	437
11.8. Переключатели	440
11.8.1. Класс <i>ToggleButton</i> и интерфейс <i>Toggle</i>	440
11.8.2. Класс <i>RadioButton</i>	441
11.8.3. Класс <i>ToggleGroup</i> : объединение переключателей в группу	442
11.9. Класс <i>CheckBox</i> : флажок	444
11.10. Класс <i>Slider</i> : шкала с ползунком	446
11.11. Индикаторы хода процесса	449
11.11.1. Класс <i>ProgressIndicator</i>	449
11.11.2. Класс <i>ProgressBar</i>	451
11.11.3. Класс <i>Task<V></i> : задача для выполнения в отдельном потоке	451
11.11.4. Интерфейс <i>Worker<V></i>	456
11.11.5. Обработка событий выполнения задачи	457
11.11.6. Класс <i>Service<V></i>	460

11.12. Класс <i>ScrollBar</i> : полоса прокрутки	462
11.13. Класс <i>ScrollPane</i> : область с полосами прокрутки	465
11.14. Класс <i>ToolBar</i> : панель инструментов	468
11.15. Класс <i>Separator</i> : разделительная линия	470

Глава 12. Работа с текстом и Интернетом.....473

12.1. Класс <i>Spinner</i> <T>	473
12.2. Класс <i>TextInputControl</i>	478
12.2.1. Основные свойства и методы	479
12.2.2. Управление позицией текстового курсора	480
12.2.3. Класс <i>IndexRange</i> : работа с диапазоном	481
12.2.4. Работа с выделением	483
12.2.5. Копирование и вставка текста	485
12.2.6. Отмена и повтор ввода	485
12.2.7. Класс <i>TextFormatter</i> <V>	486
12.3. Класс <i>TextField</i> : однострочное текстовое поле	489
12.4. Класс <i>PasswordField</i> : поле для ввода пароля	491
12.5. Класс <i>TextArea</i> : многострочное текстовое поле	491
12.6. Класс <i>HTMLEditor</i> : текстовый редактор	493
12.7. Класс <i>WebView</i> : Web-браузер	495
12.8. Класс <i>WebEngine</i>	497
12.8.1. Загрузка Web-страницы	498
12.8.2. Получение информации о Web-странице	500
12.8.3. Взаимодействие с JavaScript	501
12.8.4. Получение информации о просмотренных страницах	508

Глава 13. Списки.....511

13.1. Интерфейс <i>ObservableList</i> <E>	511
13.1.1. Создание списка	512
13.1.2. Методы из интерфейса <i>ObservableList</i> <E>	514
13.1.3. Назначение и удаление обработчиков. Интерфейс <i>ListChangeListener</i> <E>	516
13.1.4. Методы из класса <i>FXCollections</i>	520
13.2. Класс <i>ChoiceBox</i> <T>: раскрывающийся список	522
13.3. Класс <i>SingleSelectionModel</i> <T>: выбор одного элемента	525
13.4. Класс <i>ComboBoxBase</i> <T>	527
13.5. Класс <i>ComboBox</i> <T>: раскрывающийся список	530
13.6. Класс <i>ColorPicker</i> : выбор цвета	535
13.7. Класс <i>DatePicker</i> : выбор даты	537
13.8. Класс <i>ListView</i> <T>: список	540
13.8.1. Создание объекта	540
13.8.2. Основные свойства и методы	542
13.8.3. Класс <i>FocusModel</i> <T>: управление фокусом ввода	545
13.8.4. Редактирование элементов	546
13.9. Класс <i>MultipleSelectionModel</i> <T>: выбор нескольких элементов	549

Глава 14. Таблицы и иерархические списки.....553

14.1. Класс <i>TableView</i> <S>: таблица	553
14.1.1. Создание таблицы	553
14.1.2. Класс <i>TableColumn</i> <S, T>: создание столбцов	555
14.1.3. Объединение столбцов в группу	559

14.1.4. Управление размерами столбцов	560
14.1.5. Изменение видимости столбца	562
14.1.6. Управление прокруткой	563
14.1.7. Класс <i>TableViewSelectionMode<S></i> : управление выбором строк и отдельных ячеек	565
14.1.8. Класс <i>TableViewFocusModel<S></i> : управление фокусом ввода	568
14.1.9. Изменение свойств ячеек столбца	570
14.1.10. Изменение свойств ячеек строки	571
14.1.11. Сортировка элементов	572
14.1.12. Редактирование элементов	574
14.1.13. Изменение порядка следования столбцов	579
14.2. Класс <i>TreeView<T></i> : иерархический список	579
14.2.1. Создание иерархического списка	580
14.2.2. Класс <i>TreelItem<T></i> : элемент иерархического списка	581
14.2.3. Обработка событий	584
14.2.4. Управление прокруткой	587
14.2.5. Управление выбором элементов	588
14.2.6. Управление фокусом ввода	589
14.2.7. Редактирование элементов	590
14.2.8. Класс <i>CheckBoxTreelItem<T></i> : флажок в качестве элемента списка	592
14.3. Класс <i>TreeTableView<S></i> : иерархический список со столбцами	595
14.3.1. Создание списка и добавление элементов	595
14.3.2. Класс <i>TreeTableColumn<S, T></i> : создание столбцов	599
14.3.3. Объединение столбцов в группу	602
14.3.4. Управление размерами столбцов	603
14.3.5. Изменение видимости столбца	605
14.3.6. Управление прокруткой	606
14.3.7. Класс <i>TreeTableViewSelectionMode<S></i> : управление выбором строк и отдельных ячеек	607
14.3.8. Класс <i>TreeTableViewFocusModel<S></i> : управление фокусом ввода	611
14.3.9. Изменение свойств ячеек столбца	613
14.3.10. Изменение свойств ячеек строки	614
14.3.11. Сортировка элементов	615
14.3.12. Редактирование элементов	617
14.3.13. Изменение порядка следования столбцов	621
Глава 15. Меню и всплывающие окна	623
15.1. Класс <i>MenuBar</i> : панель меню	624
15.2. Класс <i>Menu</i> : отдельное или вложенное меню	625
15.3. Пункты меню	628
15.3.1. Класс <i>MenuItem</i> : пункт меню	628
15.3.2. Обработка выбора пункта меню	630
15.3.3. Назначение клавиш быстрого доступа	632
15.3.4. Класс <i>CheckMenuItem</i> : пункт меню с флажком	634
15.3.5. Класс <i>RadioMenuItem</i> : пункт меню с переключателем	635
15.3.6. Класс <i>SeparatorMenuItem</i> : разделитель пунктов меню	636
15.3.7. Класс <i>CustomMenuItem</i> : произвольный пункт меню	637
15.4. Класс <i>MenuButton</i> : кнопка вызова меню	638
15.5. Класс <i>SplitMenuButton</i> : кнопка с меню	641

15.6. Класс <i>PopupWindow</i> : всплывающее окно	642
15.6.1. Отображение и сокрытие всплывающего окна	642
15.6.2. Изменение и получение размеров всплывающего окна	644
15.6.3. Местоположение всплывающего окна на экране	645
15.6.4. Класс <i>Popup</i>	646
15.7. Класс <i>PopupControl</i> : всплывающее окно	647
15.8. Класс <i>Tooltip</i> : всплывающие подсказки	649
15.9. Класс <i>ContextMenu</i> : контекстное меню	652
15.10. Класс <i>ContextMenuEvent</i> : событие вызова контекстного меню	653
Глава 16. Диалоговые окна	655
16.1. Класс <i>Dialog<R></i>	655
16.1.1. Создание и отображение диалогового окна	655
16.1.2. Размеры и местоположение диалогового окна	658
16.1.3. Обработка событий	658
16.1.4. Класс <i>DialogPane</i>	660
16.1.5. Класс <i>ButtonType</i> : кнопки внутри диалогового окна	662
16.1.6. Получение результата	663
16.2. Класс <i>Alert</i> : окно с сообщением	664
16.3. Класс <i>TextInputDialog</i> : окно с текстовым полем	667
16.4. Класс <i>ChoiceDialog<T></i> : окно со списком	668
16.5. Класс <i>DirectoryChooser</i> : окно для выбора папки	669
16.6. Класс <i>FileChooser</i> : окно для выбора файла	671
Глава 17. Графики и диаграммы	675
17.1. Класс <i>Chart</i>	675
17.2. Класс <i>PieChart</i> : круговая диаграмма	676
17.3. Класс <i>XYChart<X, Y></i>	678
17.3.1. Класс <i>Axis<T></i> : ось диаграммы	679
17.3.2. Классы <i>ValueAxis<T></i> и <i>NumberAxis</i> : ось диаграммы с числовыми значениями	681
17.3.3. Свойства класса <i>XYChart<X, Y></i>	683
17.3.4. Класс <i>XYChart.Series<X, Y></i> : серия данных	684
17.3.5. Класс <i>XYChart.Data<X, Y></i> : данные	685
17.3.6. Класс <i>CategoryAxis</i> : ось диаграммы со строковыми значениями	686
17.4. Класс <i>LineChart<X, Y></i> : линейный график	687
17.5. Класс <i>BarChart<X, Y></i> : гистограмма	689
17.6. Класс <i>StackedBarChart<X, Y></i> : гистограмма	691
17.7. Класс <i>BubbleChart<X, Y></i> : пузырьковая диаграмма	693
17.8. Класс <i>AreaChart<X, Y></i> : линейный график с заливкой области	694
17.9. Класс <i>StackedAreaChart<X, Y></i> : линейный график с заливкой области	696
17.10. Класс <i>ScatterChart<X, Y></i> : диаграмма в виде символов	697
Глава 18. Аудио и видео	699
18.1. Класс <i>AudioClip</i> : воспроизведение аудио	699
18.2. Класс <i>Media</i>	702
18.3. Класс <i>MediaPlayer</i>	703
18.3.1. Обработка изменения статуса и ошибок	704
18.3.2. Управление воспроизведением	705
18.3.3. Настройки воспроизведения аудио и видео	707

18.4. Класс <i>MediaView</i>	708
18.5. Класс <i>Track</i> : дорожка	710
Глава 19. JavaFX CSS: изменяем вид интерфейса с помощью стилей	713
19.1. Способы встраивания определения стиля	713
19.1.1. Встраивание определения стиля в узел	713
19.1.2. Вынесение таблицы стилей в отдельный файл	714
19.1.3. Приоритет применения стилей	716
19.2. Указание значений атрибутов	717
19.2.1. Числа	717
19.2.2. Размеры	718
19.2.3. Цвет	718
19.2.4. Строки	720
19.2.5. Углы	720
19.3. Селекторы	720
19.3.1. Основные селекторы	720
19.3.2. Псевдоклассы	721
19.4. Форматирование шрифта	722
19.4.1. Имя шрифта	723
19.4.2. Стиль шрифта	723
19.4.3. Размер шрифта	723
19.4.4. Цвет текста	723
19.4.5. Жирность шрифта	723
19.4.6. Одновременное указание характеристик шрифта	724
19.4.7. Загружаемые шрифты	724
19.5. Форматирование текста	725
19.5.1. Подчеркивание и зачеркивание текста	725
19.5.2. Обрезка или перенос текста на новую строку	725
19.5.3. Горизонтальное выравнивание текста	725
19.5.4. Взаимодействие между изображением и текстом	726
19.5.5. Выравнивание изображения и текста внутри области	726
19.6. Размеры и внутренние отступы	727
19.7. Рамки	727
19.7.1. Стиль линии рамки	727
19.7.2. Толщина линии рамки	729
19.7.3. Цвет линии рамки	729
19.7.4. Рамки со скругленными углами	729
19.7.5. Расстояние между рамкой и границей	730
19.8. Фон	730
19.8.1. Цвет фона	730
19.8.2. Расстояние между фоном и границей	731
19.8.3. Скругление углов фона	731
19.8.4. Фоновый рисунок	731
19.8.5. Режим повтора фонового рисунка	731
19.8.6. Положение фонового рисунка	732
19.8.7. Размеры фонового рисунка	732
19.9. Трансформации и эффекты	733
19.9.1. Изменение прозрачности	733
19.9.2. Смещение	733

19.9.3. Масштабирование.....	734
19.9.4. Вращение.....	734
19.9.5. Изменение режима наложения	734
19.9.6. Применение эффектов.....	734
19.10. Изменение внешнего вида указателя мыши	735
19.11. Изменение характеристик фигуры	735
19.11.1. Фон фигуры.....	735
19.11.2. Обводка	735
19.11.3. Сглаживание.....	736
19.11.4. Скругление углов прямоугольника	736
19.12. Изменение характеристик полос прокрутки.....	737
Глава 20. Разное.....	739
20.1. Интернационализация приложения.....	739
20.2. Настройка запуска приложения.....	742
20.2.1. Опция командной строки <i>-splash</i>	743
20.2.2. Директива <i>SplashScreen-Image</i>	743
20.2.3. Класс <i>Preloader</i>	744
20.3. Взаимодействие между JavaFX и Swing	747
20.3.1. Класс <i>SwingNode</i>	747
20.3.2. Класс <i>JFXPanel</i>	748
20.4. Утилита <i>jlink.exe</i> : создание дистрибутива	750
Заключение.....	753
Приложение. Описание электронного архива.....	754
Предметный указатель	755

Предисловие

Добро пожаловать в мир JavaFX!

JavaFX — это библиотека для создания на языке Java приложений с графическим интерфейсом, насыщенных графикой, анимацией и интерактивными элементами. Библиотека является кроссплатформенной, поэтому мы можем создавать оконные приложения под Windows, Linux и Mac, а также мобильные приложения под Android и iOS (подробную информацию о создании мобильных приложений можно найти на странице <https://gluonhq.com/products/mobile/>). В этой книге мы рассмотрим процесс создания оконных приложений, применительно к операционной системе Windows.

До версии Java 8 библиотека JavaFX была доступна отдельно. В версиях Java с 8 по 10 включительно библиотека JavaFX входит в состав стандартной библиотеки, поэтому ее дополнительно устанавливать не нужно. Начиная с Java 11, JavaFX поставляется отдельно в виде набора модулей, доступных на странице <https://gluonhq.com/products/javafx/>. В этой книге мы будем изучать возможности библиотеки JavaFX версий 11 и 12.

Давайте рассмотрим структуру книги:

- ❑ *Глава 1* является вводной. Мы установим необходимое программное обеспечение, настроим среду разработки, скомпилируем и запустим первую программу — как из командной строки, так и из редактора Eclipse. Кроме того, вкратце разберемся со структурой программы, научимся управлять объектом приложения, а также рассмотрим процесс создания формы с помощью программы Scene Builder.
- ❑ В *главе 2* рассматривается жизненный цикл главного окна приложения: создание окна, его отображение, сокрытие и закрытие, сворачивание и разворачивание до максимального размера или во весь экран, отображение окна поверх всех окон, а также возможность блокировки других окон приложения до момента закрытия окна. Мы научимся изменять размеры окна, управлять его местоположением на экране, менять значок в заголовке окна и др.
- ❑ *Глава 3* полностью посвящена менеджерам компоновки (сцене, контейнерам и панелям), позволяющим управлять размещением компонентов внутри окна.

- При взаимодействии пользователя с окном происходят *события*. В ответ на события генерируются определенные сигналы — своего рода извещения о том, что пользователь выполнил какое-либо действие, или в самой системе возникло некоторое условие. События являются важнейшей составляющей приложения с графическим интерфейсом, поэтому необходимо знать, как назначить обработчик события, как удалить обработчик, а также уметь правильно обработать событие. Обработку событий мы рассмотрим в *главе 4*.
- В *главе 5* мы научимся работать с 2D-графикой, а в *главе 6* — с изображениями.
- В *главе 7* будет рассмотрен компонент `Canvas`, позволяющий программно рисовать на поверхности, которую мы называем *холстом*. Можно нарисовать любую геометрическую фигуру с заливкой или обводкой, вывести текст различными шрифтами, добавить на холст изображение, а также применить к графическим объектам эффекты или трансформации.
- В *главе 8* мы научимся применять различные трансформации — как к элементам холста, так и к обычным узлам, — например, сдвигать их, масштабировать и вращать. Помимо трансформаций мы рассмотрим применение различных эффектов — например, добавление внешней или внутренней тени, создание зеркального отражения и др.
- Все рассмотренные в *главе 8* трансформации применяются сразу. Однако часто нужно, чтобы трансформации применялись постепенно. Например, при наведении указателя мыши на пункт меню надо плавно увеличить значок и изменить его прозрачность, а при отведении указателя также плавно вернуть настройки в первоначальное положение. В *главе 9* мы как раз и научимся это делать, а также рассмотрим создание произвольной анимации.
- В *главе 10* мы научимся работать с 3D-графикой.
- *Глава 11* посвящена основным компонентам пользовательского интерфейса, таким как надписи, кнопки, переключатели, флажки и др. Кроме того, мы рассмотрим запуск задач в отдельных потоках, с отображением хода выполнения задачи, а также использование `JavaFX`-свойств.
- В *главе 12* мы рассмотрим различные компоненты, позволяющие работать с текстом и Web-страницами, — например, однострочные и многострочные текстовые поля.
- В `JavaFX` имеется широкий выбор компонентов, позволяющих отображать одномерный список строк в свернутом или развернутом состояниях. Списки поддерживают концепцию «модель/представление», позволяющую отделить данные от их отображения. В роли модели выступает объект, реализующий интерфейс `ObservableList<E>`, а в роли представления — компоненты `ChoiceBox<T>`, `ComboBox<T>` и `ListView<T>`. Одномерные списки мы рассмотрим в *главе 13*, а в *главе 14* научимся работать с таблицами и иерархическими списками.
- *Глава 15* посвящена созданию главного меню приложения — основного компонента пользовательского интерфейса, позволяющего компактно разместить множество команд, объединяя их в логические группы. Кроме того, мы рассмот-

рим возможность отображения контекстного меню, а также различных всплывающих окон, например, с текстом подсказки.

- В *главе 16* мы научимся создавать различные диалоговые окна, которые предназначены для информирования пользователя, а также для получения данных от него.
- Библиотека JavaFX содержит множество классов, реализующих различные виды диаграмм, — например, круговую диаграмму, линейный график, гистограмму и т. д. Эти классы мы подробно рассмотрим в *главе 17*.
- В *главе 18* мы научимся воспроизводить аудио и видео.
- Библиотека JavaFX позволяет изменять внешний вид окна, контейнеров и компонентов с помощью *каскадных таблиц стилей* (CSS, Cascading Style Sheets), точно так же, как это делается в Web-программировании. Применение стилей позволяет задавать точные характеристики практически всех элементов пользовательского интерфейса. Применять стили мы научимся в *главе 19*.
- И наконец, в *главе 20* мы рассмотрим возможность вывода текста надписей на различных языках в зависимости от настроек локали, научимся отображать окно с заставкой на время запуска приложения, а также создадим дистрибутив для запуска приложения, включающий только необходимые модули и компоненты.

Чтобы уменьшить объем книги, основная часть кода примеров вынесена в отдельные проекты, которые размещены в электронном архиве. Электронный архив можно загрузить с FTP-сервера издательства «БХВ-Петербург» по ссылке: <ftp://ftp.bhv.ru/9785977540728.zip> или со страницы книги на сайте www.bhv.ru (см. *приложение*).

Каждый проект состоит из двух файлов:

- JAR-архив — содержит скомпилированные файлы с классами (Java 11), файл манифеста, необходимые ресурсы (например, изображения), а также файлы с исходным кодом классов. Чтобы посмотреть исходный код примеров, достаточно распаковать JAR-архив с помощью любого архиватора. Исходный код дополнительно содержит фрагменты, которые закомментированы. Эти фрагменты являются либо альтернативным способом написания кода, либо ограничивают количество данных, выводимых в консоль. Запустить примеры можно двойным щелчком левой кнопкой мыши на значке JAR-архива при условии создания ассоциации запуска с файлом `runJAR11.bat` (см. *разд. 1.3*). Такой способ запуска не отображает окно консоли, поэтому отладочные данные, выводимые в консоль, будут недоступны;
- BAT-файл — содержит код для запуска JAR-архива в Windows с отображением окна консоли, в которое будет выводиться отладочная информация. Запустить файл можно двойным щелчком левой кнопкой мыши на значке BAT-файла. Предварительно нужно указать в файле `path_to_JRE.txt` полный путь к JRE, а в файле `path_to_JavaFX.txt` — полный путь до библиотеки JavaFX. В составе пути следует избегать пробелов и русских букв, а в файлах должна быть только одна строка без символов перевода строки, иначе получите ошибку.

Желаю приятного изучения и надеюсь, что книга поможет вам реализовать как самые простые, так и самые сложные приложения.

ГЛАВА 1



Первые шаги

Надеюсь, что вы владеете основами языка Java и знаете, как запустить командную строку. Если это не так, то вначале следует изучить язык Java, например, по моей книге «Основы Java»¹. Материала настолько много, что у меня нет возможности объяснять здесь, что такое класс, метод, поле и т. д. Мы будем изучать библиотеку JavaFX и не станем тратить объем книги на основы языка.

1.1. Установка OpenJDK

Для изучения возможностей библиотеки JavaFX нужно установить на компьютер Java 11. Мы воспользуемся не Oracle JDK, а свободной реализацией JDK, которая имеет название OpenJDK. OpenJDK представляет собой комплект разработчика Java Development Kit (сокращенно JDK), включающий компилятор `javac.exe`, стандартную библиотеку, исполнительную среду Java Runtime Environment (сокращенно JRE) и различные утилиты. Для загрузки дистрибутива переходим на страницу <http://jdk.java.net/11/> и скачиваем архив `openjdk-<Версия>_windows-x64_bin.zip`. Далее в корне диска C: создаем каталог OpenJDK и копируем в него каталог `jdk-11` из архива. Путь до компилятора должен быть таким: `C:\OpenJDK\jdk-11\bin\javac.exe`.

Далее необходимо добавить путь к каталогу `C:\OpenJDK\jdk-11\bin` в системную переменную `PATH`. Если вы устанавливали предыдущие версии Java, то нужно дополнительно удалить путь `C:\ProgramData\Oracle\Java\javapath`, иначе будет запускаться другая версия Java.

Чтобы изменить системную переменную, переходим в **Параметры | Панель управления | Система и безопасность | Система | Дополнительные параметры системы**. В результате откроется окно **Свойства системы**. На вкладке **Дополнительно** нажимаем кнопку **Переменные среды**. В списке **Системные переменные** открывшегося окна выделяем строку с переменной **Path** и нажимаем кнопку **Изменить**. В открывшемся окне изменяем значение в поле **Значение переменной** — для

¹ Прохоренко Н. А. Основы Java. — СПб.: БХВ-Петербург, 2017. — 704 с.: ил. (см. <http://www.bhv.ru/books/book.php?id=198051>).

этого переходим в конец существующей строки, ставим точку с запятой, а затем вводим путь к каталогу `C:\OpenJDK\jdk-11\bin`:

```
<Текущее значение>;C:\OpenJDK\jdk-11\bin
```

ВНИМАНИЕ!

Случайно не удалите существующее значение переменной `PATH`, иначе другие приложения перестанут запускаться.

Помимо добавления в переменную `PATH` пути к каталогу `C:\OpenJDK\jdk-11\bin`, во избежание проблем с настройками различных редакторов, необходимо прописать переменную окружения `JAVA_HOME` и присвоить ей путь к каталогу с установленным JDK. Делается это также в окне **Переменные среды**, но в списке переменных для пользователя, хотя вы можете добавить ее и в список системных переменных. Нажимаем в этом окне кнопку **Создать**, в поле **Имя переменной** вводим значение `JAVA_HOME`, а в поле **Значение переменной** — `C:\OpenJDK\jdk-11`. Сохраняем все изменения.

Запускаем командную строку и проверяем установку переменной `JAVA_HOME`:

```
C:\book>set JAVA_HOME
JAVA_HOME=C:\OpenJDK\jdk-11
```

Чтобы проверить правильность установки Java 11, выполняем следующие команды:

```
java --version
javac --version
```

Результат должен быть примерно следующим:

```
C:\book>java --version
openjdk 11 2018-09-25
OpenJDK Runtime Environment 18.9 (build 11+28)
OpenJDK 64-Bit Server VM 18.9 (build 11+28, mixed mode)
```

```
C:\book>javac --version
javac 11
```

ПРИМЕЧАНИЕ

Если вы получили в результате другую версию Java или компилятора, то вначале перезапустите командную строку и повторите проверку. Если версия не изменилась, то, скорее всего, в переменной `PATH` прописан путь к предыдущей версии Java. Путь к старой версии Java нужно удалить.

1.2. Установка библиотеки JavaFX

Для начала создадим следующую структуру каталогов:

```
book\
  src\
  bin\
  modules\
```

```
JavaSE\  
  eclipse\  
    JavaFX\  
      projects\
```

Каталоги `book` и `JavaSE` лучше разместить в корне какого-либо диска. В моем случае это будет диск `C:`, следовательно, пути к содержимому каталогов: `C:\book` и `C:\JavaSE`. Можно создать каталог и в любом другом месте, но в пути не должно быть русских букв и пробелов — только английские буквы, цифры, тире и подчеркивание. Остальных символов лучше избегать, если не хотите проблем с компиляцией и запуском программ.

Кстати, рекомендация не использовать русские буквы относится и к имени пользователя. Многие программы сохраняют настройки в каталоге `C:\Users\<Имя пользователя>`. В результате в пути окажутся русские буквы, которые могут стать причиной множества проблем.

В каталоге `book` мы станем размещать наши тестовые программы. Вложенный каталог `src` мы будем использовать для файлов с исходным кодом, каталог `bin` — для скомпилированных файлов, а каталог `modules` — для модульных JAR-архивов.

Внутри каталога `JavaSE` у нас созданы два вложенных каталога:

- ☐ `eclipse` — в этот каталог мы установим редактор Eclipse;
- ☐ `JavaFX` — в этот каталог мы установим библиотеку JavaFX и программу Scene Builder, а во вложенном каталоге `projects` станем сохранять проекты из редактора Eclipse.

Как уже отмечалось ранее, библиотека JavaFX входит в состав стандартной библиотеки, начиная с Java 8 и заканчивая Java 10. При использовании указанных версий Java дополнительно устанавливать JavaFX не нужно. Начиная с Java 11, библиотека JavaFX не входит в состав стандартной библиотеки, и ее приходится устанавливать дополнительно в виде комплекта модулей, называемого JavaFX SDK. Официальный сайт библиотеки JavaFX: <https://openjfx.io/>. Документацию к библиотеке JavaFX можно найти на странице: <https://openjfx.io/javadoc/<Версия>/>.

Для загрузки библиотеки JavaFX 12 переходим на страницу <https://gluonhq.com/products/javafx/> и из раздела **JavaFX Windows SDK** скачиваем архив `openjfx-12_windows-x64_bin-sdk.zip`. Распаковываем этот архив в каталог `C:\JavaSE\JavaFX`, а затем переименовываем каталог `javafx-sdk-12` в `12`, чтобы сделать путь к библиотеке более коротким, т. к. нам придется его указывать каждый раз в качестве пути поиска модулей при компиляции и запуске программ. Путь до модуля `javafx.base` должен быть таким: `C:\JavaSE\JavaFX\12\lib\javafx.base.jar`.

Для создания дистрибутива нам нужно еще скачать архив с файлами в формате `jmod`. Со страницы <https://gluonhq.com/products/javafx/> из раздела **JavaFX Windows jmods** скачиваем архив `openjfx-12_windows-x64_bin-jmods.zip`. Распаковываем архив и переименовываем каталог с модулями в `jmods`, а затем копируем его в каталог `C:\JavaSE\JavaFX\12`. Путь до модуля `javafx.base` должен быть таким: `C:\JavaSE\JavaFX\12\jmods\javafx.base.jmod`. Процесс создания дистрибутива мы рассмотрим в разд. 20.4.

JavaFX SDK состоит из трех каталогов:

- ❑ **bin** — содержит библиотеки динамической компоновки (DLL), необходимые для запуска JavaFX-приложений;
- ❑ **legal** — содержит различную информацию о модулях, в частности лицензии;
- ❑ **lib** — содержит модули, файл с настройками и файл `src.zip` с исходным кодом модулей. Путь к этому каталогу нужно прописывать в качестве пути поиска модулей в опции `--module-path` (или `-p`) при компиляции и запуске JavaFX-приложений.

Библиотека JavaFX состоит из следующих модулей (расположены в каталоге `C:\JavaSE\JavaFX\12\lib`):

- ❑ **javafx.base** — базовый модуль, от которого зависят все остальные модули библиотеки JavaFX. Зависимость от других модулей:

```
C:\book>jdeps --module-path C:\JavaSE\JavaFX\12\lib -s
           -m javafx.base
javafx.base -> java.base
javafx.base -> java.desktop
```

- ❑ **javafx.graphics** — содержит пакеты с классами, позволяющими создать объект приложения, различные окна, управлять размещением компонентов внутри сцены, работать с изображениями, графикой, анимацией, эффектами, трансформациями и пр. Зависимость от других модулей:

```
C:\book>jdeps --module-path C:\JavaSE\JavaFX\12\lib -s
           -m javafx.graphics
javafx.graphics -> java.base
javafx.graphics -> java.desktop
javafx.graphics -> java.xml
javafx.graphics -> javafx.base
javafx.graphics -> jdk.unsupported
```

- ❑ **javafx.controls** — содержит пакеты с классами, позволяющими создать различные компоненты (например, надпись, кнопку, график и т. д.). Зависимость от других модулей:

```
C:\book>jdeps --module-path C:\JavaSE\JavaFX\12\lib -s
           -m javafx.controls
javafx.controls -> java.base
javafx.controls -> javafx.base
javafx.controls -> javafx.graphics
```

- ❑ **javafx.media** — позволяет работать с аудио и видео. Зависимость от других модулей:

```
C:\book>jdeps --module-path C:\JavaSE\JavaFX\12\lib -s
           -m javafx.media
javafx.media -> java.base
javafx.media -> javafx.base
javafx.media -> javafx.graphics
```

- **javafx.web** — содержит классы, предназначенные для работы с Интернетом. Зависимость от других модулей:

```
C:\book>jdeps --module-path C:\JavaSE\JavaFX\12\lib -s
-m javafx.web
javafx.web -> java.base
javafx.web -> java.desktop
javafx.web -> java.xml
javafx.web -> javafx.base
javafx.web -> javafx.controls
javafx.web -> javafx.graphics
javafx.web -> javafx.media
javafx.web -> jdk.jsobject
javafx.web -> jdk.xml.dom
```

- **javafx.fxml** — позволяет использовать язык разметки FXML. Зависимость от других модулей:

```
C:\book>jdeps --module-path C:\JavaSE\JavaFX\12\lib -s
-m javafx.fxml
javafx.fxml -> java.base
javafx.fxml -> java.scripting
javafx.fxml -> java.xml
javafx.fxml -> javafx.base
javafx.fxml -> javafx.graphics
```

При работе с этим модулем следует учитывать, что используется рефлексия. Если вы создаете модульное приложение, то в файле `module-info.java` нужно объявить пакет с классом открытым (с помощью ключевого слова `opens`), иначе получите исключение:

```
opens <Пакет>;
```

Если вы не хотите открывать пакет для всех, то после ключевого слова `to` нужно указать для каких модулей пакет является открытым:

```
opens <Пакет> to <Модули через запятую>;
```

- **javafx.swing** — позволяет встроить компоненты Swing внутрь JavaFX-приложения и наоборот. Зависимость от других модулей:

```
C:\book>jdeps --module-path C:\JavaSE\JavaFX\12\lib -s
-m javafx.swing
javafx.swing -> java.base
javafx.swing -> java.datatransfer
javafx.swing -> java.desktop
javafx.swing -> javafx.base
javafx.swing -> javafx.graphics
javafx.swing -> jdk.unsupported.desktop
```

1.3. Создание и запуск JavaFX-приложения из командной строки

Давайте попробуем создать наше первое JavaFX-приложение для запуска из командной строки. Вначале создаем следующую структуру каталогов: C:\book\src\TestJavaFX\application. Затем добавляем в каталог application файл Main.java с кодом из листинга 1.1.

Для создания файла с программой можно воспользоваться любым текстовым редактором, например редактором Notepad++. Файл с программой должен быть сохранен в кодировке UTF-8. Мы будем пользоваться именно этой кодировкой, т. к. она позволяет хранить любые символы. Работая с редактором Notepad++, убедитесь, что в меню **Кодировки** установлен флажок **Кодировать в UTF-8 (без BOM)**. Если это не так, то в меню **Кодировки** следует выбрать пункт **Преобразовать в UTF-8 без BOM**.

Листинг 1.1. Первое JavaFX-приложение

```
package application;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.layout.BorderPane;
import javafx.scene.text.Font;
import javafx.stage.Stage;

public class Main extends Application {

    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) throws Exception {
        BorderPane root = new BorderPane();
        Label text = new Label("Привет, мир!");
        text.setFont(new Font(32));
        root.setCenter(text);
        Scene scene = new Scene(root, 400, 150);

        stage.setTitle("Заголовок окна");
        stage.setScene(scene);
        stage.show();
    }
}
```

Скомпилируем этот файл в командной строке (команда состоит не из трех строк, как показано здесь, а из одной общей длинной строки):

```
C:\book>javac -encoding utf-8 --module-path C:\JavaSE\JavaFX\12\lib
--add-modules javafx.controls -d C:\book\bin\TestJavaFX
C:\book\src\TestJavaFX\application\*.java
```

В результате компиляции в каталоге C:\book\bin\TestJavaFX\application будет создан файл Main.class с байт-кодом. Попробуем запустить наше первое JavaFX-приложение из командной строки:

```
C:\book>java --module-path C:\JavaSE\JavaFX\12\lib
--add-modules javafx.controls
-cp C:\book\bin\TestJavaFX application.Main
```

В результате отобразится окно, показанное на рис. 1.1. Окно можно перемещать по экрану, изменять его размеры, взявшись мышью за границы, а также свернуть или развернуть его с помощью соответствующих кнопок в заголовке окна. Для закрытия окна следует воспользоваться кнопкой **Заккрыть** в заголовке окна, кнопкой **Заккрыть** в меню окна (щелкните левой кнопкой мыши на значке в заголовке окна, чтобы отобразить меню) или нажать комбинацию клавиш <Alt>+<F4>.

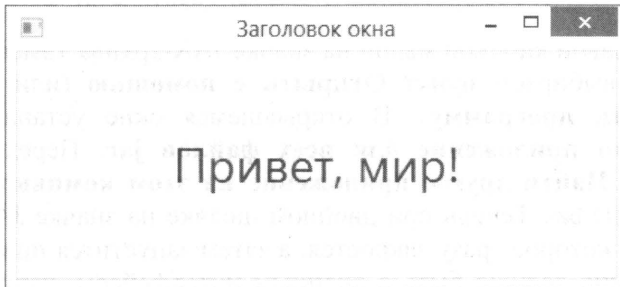


Рис. 1.1. Результат выполнения программы из листинга 1.1

В приведенных командах мы указали следующие опции:

- ❑ `-encoding` — задает кодировку исходных файлов;
- ❑ `--module-path` — указывает путь поиска модулей;
- ❑ `--add-modules` — добавляет модули. Для компиляции и запуска нашего приложения нужны модули `java.base`, `javafx.base`, `javafx.graphics` и `javafx.controls`. Так как модуль `javafx.controls` зависит от всех остальных модулей, достаточно добавить только этот модуль. Вместо указания модулей через запятую можно задать значения `ALL-DEFAULT`, `ALL-SYSTEM` и `ALL-MODULE-PATH`;
- ❑ `-cp` — задает путь поиска классов;
- ❑ `-d` — указывает каталог, в который будут записаны скомпилированные файлы.

Теперь создадим исполняемый JAR-архив:

```
C:\book>jar cvfe C:\book\TestJavaFX.jar application.Main
-c C:\book\bin\TestJavaFX\ .
```

```
added manifest
adding: application/(in = 0) (out= 0)(stored 0%)
adding: application/Main.class(in = 1036) (out= 644) (deflated 37%)
```

В результате в каталоге C:\book будет создан файл TestJavaFX.jar. Вначале попробуем запустить его из командной строки:

```
C:\book>java --module-path C:\JavaSE\JavaFX\12\lib
--add-modules javafx.controls -jar C:\book\TestJavaFX.jar
```

Как видно из приведенных примеров, всегда нужно указывать путь к библиотеке JavaFX и добавлять нужные модули. Если мы сейчас попробуем запустить программу двойным щелчком мыши на значке JAR-архива, то ничего не получится, т. к. путь поиска модулей не указан, и модули не добавлены. Давайте сделаем так, чтобы исполняемые JAR-архивы с JavaFX-приложениями запускались путем двойного щелчка на значке архива. Для этого сначала в каталоге C:\OpenJDK\jdk-11 создаем файл runJAR11.bat (кодировка файла должна быть windows-866) со следующим содержимым (вторая команда состоит из одной длинной строки, а не из двух строк, как здесь показано):

```
@echo off
start C:\OpenJDK\jdk-11\bin\javaw.exe --module-path
C:\JavaSE\JavaFX\12\lib --add-modules ALL-MODULE-PATH -jar %1
```

Далее щелкаем правой кнопкой мыши на значке JAR-архива TestJavaFX.jar и из контекстного меню выбираем пункт **Открыть с помощью** (или **Открыть с помощью | Выбрать программу**). В открывшемся окне устанавливаем флажок **Использовать это приложение для всех файлов jar**. Переходим по ссылке **Дополнительно | Найти другое приложение на этом компьютере** и выбираем программу runJAR11.bat. Теперь при двойном щелчке на значке JAR-архива откроется черное окно, которое сразу закроется, а затем запустится приложение из JAR-архива. Приложение должно быть с оконным интерфейсом, например, как в листинге 1.1, иначе результат выполнения вы не увидите.

После этих действий можно запустить JavaFX-приложение с помощью двойного щелчка мышью на значке JAR-архива TestJavaFX.jar. В результате откроется окно, показанное на рис. 1.1.

Вместо обычного приложения мы можем сделать модульное приложение. Для этого в каталоге C:\book\src\TestJavaFX создаем файл module-info.java со следующим содержимым:

```
module TestJavaFX {
    exports application;
    requires javafx.controls;
}
```

Скомпилируем программу и создадим модульный JAR-архив:

```
C:\book>javac -encoding utf-8 --module-path C:\JavaSE\JavaFX\12\lib
-d C:\book\bin\TestJavaFX
C:\book\src\TestJavaFX\module-info.java
C:\book\src\TestJavaFX\application\*.java
```

```
C:\book>jar cvf C:\book\modules\TestJavaFX.jar
        -C C:\book\bin\TestJavaFX\ .
added manifest
added module-info: module-info.class
adding: application/(in = 0) (out= 0) (stored 0%)
adding: application/Main.class(in = 1036) (out= 644) (deflated 37%)
```

В результате в каталоге `C:\book\modules` будет создан модульный JAR-архив `TestJavaFX.jar`. Запустим его из командной строки:

```
C:\book>java --module-path C:\JavaSE\JavaFX\12\lib;C:\book\modules
        --module TestJavaFX/application.Main
```

В этой команде мы добавили в путь поиска модулей и каталог с JavaFX-модулями, и каталог с нашим модулем через точку с запятой. Обратите внимание: при компиляции и запуске модуля не нужно указывать опцию `--add-modules`, т. к. все зависимости от других модулей указываются в файле `module-info`.

Еще раз обращаю ваше внимание на то, что при использовании FXML в файле `module-info` нужно указывать инструкцию `opens`, а не `exports`, т. к. используется рефлексия:

```
module TestJavaFX {
    opens application;
    requires javafx.fxml;
    requires javafx.controls;
}
```

В этом примере пакет `application` открыт для всех модулей. Чтобы доступ был только для некоторых модулей, следует указать эти модули через запятую после ключевого слова `to`:

```
module TestJavaFX {
    opens application to javafx.fxml, javafx.graphics, javafx.controls;
    requires javafx.fxml;
    requires javafx.controls;
}
```

1.4. Установка и настройка редактора Eclipse

Набирать текст программы в обычном текстовом редакторе, а затем компилировать программу и запускать ее в командной строке, не очень удобно. Поэтому в дальнейшем мы воспользуемся специализированным редактором Eclipse, в котором подсвечивается код программы, выводятся подсказки и справочная информация, а также имеется возможность скомпилировать программу и сразу запустить ее нажатием всего одной кнопки.

Для загрузки редактора Eclipse переходим на страницу <http://www.eclipse.org/downloads/packages/> и скачиваем архив с программой из раздела **Eclipse IDE for Java Developers**. Распаковываем скачанный архив и копируем каталог `eclipse` с фай-

лами редактора в каталог C:\JavaSE. Редактор не нуждается в установке, поэтому просто переходим в каталог C:\JavaSE\eclipse и запускаем файл eclipse.exe.

Eclipse хорошо работает на Java 11, но некоторые дополнительные плагины могут работать некорректно. Если после установки плагина редактор не запускается, то нужно явным образом указать путь к более старой версии Java в настройках редактора. Чтобы узнать предпочтительную для редактора версию Java, открываем файл C:\JavaSE\eclipse\eclipse.ini и смотрим значение опции `requiredJavaVersion`. В моем случае опция имеет такое значение:

```
-Dosgi.requiredJavaVersion=1.8
```

Значение 1.8 означает, что нужна Java 8, а мы используем Java 11. Устанавливаем на компьютер требуемую версию Java и указываем путь к ней с помощью опции `-vm`. Самый простой способ: создать ярлык для eclipse.exe, отобразить **Свойства** и в поле **Объект** указать такое значение (замените фрагмент <Путь к JRE> реальным значением):

```
C:\JavaSE\eclipse\eclipse.exe -vm "<Путь к JRE>\bin\server\jvm.dll"
```

Пример:

```
C:\JavaSE\eclipse\eclipse.exe  
-vm "C:\Program Files\Java\jre1.8.0_77\bin\server\jvm.dll"
```

ПРИМЕЧАНИЕ

Все способы указания значения подробно описаны на странице <http://wiki.eclipse.org/Eclipse.ini>.

При запуске редактор попросит указать каталог с рабочим пространством (рис. 1.2). Указываем C:\JavaSE\JavaFX\projects и нажимаем кнопку **Launch**.

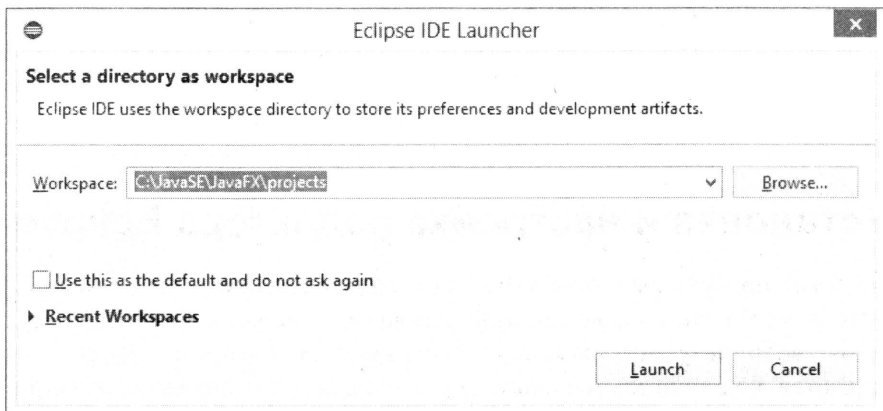


Рис. 1.2. Указание каталога с рабочим пространством при запуске редактора

Редактор Eclipse использует встроенный компилятор, а не внешний компилятор из каталога с установленным JDK. Причем доступны компиляторы практически всех версий. Для выбора версии компилятора в меню **Window** выбираем пункт

Preferences. В открывшемся окне переходим на вкладку **Java | Compiler** (рис. 1.3) и выбираем нужную версию из списка **Compiler compliance level**. Мы будем использовать Java 11, поэтому в этом списке должен быть выбран пункт **11**.

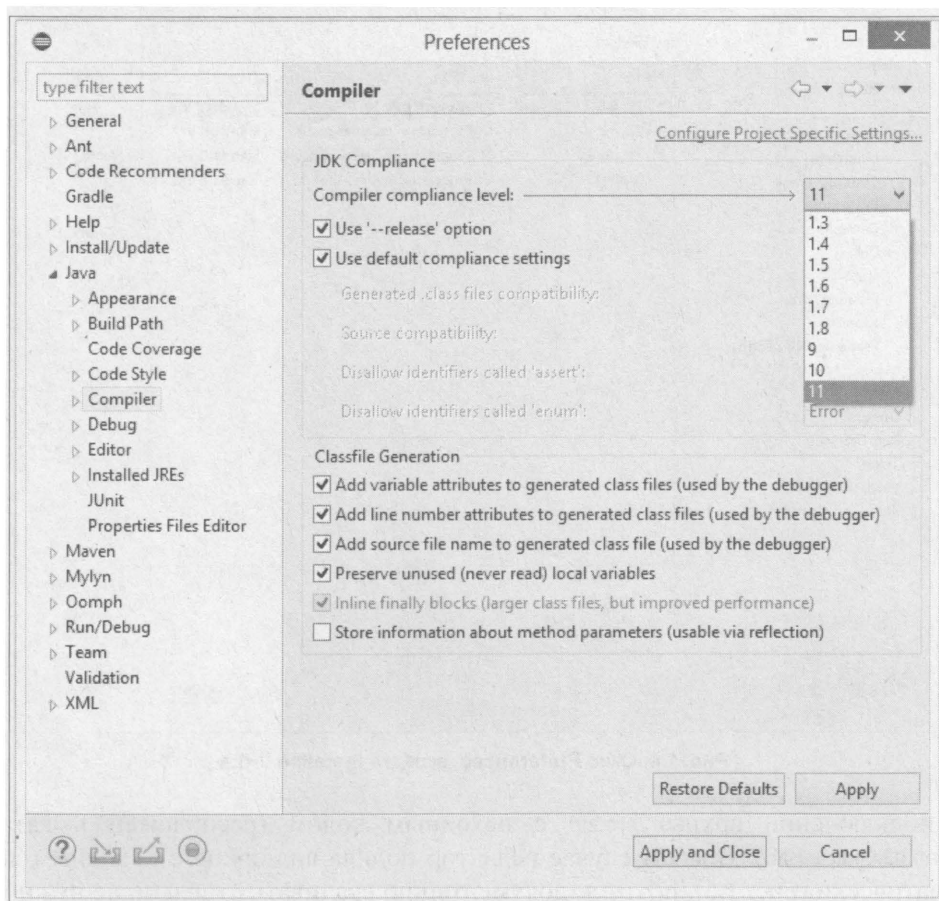


Рис. 1.3. Окно **Preferences**: вкладка **Compiler**

Если в списке **Compiler compliance level** нет пункта **11**, то в меню **Help** выбираем пункт **Eclipse Marketplace**. В открывшемся окне в поле **Find** вводим **Java 11** и нажимаем клавишу **<Enter>**. В списке находим пункт **Java 11 support for Eclipse** и нажимаем кнопку **Install**. После установки перезапускаем редактор, чтобы изменения вступили в силу.

Теперь добавим Java 11 в настройки редактора. Для этого в меню **Window** выбираем пункт **Preferences**. В открывшемся окне переходим на вкладку **Java | Installed JREs** (рис. 1.4). Если в списке нет пункта **jdk-11**, то нажимаем кнопку **Search** и указываем каталог **C:\OpenJDK**. Все найденные версии отобразятся в списке. Устанавливаем флажок **jdk-11** и сохраняем настройки.

Редактор позволяет получить справку при наведении указателя мыши на какое-либо название. Это будет работать при активном подключении к Интернету или

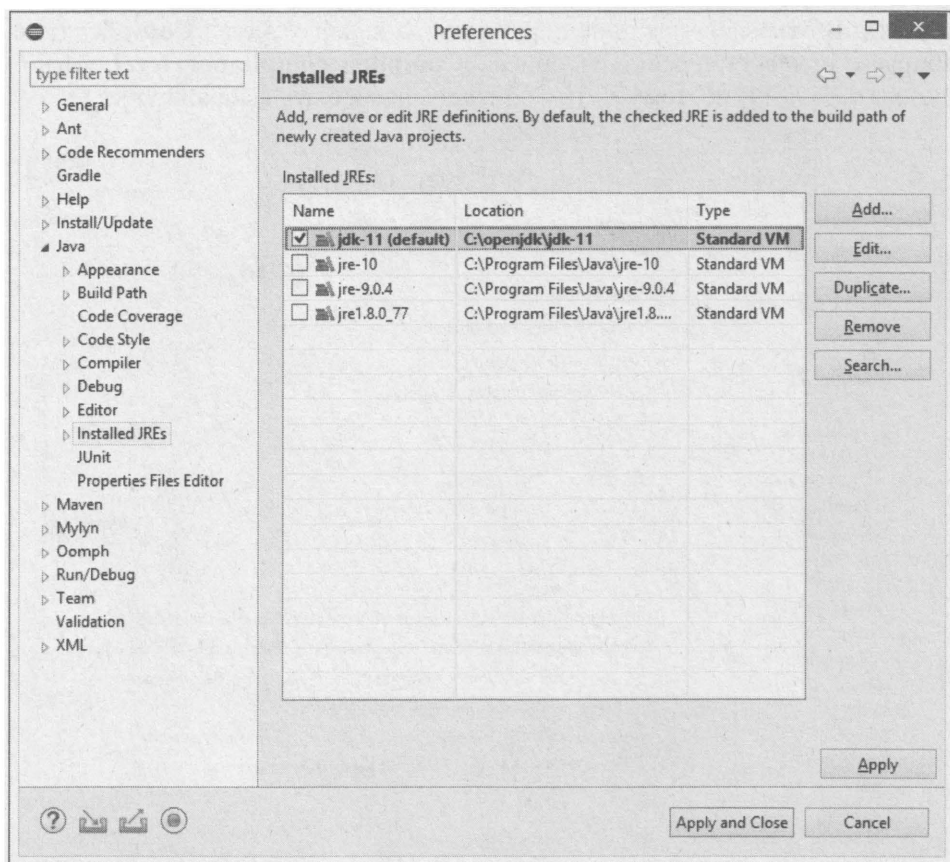


Рис. 1.4. Окно Preferences: вкладка Installed JREs

при подключении архива src.zip с исходным кодом (расположен в каталоге C:\OpenJDK\jdk-11\lib). В моем случае редактор подключил архив с исходным кодом автоматически (рис. 1.5), поэтому документацию скачивать отдельно не нужно, она входит в состав исходного кода.

Работать мы будем с кодировкой UTF-8, поэтому давайте настроим кодировку файлов по умолчанию. Для этого в меню **Window** выбираем пункт **Preferences**. В открывшемся окне переходим на вкладку **General | Workspace** (рис. 1.6). В группе **Text file encoding** устанавливаем флажок **Other** и из списка выбираем кодировку UTF-8. Сохраняем изменения. Если необходимо изменить кодировку уже открытого файла, в меню **Edit** выбираем пункт **Set Encoding**.

По умолчанию редактор вместо пробелов вставляет символы табуляции. Нас это не устраивает. Давайте изменим настройку форматирования кода. Для этого в меню **Window** выбираем пункт **Preferences**. В открывшемся окне переходим на вкладку **Java | Code Style | Formatter** (рис. 1.7). Нажимаем кнопку **New**. В открывшемся окне (рис. 1.8) в поле **Profile name** вводим название стиля, например **MyStyle**, а из списка выбираем пункт **Eclipse [built-in]**. Нажимаем кнопку **OK**. Откроется окно, в котором можно изменить настройки нашего стиля. На вкладке **Indentation**

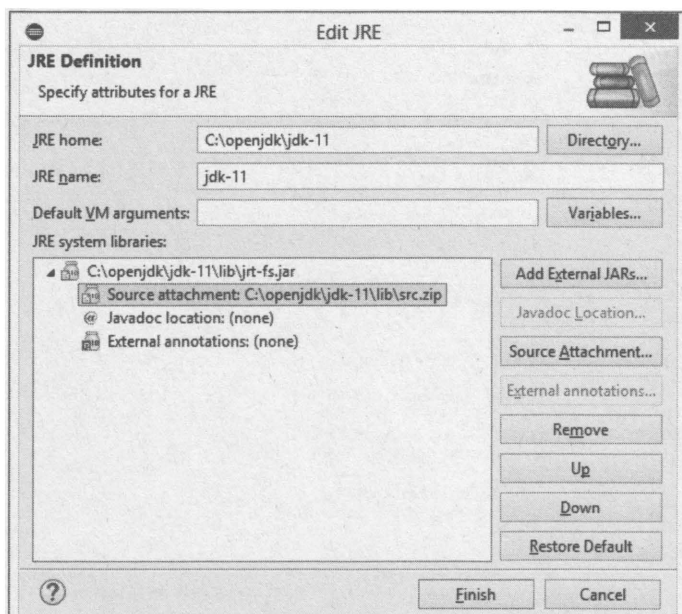


Рис. 1.5. Подключение архива с исходным кодом в окне Edit JRE

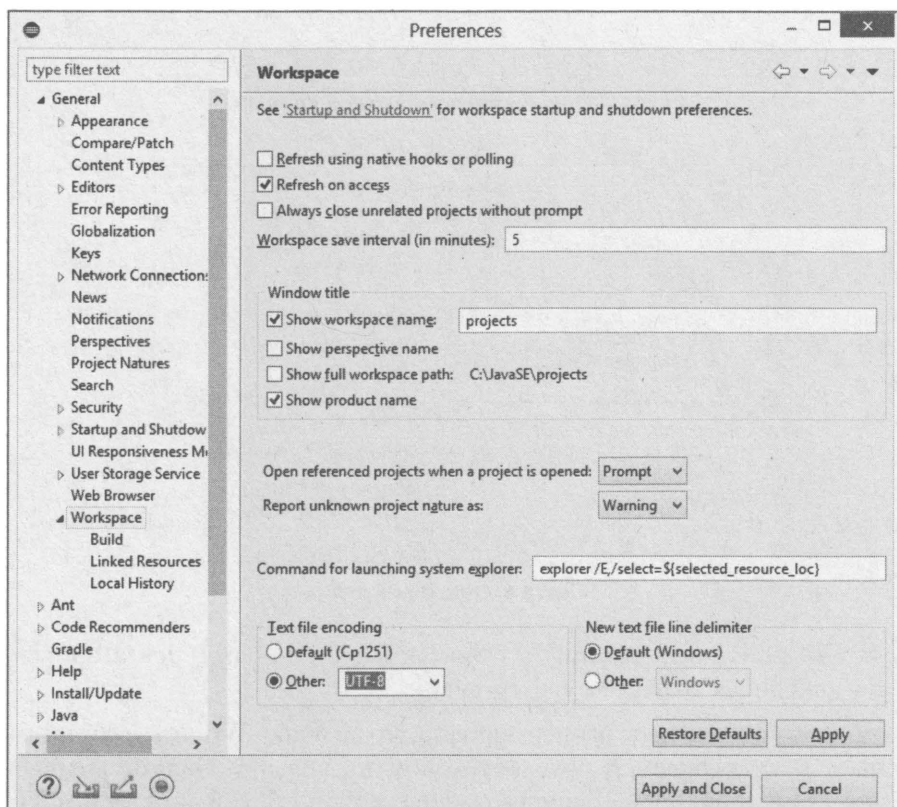


Рис. 1.6. Указание кодировки файлов

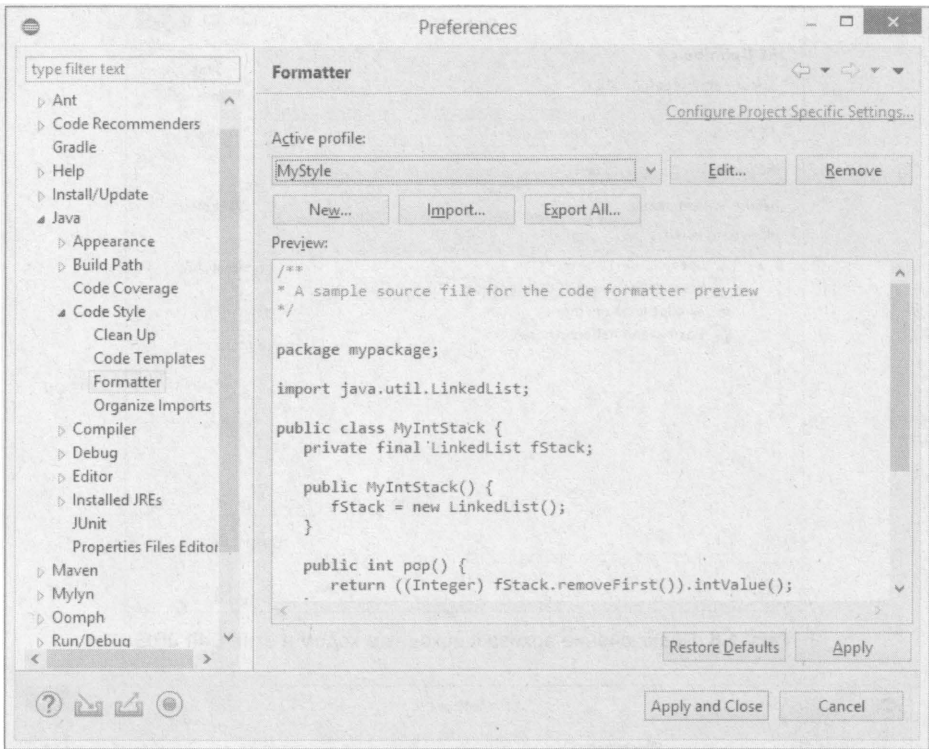


Рис. 1.7. Окно Preferences: вкладка Formatter

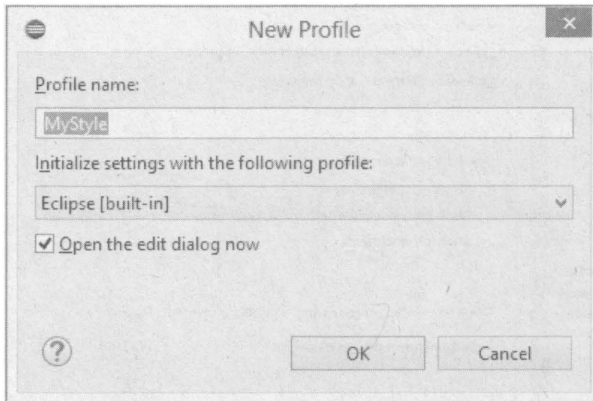


Рис. 1.8. Окно New Profile

из списка **Tab policy** выбираем пункт **Spaces only**, а в поля **Indentation size** и **Tab size** вводим число 3. Сохраняем все изменения.

Если необходимо изменить размер шрифта, то в меню **Window** выбираем пункт **Preferences**. В открывшемся окне переходим на вкладку **General | Appearance | Colors and Fonts** (рис. 1.9). Из списка выбираем пункт **Java | Java Editor Text Font** и нажимаем кнопку **Edit**.

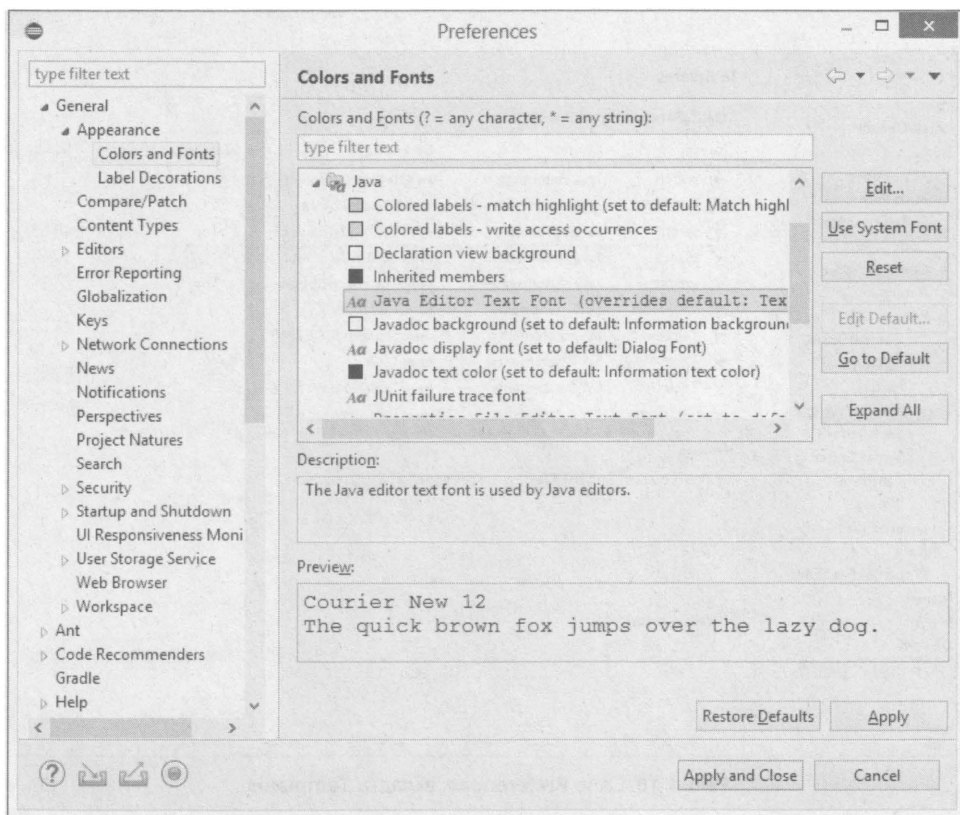


Рис. 1.9. Окно Preferences: вкладка Colors and Fonts

Если после названия объекта вставить точку, то редактор отобразит список методов и свойств. Если при этом набирать первые буквы, то содержимое списка сократится. Достаточно выбрать метод из списка, и его название будет вставлено в код. Каким бы длинным ни было название метода, мы можем его вставить в код без ошибок в его названии и очень быстро.

Редактор позволяет также закончить частично набранное слово. Для этого достаточно нажать комбинацию клавиш <Ctrl>+<Пробел>. В результате редактор автоматически закончит слово или предложит список с возможными вариантами. Причем этот способ отобразит еще и названия шаблонов кода. При выборе шаблона будет вставлено не просто слово, а целый фрагмент кода. Например, введите слово `sysout` и нажмите комбинацию клавиш <Ctrl>+<Пробел>. В результате будет вставлен следующий код:

```
System.out.println();
```

Это очень удобно и экономит много времени. Чтобы увидеть все доступные шаблоны, в меню **Window** выбираем пункт **Preferences**. В открывшемся окне переходим на вкладку **Java | Editor | Templates** (рис. 1.10). С помощью этой вкладки можно не только посмотреть все шаблоны, но и отредактировать их, а также создать свой собственный шаблон.

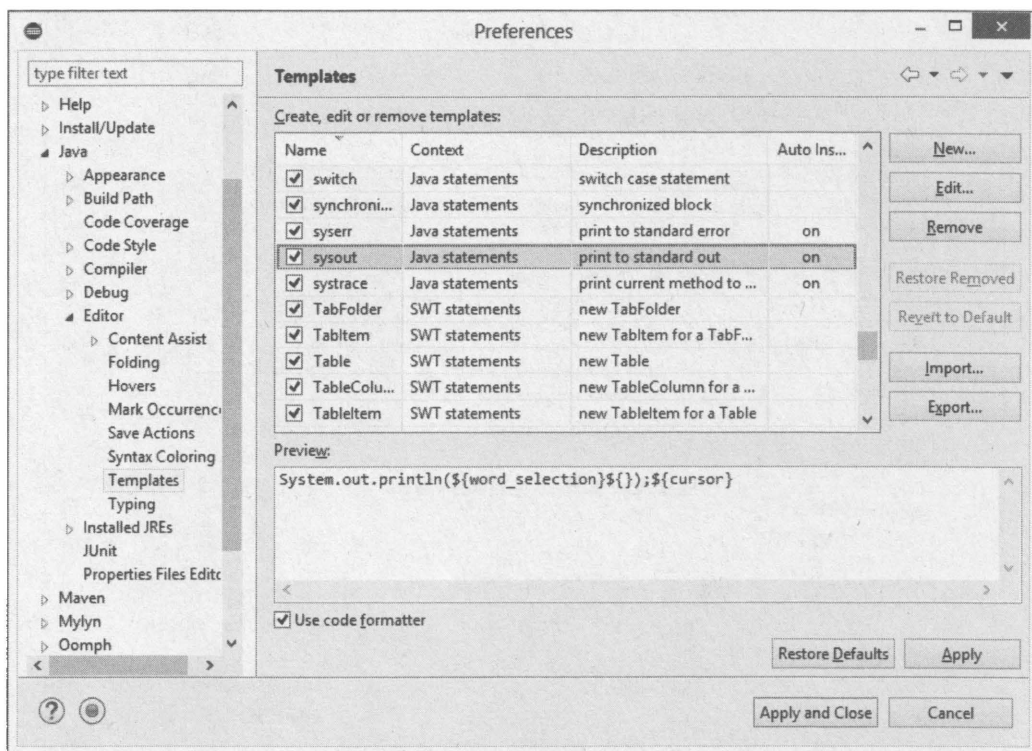


Рис. 1.10. Окно Preferences: вкладка Templates

1.5. Установка модуля e(fx)clipse

Чтобы создавать полноценные JavaFX-приложения с поддержкой FXML и CSS, нужно установить модуль для редактора Eclipse под названием **e(fx)clipse**. Для установки модуля (потребуется подключение к Интернету) в меню **Help** выбираем пункт **Eclipse Marketplace**. В открывшемся окне (рис. 1.11) в поле для поиска вводим **e(fx)clipse**. Нажимаем клавишу <Enter>. Находим пункт в списке и нажимаем кнопку **Install**. Отобразится окно (рис. 1.12), в котором будут указаны устанавливаемые компоненты. Соглашаемся с лицензионным соглашением и нажимаем кнопку **Finish**. После установки перезапускаем редактор Eclipse, чтобы изменения вступили в силу.

ПРИМЕЧАНИЕ

Подробную информацию о модуле **e(fx)clipse** можно получить на странице <http://www.eclipse.org/efxclipse/index.html>.



Рис. 1.11. Окно Eclipse Marketplace

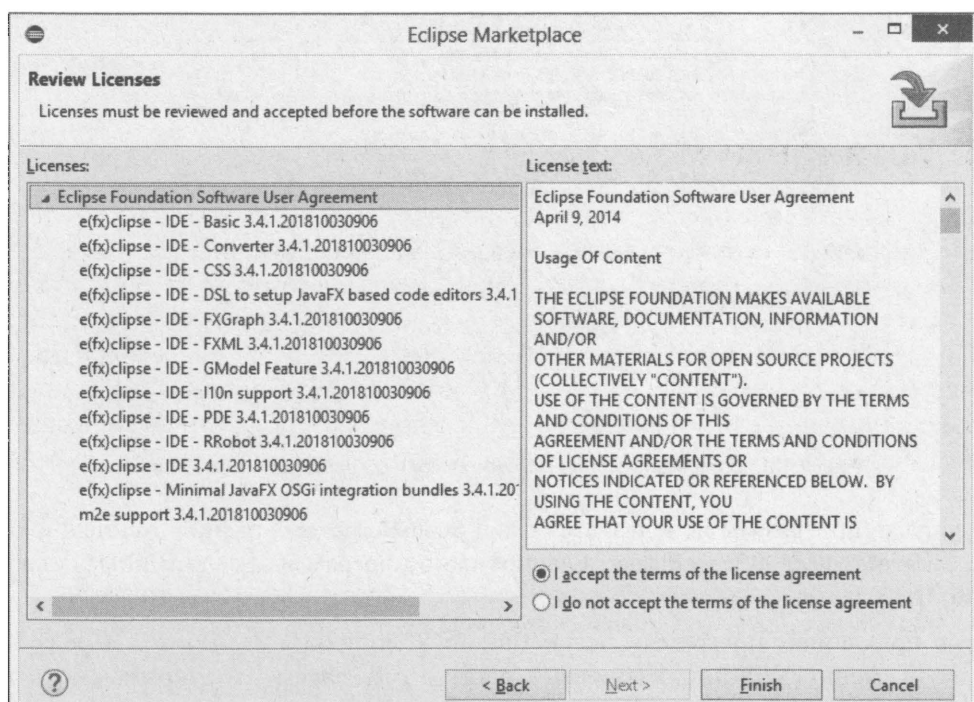


Рис. 1.12. Список устанавливаемых компонентов модуля e(fx)clipse

1.6. Установка программы Scene Builder

Если вы ранее пользовались Visual Studio или Delphi, то вспомните, что размещение компонентов на форме производили с помощью мыши. Щелкали левой кнопкой мыши на соответствующей кнопке на панели инструментов и перетаскивали компонент на форму. Далее с помощью инспектора свойств производили настройку значений некоторых свойств, а остальные свойства получали значения по умолчанию. При этом весь код генерировался автоматически. Произвести аналогичную операцию в JavaFX позволяет программа Scene Builder.

Скачать дистрибутив программы Scene Builder можно со страницы <https://gluonhq.com/products/scene-builder/>. Находим раздел **Download Scene Builder for Java 10** и нажимаем кнопку **Download** у пункта **Windows Installer**. Скачиваем программу установки и запускаем ее. В открывшемся окне (рис. 1.13) соглашаемся с лицензией и нажимаем кнопку **Next**. В следующем окне (рис. 1.14) указываем путь к каталогу C:\JavaSE\JavaFX\SceneBuilder и нажимаем кнопку **Next** для начала установки.

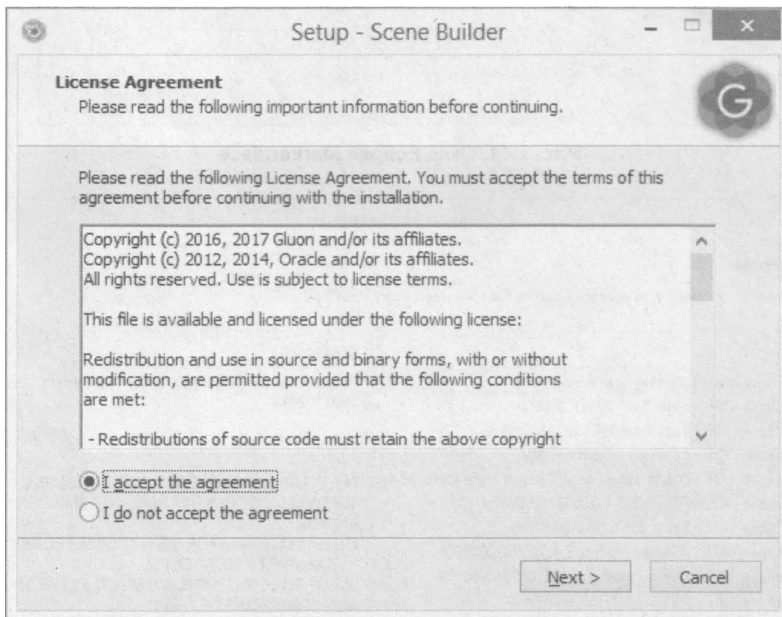


Рис. 1.13. Установка программы Scene Builder: соглашаемся с лицензией

Для запуска программы используется файл SceneBuilder.exe, расположенный в каталоге C:\JavaSE\JavaFX\SceneBuilder. Главное окно программы Scene Builder показано на рис. 1.15.

Теперь подключим программу Scene Builder к модулю `e(fx)clipse`. В редакторе Eclipse в меню **Window** выбираем пункт **Preferences**. В открывшемся окне (рис. 1.16) из списка слева выбираем пункт **JavaFX** и на вкладке справа нажимаем кнопку **Browse**. Находим файл C:\JavaSE\JavaFX\SceneBuilder\SceneBuilder.exe. В итоге

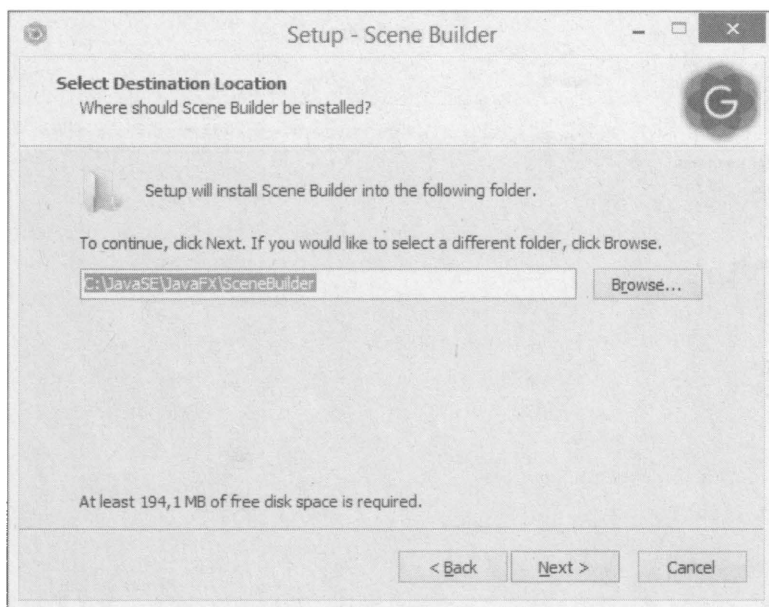


Рис. 1.14. Установка программы Scene Builder: указываем путь к каталогу с программой



Рис. 1.15. Главное окно программы Scene Builder

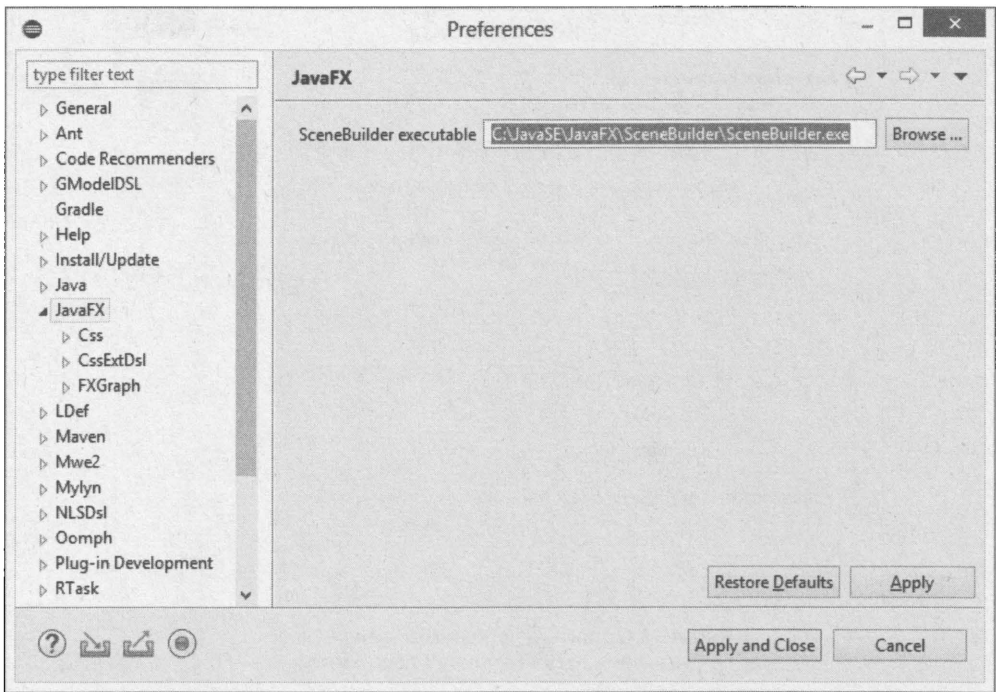


Рис. 1.16. Подключение программы Scene Builder к модулю e(fx)clipse

путь к программе отобразится в поле **SceneBuilder executable**. Сохраняем изменения.

1.7. Создание JavaFX-приложения в Eclipse. Способ 1

Создать JavaFX-приложение можно несколькими способами. Первый способ совпадает с созданием обычного Java-приложения. Давайте попробуем создать Java-приложение в редакторе Eclipse. Вначале создадим проект. Для этого в меню **File** выбираем пункт **New | Java Project**. В открывшемся окне (рис. 1.17) в поле **Project name** вводим `FX_CH01_01_Start`, выбираем версию JRE и нажимаем кнопку **Finish**. Если редактор предложит создать файл `module-info`, то отказываемся. Далее создадим пакет. Для этого в меню **File** выбираем пункт **New | Package**. В открывшемся окне (рис. 1.18) в поле **Name** вводим `application` и нажимаем кнопку **Finish**. Теперь можно добавить класс в пакет. Для этого в меню **File** выбираем пункт **New | Class**. В открывшемся окне (рис. 1.19) в поле **Name** вводим `Main`, в поле **Package** — `application`, а затем нажимаем кнопку **Finish**.

В результате в каталоге `C:\JavaSE\JavaFX\projects\FX_CH01_01_Start\src\application` будет создан файл `Main.java` и открыт на редактирование на отдельной вкладке. Добавляем в этот файл код из листинга 1.1.

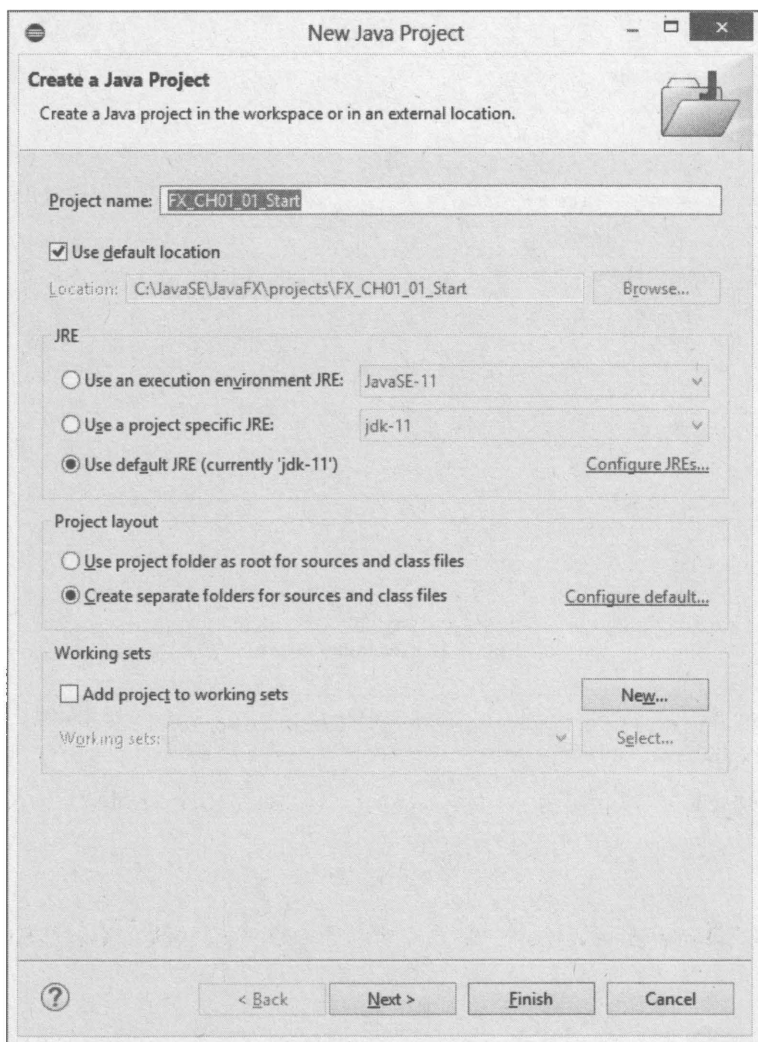


Рис. 1.17. Создание проекта

После вставки кода редактор подчеркнет красной волнистой линией все классы и пакеты, входящие в состав библиотеки JavaFX, т. к. пути к библиотеке редактор не знает. Чтобы подключить библиотеку к проекту, в окне **Package Explorer** щелкаем правой кнопкой мыши на названии проекта и из контекстного меню (рис. 1.20) выбираем пункт **Properties**. Запомните этот способ отображения свойств проекта, т. к. в дальнейшем мы будем просто говорить «отобразите свойства проекта» или «в свойствах проекта» без явного указания на то, как это сделать.

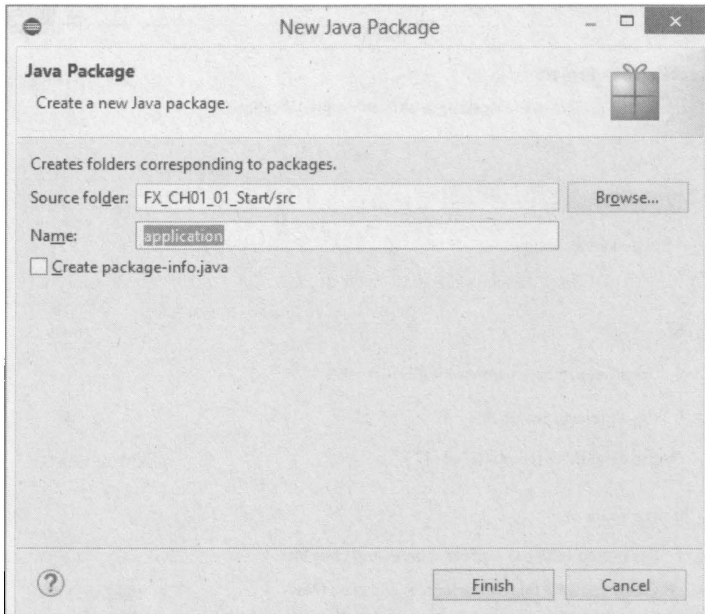


Рис. 1.18. Создание пакета

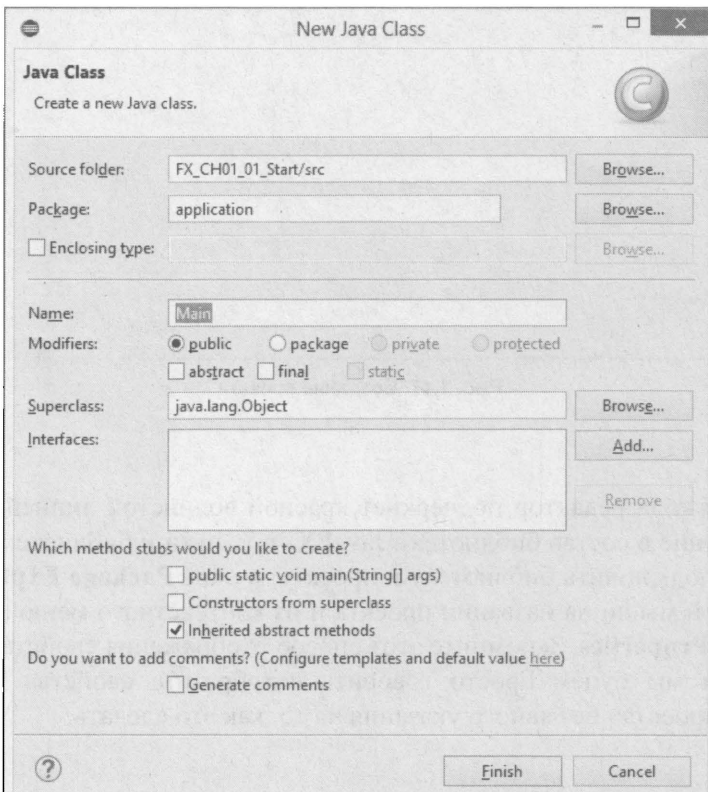


Рис. 1.19. Создание класса

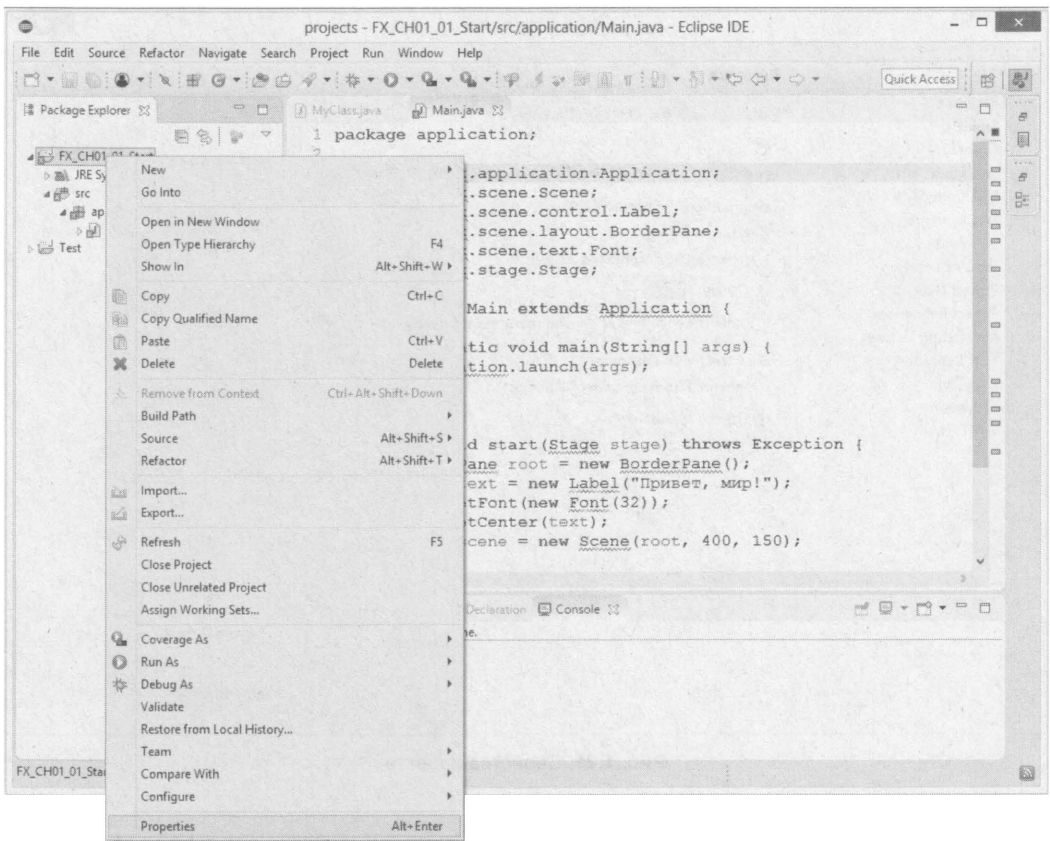


Рис. 1.20. Контекстное меню проекта

В результате откроется окно **Properties for FX_CH01_01_Start** (рис. 1.21). На вкладке **Resource** можно указать кодировку файлов проекта, а также символ-разделитель строк. По умолчанию эти два параметра наследуются из основных настроек редактора. Советую всегда явным образом задавать эти два параметра, иначе при смене кодировки по умолчанию редактор автоматически преобразует кодировку файлов проекта, что может привести к потере данных.

Для подключения библиотеки JavaFX в списке слева выбираем пункт **Java Build Path**. Далее отображаем вкладку **Libraries** и выделяем пункт **Modulepath**. Нажимаем кнопку **Add External JARs**. В открывшемся окне (рис. 1.22) переходим в каталог `C:\JavaSE\JavaFX\12\lib`, выделяем все модули (или только нужные для проекта) и нажимаем кнопку **Открыть**. Выбранные модули будут добавлены в список **Modulepath**. Чтобы иметь возможность получать справку при наведении указателя мыши на метод, класс и другие элементы, нужно подключить файл `C:\JavaSE\JavaFX\12\lib\src.zip` к каждому модулю (рис. 1.23). Нажимаем кнопку **Apple and Close** для сохранения свойств проекта. Все подчеркивания в коде должны исчезнуть, а при наведении указателя мыши на любой класс должна отобразиться справочная информация. Структура проекта с подключенными модулями в окне **Package Explorer** показана на рис. 1.24.

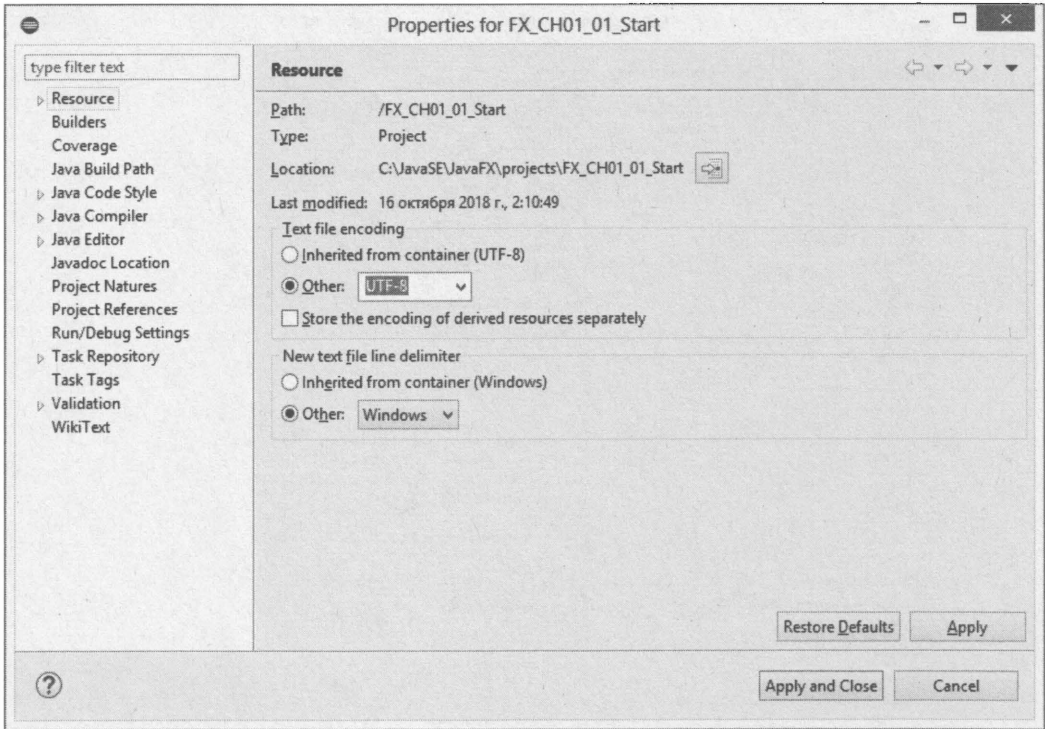


Рис. 1.21. Свойства проекта

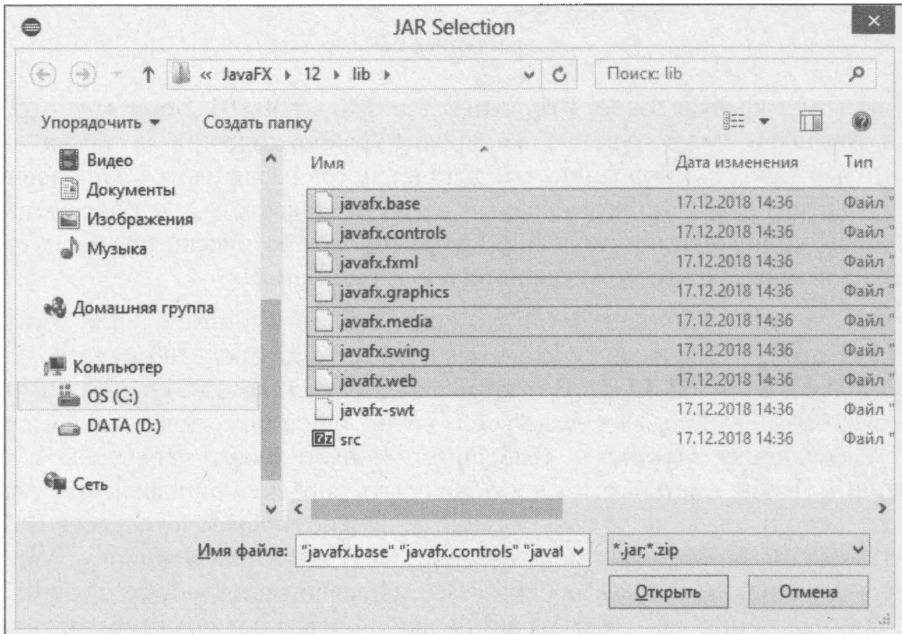


Рис. 1.22. Выбор подключаемых модулей

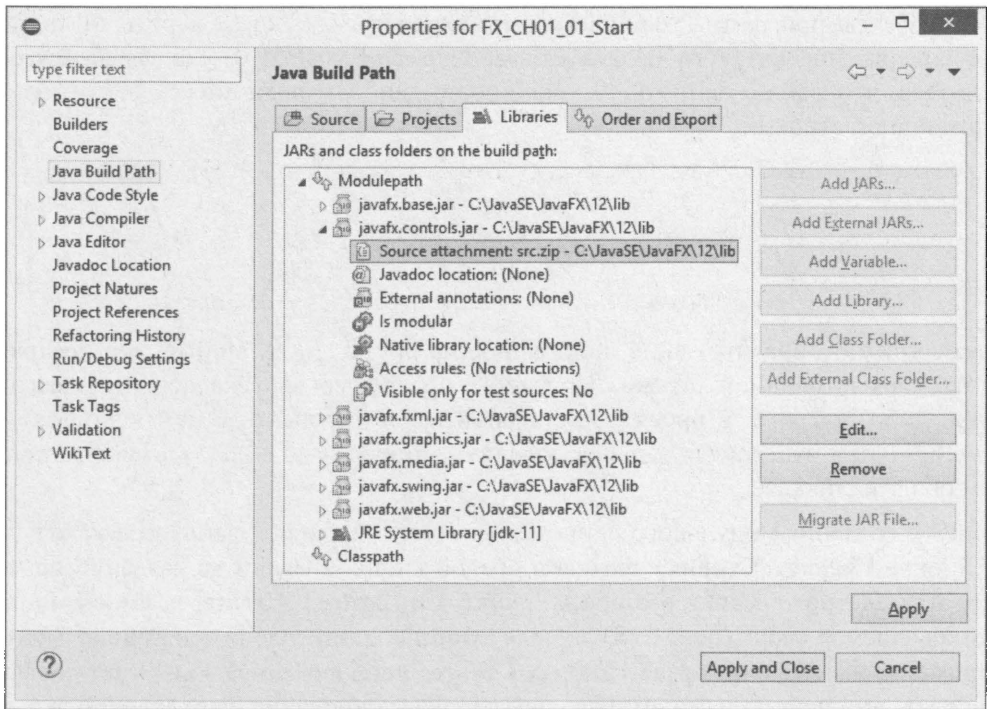


Рис. 1.23. Подключенные модули в списке Modulepath

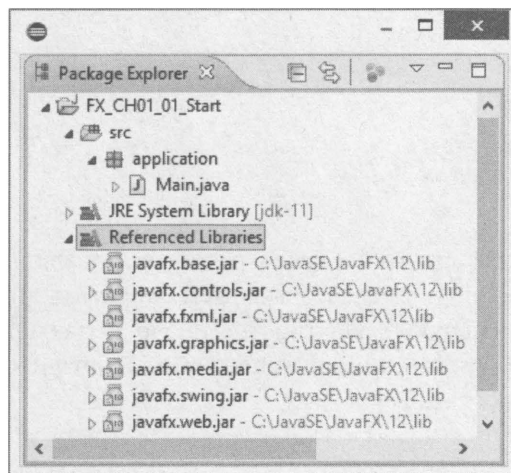


Рис. 1.24. Структура проекта с подключенными модулями в окне Package Explorer

Запуск JavaFX-приложения осуществляется точно так же, как и запуск обычного Java-приложения: в меню **Run** выбираем пункт **Run**, или нажимаем комбинацию клавиш <Ctrl>+<F11>, или нажимаем кнопку на панели инструментов. Однако сейчас после запуска мы получим следующее сообщение об ошибке:

```
Error: Could not find or load main class application.Main
Caused by: java.lang.NoClassDefFoundError: javafx/application/Application
```

На самом деле программа успешно скомпилировалась. Если не верите, то посмотрите содержимое каталога `C:\JavaSE\JavaFX\projects\FX_CH01_01_Start\bin\application` и убедитесь, что файл с байт-кодом успешно создан. Мы даже можем запустить его из командной строки:

```
C:\Users\Unicross>cd C:\JavaSE\JavaFX\projects\FX_CH01_01_Start\bin

C:\JavaSE\JavaFX\projects\FX_CH01_01_Start\bin>
    java --module-path C:\JavaSE\JavaFX\12\lib
        --add-modules javafx.controls application.Main
```

В результате отобразится окно, показанное на рис. 1.1. Как видите, все прекрасно работает из командной строки. Но чтобы программа запускалась из редактора, нужно либо добавить в проект файл `module-info` и прописать в нем используемые модули, либо в настройках запуска добавить опцию `--add-modules`. Давайте попробуем оба эти способа.

Начнем с создания модульного приложения. Чтобы добавить файл `module-info` в проект, в окне **Package Explorer** щелкаем правой кнопкой мыши на названии проекта и из контекстного меню выбираем пункт **Configure | Create module-info.java**. В открывшемся окне (рис. 1.25) в поле **Module name** вводим название модуля, например `mymodule`, и нажимаем кнопку **Create**. Файл `module-info.java` будет добавлен в пакет по умолчанию и станет доступен для редактирования на отдельной вкладке. В файл `module-info.java` редактор вставит следующий код:

```
module mymodule {
    exports application;

    requires javafx.base;
    requires javafx.controls;
    requires javafx.graphics;
}
```

Как видите, все зависимости от модулей редактор добавил самостоятельно. Зная зависимости модуля `javafx.controls`, мы можем сократить этот код и дополнительно определить транзитивную зависимость (иначе редактор будет подчеркивать класс `Stage` желтой волнистой линией и предлагать добавить перед классом аннотацию `@SuppressWarnings("exports")`):

```
module mymodule {
    exports application;
    requires transitive javafx.controls;
}
```

Если мы сейчас запустим приложение, то получим окно, показанное на рис. 1.1.

Теперь рассмотрим второй способ, которым будем пользоваться в дальнейших примерах. Сначала удаляем файл `module-info.java` из проекта. Для этого в окне **Package Explorer** щелкаем правой кнопкой мыши на названии файла и из контекстного меню выбираем пункт **Delete**. В открывшемся окне подтверждаем удаление

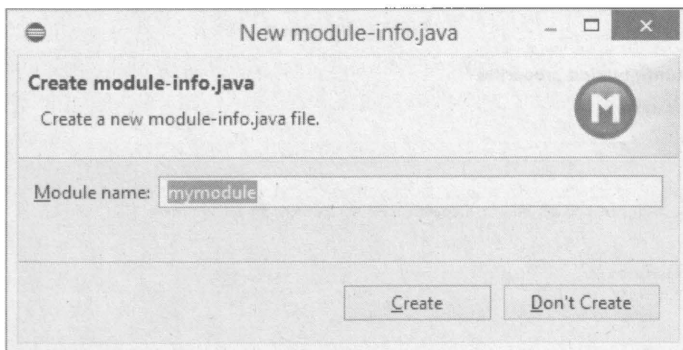


Рис. 1.25. Создание модуля

файла. Далее нам нужно добавить опцию `--add-modules` в командную строку для запуска приложения. Для этого открываем свойства проекта и в списке слева выбираем пункт **Run/Debug Settings** (рис. 1.26). Выделяем пункт с названием конфигурации запуска и нажимаем кнопку **Edit**. В открывшемся окне (рис. 1.27) переходим на вкладку **Arguments** и в поле **VM arguments** вводим следующую команду:

```
--add-modules ALL-MODULE-PATH
```

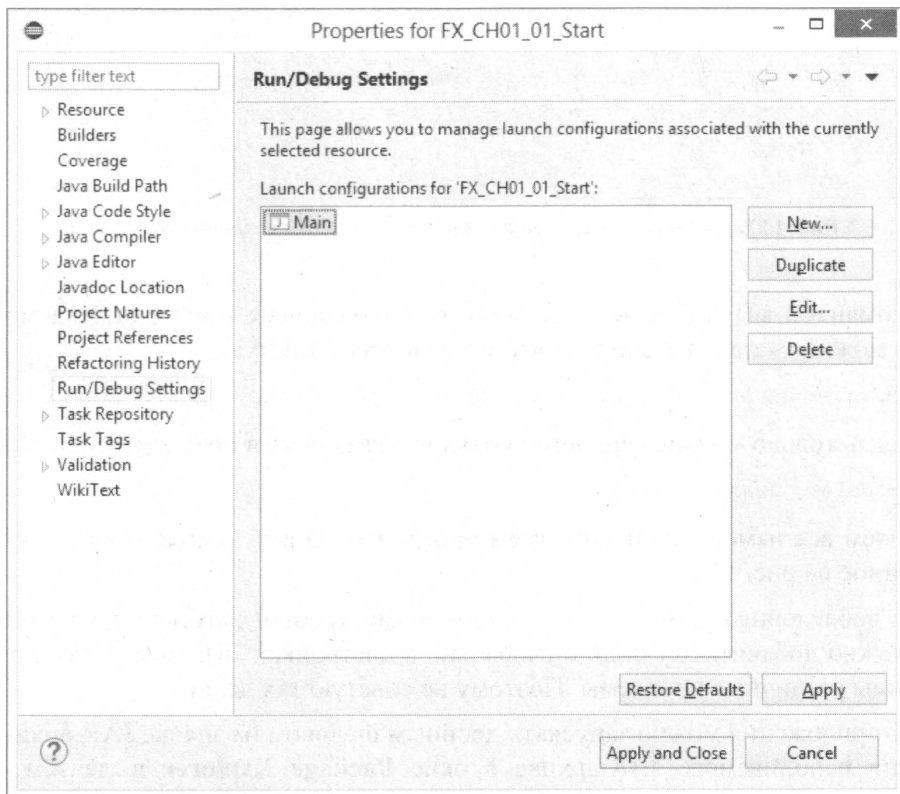


Рис. 1.26. Свойства проекта: вкладка Run/Debug Settings

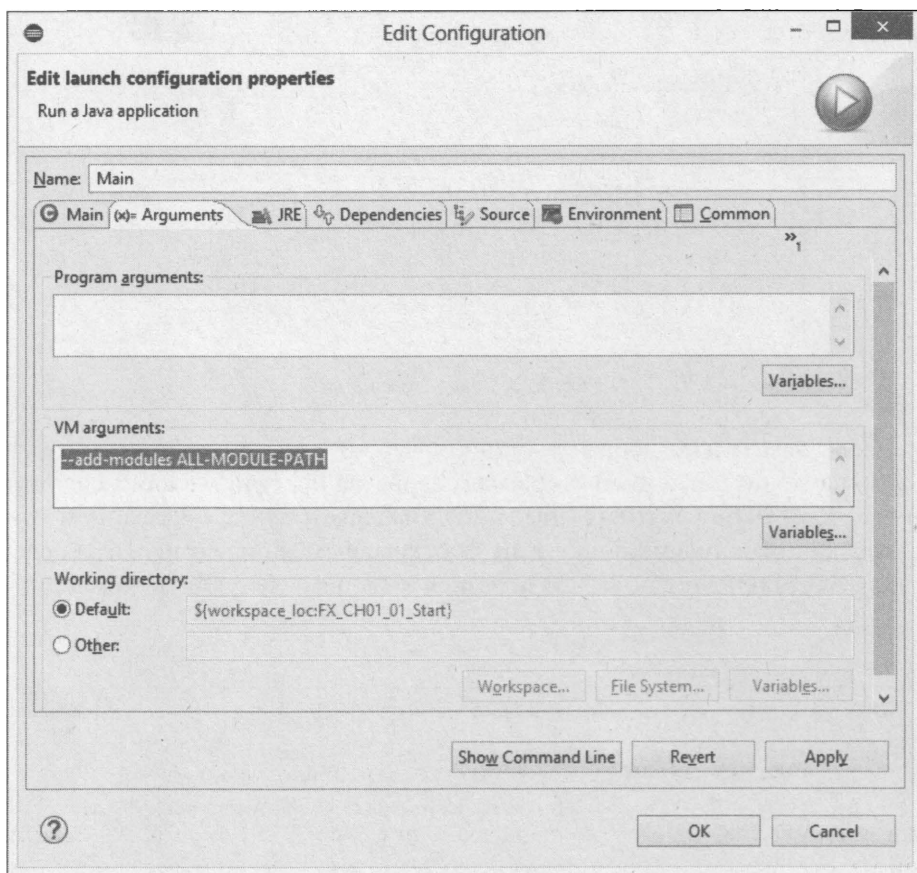


Рис. 1.27. Добавление модулей в конфигурацию запуска отдельного проекта

Этой командой мы добавляем все модули, находящиеся в пути поиска модулей, однако можно указать только нужные модули через запятую:

```
--add-modules javafx.base,javafx.graphics,javafx.controls
```

Или задать только модуль верхнего уровня из указанных в предыдущей команде:

```
--add-modules javafx.controls
```

Сохраняем все изменения и запускаем программу. В результате отобразится окно, показанное на рис. 1.1.

Вместо добавления опции `--add-modules` в конфигурацию запуска конкретного проекта, можно добавить эту опцию глобально в настройках JRE (рис. 1.28), но потом возможны различные проблемы. Поэтому не советую так делать.

JavaFX-приложения можно запускать двойным щелчком на значке JAR-архива. Для создания исполняемого JAR-архива в окне **Package Explorer** выделяем строку с пакетом, щелкаем на ней правой кнопкой мыши и из контекстного меню выбираем пункт **Export**. В открывшемся окне из списка выбираем пункт **Java | JAR file** и

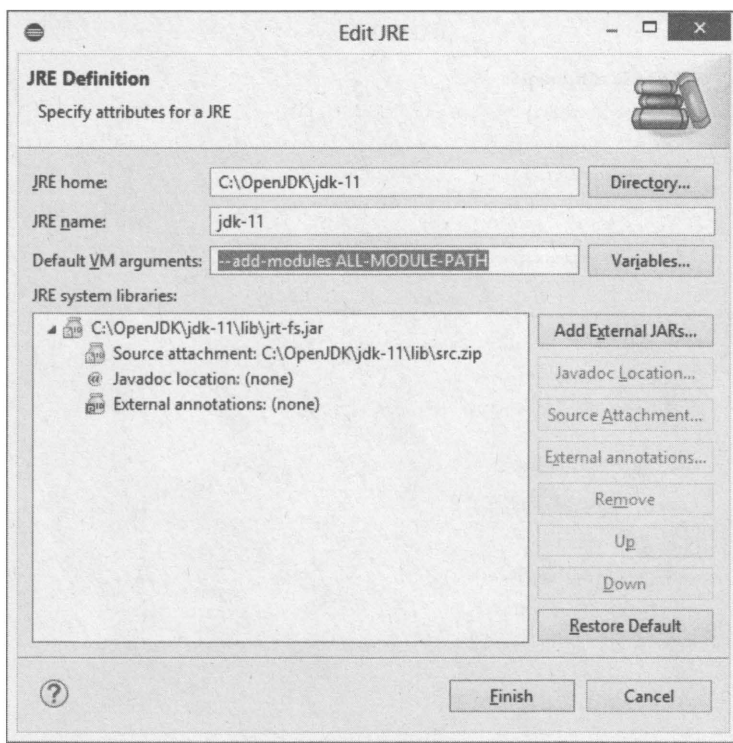


Рис. 1.28. Добавление модулей в глобальную конфигурацию запуска

нажимаем кнопку **Next**. В открывшемся окне (рис. 1.29) будет выделен только нужный пакет. Нажимаем кнопку **Browse** и выбираем местоположение и название JAR-архива, например `C:\book\TestJavaFX.jar`. Дважды нажимаем кнопку **Next**. В открывшемся окне (рис. 1.30) напротив поля **Main class** нажимаем кнопку **Browse** и выбираем класс `Main`. Нажимаем кнопку **Finish**. Находим созданный JAR-архив и запускаем его двойным щелчком левой кнопки мыши. Приложение запустится, если вы настроили запуск JAR-архивов с помощью программы `runJAR11.bat` (см. разд. 1.3).

Давайте рассмотрим код из листинга 1.1 построчно. Вначале с помощью инструкции `package` указывается, что класс `Main` принадлежит пакету `application`. Далее импортируются необходимые классы. Обратите внимание: все названия пакетов начинаются с префикса `javafx`. Важно следить, чтобы это было именно так. Например, при импорте класса `Button` можно случайно импортировать класс из пакета `java.awt` вместо пакета `javafx.scene.control`.

После импорта классов идет объявление класса `Main`. Ключевое слово `extends` после названия класса означает, что класс `Main` наследует класс `Application`. Внутри блока класса расположены два метода:

- `main()` — этот метод должен быть вам хорошо знаком. Именно этот метод вызывается при запуске Java-приложения. JavaFX-приложение также может иметь этот метод, а может и не иметь. Для запуска JavaFX-приложения из командной

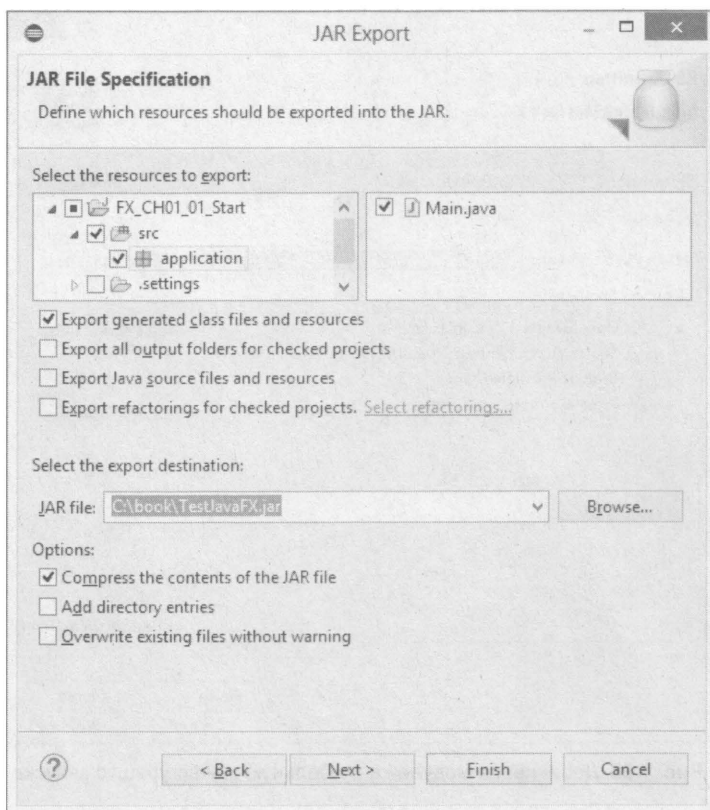


Рис. 1.29. Окно JAR Export

строки достаточно наследовать класс `Application` и переопределить метод `start()`, который и будет вызван при отсутствии метода `main()`. Если же метод `main()` существует, то вначале будет вызван именно он. Внутри метода `main()` вызывается статический метод `launch()` из класса `Application` и ему передаются параметры, введенные в командной строке:

- `start()` — этот метод из класса `Application`, переопределяемый в классе `Main`, является точкой входа в JavaFX-приложении. В качестве параметра метод принимает объект класса `Stage`, который описывает главное окно приложения.

Внутри метода `start()` вначале создается объект контейнера, внутрь которого будут добавляться различные компоненты. Класс `BorderPane` описывает контейнер, внутри которого компоненты могут быть прикреплены к одной из четырех сторон или помещены в центр. В нашем примере в центр контейнера добавляется только надпись (объект класса `Label`) с помощью метода `setCenter()`. Свойства надписи — в нашем примере размер шрифта — задаются с помощью метода `setFont()` и класса `Font`. Далее создается объект класса `Scene`, и конструктору класса передается объект контейнера и размеры окна. Затем с помощью метода `setTitle()` из класса `Stage` задается текст, который будет отображаться в заголовке окна. С помощью метода

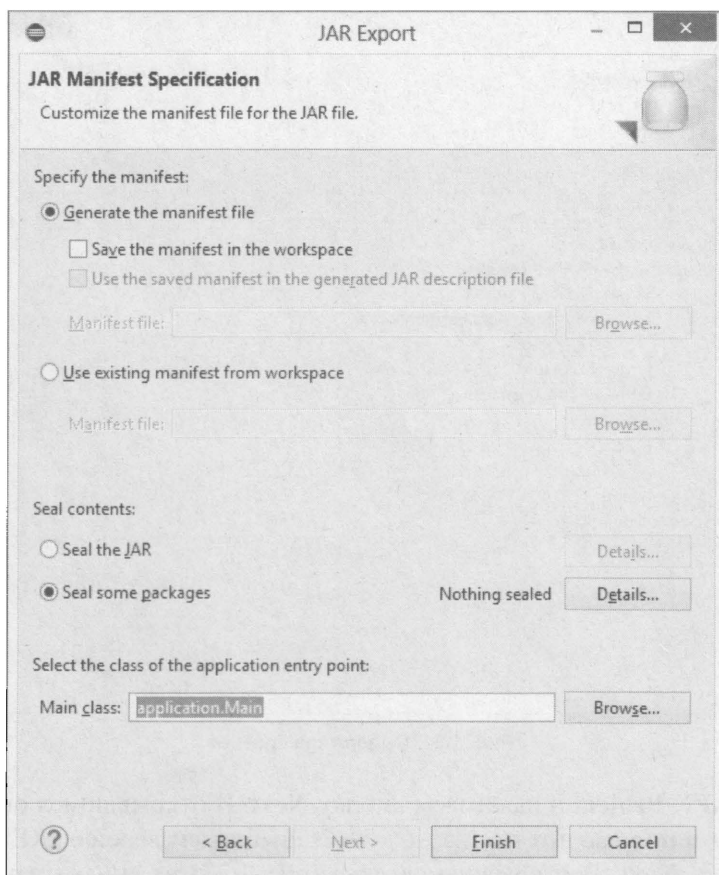


Рис. 1.30. Указание исполняемого класса внутри JAR-архива

`setScene()` объект класса `Scene` прикрепляется к главному окну, и с помощью метода `show()` главное окно отображается на экране.

JavaFX-приложение можно сравнить с театром. Класс `Stage` описывает сцену, а класс `Scene` — декорации, размещаемые на сцене. Сцена одна, а декораций может быть много, и они могут сменять друг друга.

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH01_01_Start`, приведенном в каталоге `CH01_first_steps` сопровождающего книгу электронного архива (см. приложение).

1.8. Создание JavaFX-приложения в Eclipse. Способ 2

После установки модуля `e(fx)clipse` в редакторе Eclipse становится доступным второй способ создания JavaFX-приложения. Для этого в меню **File** выбираем пункт **New | Other**. В открывшемся окне (рис. 1.31) в списке выбираем пункт

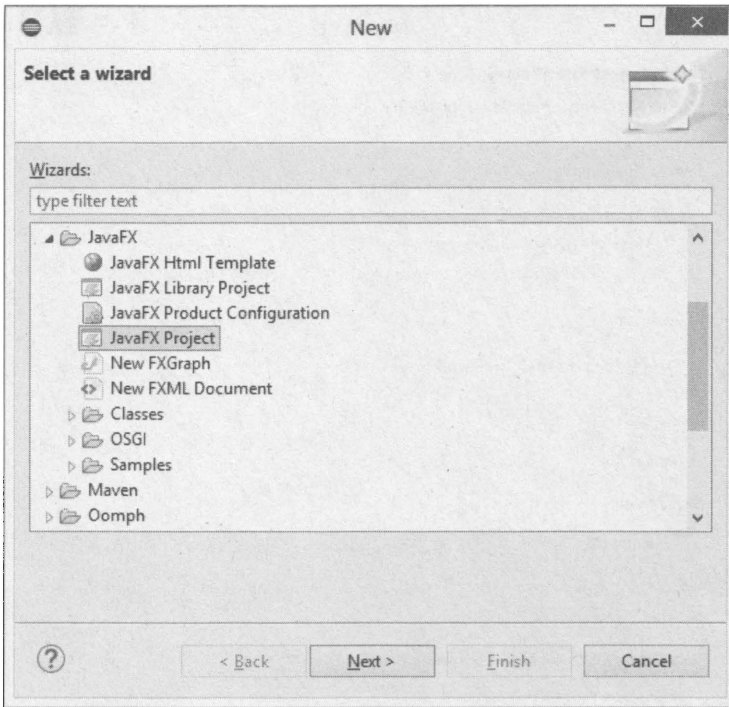


Рис. 1.31. Выбор типа проекта

JavaFX | JavaFX Project и нажимаем кнопку **Next**. В открывшемся окне (рис. 1.32) в поле **Project name** вводим `FX_CH01_02_Start`, выбираем версию JRE и дважды нажимаем кнопку **Next**. В открывшемся окне (рис. 1.33) из списка **Application type** выбираем пункт **Desktop**. Название пакета оставляем по умолчанию (`application`). Из списка **Language** выбираем пункт **FXML**, а из списка **Root-Type** — пункт `javafx.scene.layout.BorderPane`. В поле **File Name** оставляем название `Sample`, а в поле **Controller Name** — `SampleController`. Нажимаем кнопку **Finish**. Переходим в свойства проекта и добавляем все модули из библиотеки JavaFX в **Modulepath**.

В результате редактор Eclipse создаст несколько файлов в каталоге `C:\JavaSE\JavaFX\projects\FX_CH01_02_Start\src\application` (структура проекта показана на рис. 1.34):

- ☐ **Main.java** — главный файл с одноименным классом, наследующим класс `Application`;
- ☐ **Sample.fxml** — содержит описание контейнера и компонентов в формате FXML, который является разновидностью разметки XML;
- ☐ **SampleController.java** — класс, внутри которого мы будем обрабатывать различные события и изменять свойства компонентов. Название класса задается внутри файла `Sample.fxml` с помощью атрибута `fx:controller` тега `<BorderPane>`;
- ☐ **application.css** — файл, внутри которого можно задавать оформление различных компонентов с помощью каскадных таблиц стилей, как в Web-приложениях. Пока этот файл пустой.

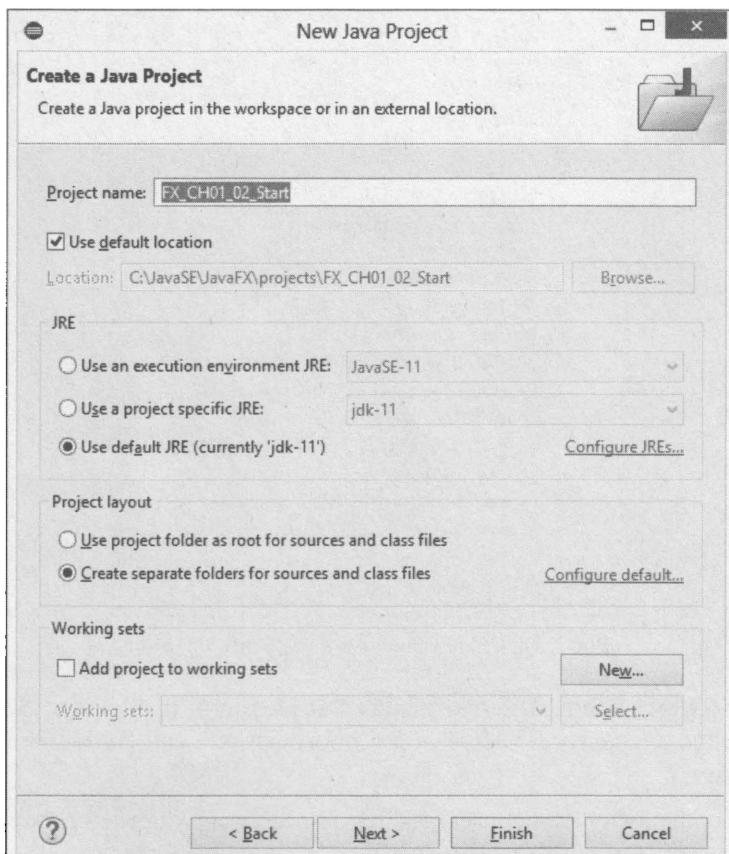


Рис. 1.32. Создание проекта

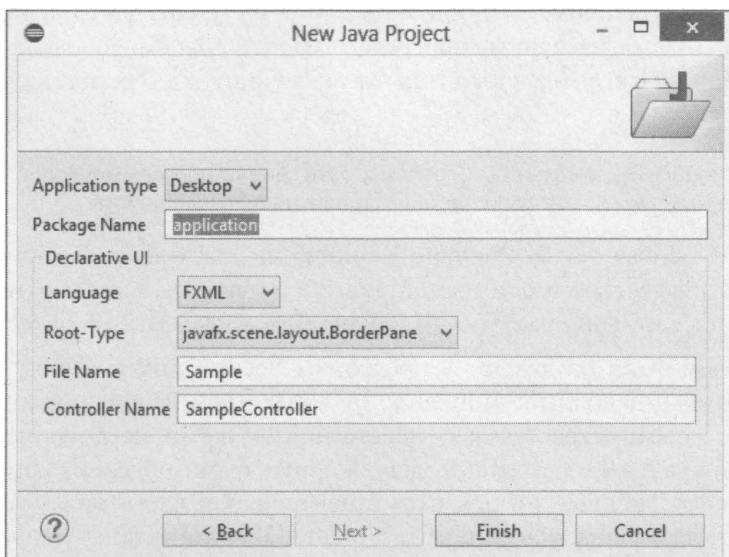


Рис. 1.33. Настройка параметров проекта

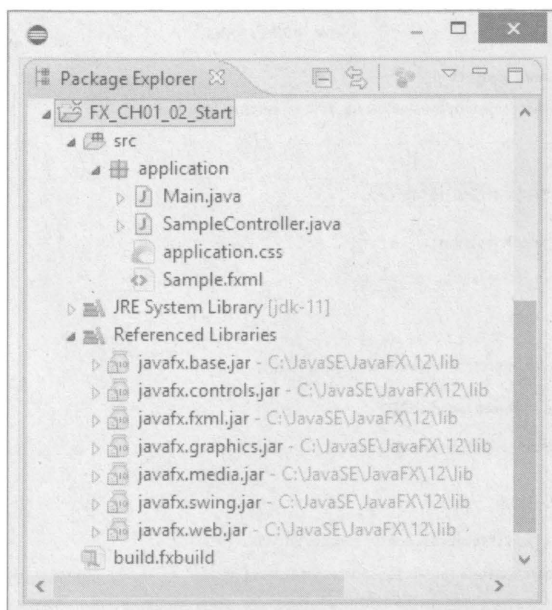


Рис. 1.34. Структура проекта FX_CH01_02_Start

Давайте попробуем отредактировать файл Sample.fxml в программе Scene Builder и добавим надпись. Открыть FXML-файл в программе Scene Builder можно несколькими способами:

- ☐ запускаем программу SceneBuilder.exe (C:\JavaSE\JavaFX\SceneBuilder) и в меню **File** выбираем пункт **Open**. В открывшемся окне находим файл Sample.fxml и нажимаем кнопку **Открыть**;
- ☐ делаем двойной щелчок левой кнопкой мыши на значке файла Sample.fxml (предварительно нужно назначить программу SceneBuilder.exe для открытия файлов с расширением fxml, выбрав в контекстном меню пункт **Открыть с помощью**);

ПРИМЕЧАНИЕ

В пути к файлу не должно быть русских букв, иначе программа Scene Builder не сможет открыть файл с помощью двойного щелчка на значке файла.

- ☐ в редакторе Eclipse в окне **Package Explorer** щелкаем правой кнопкой мыши на названии файла Sample.fxml и из контекстного меню выбираем пункт **Open with SceneBuilder**. Это наиболее удобный способ, которым мы и будем пользоваться.

Итак, открываем файл Sample.fxml в программе Scene Builder. Слева переходим на вкладку **Controls** и находим компонент Label. Щелкаем на компоненте левой кнопкой мыши и, не отпуская кнопку, перетаскиваем компонент на надпись insert Center контейнера BorderPane. Как только надпись будет обведена рамкой, отпускаем кнопку мыши. Надпись отобразится в центральной части окна. Теперь необходимо изменить текст надписи и размер шрифта. Для этого при выделенном компоненте в правой части окна открываем вкладку **Properties** и в поле **Text** вводим Привет, мир!. Чтобы изменить размер шрифта, открываем список **Font** и в поле **Size**

вводим значение 32. Все изменения отобразятся в центральной части окна. После окончания редактирования в меню **File** выбираем пункт **Save**. Результат редактирования приведен в листинге 1.2.

Листинг 1.2. Содержимое файла `Sample.fxml`

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.Label?>
<?import javafx.scene.layout.BorderPane?>
<?import javafx.scene.text.Font?>

<BorderPane xmlns:fx="http://javafx.com/fxml/1"
            xmlns="http://javafx.com/javafx/10.0.1"
            fx:controller="application.SampleController">
    <center>
        <Label text="Привет, мир!" BorderPane.alignment="CENTER">
            <font>
                <Font size="32.0" />
            </font>
        </Label>
    </center>
</BorderPane>
```

Для загрузки FXML-файла в программе используется статический метод `load()` из класса `FXMLLoader`. В качестве параметра метод принимает путь в виде объекта класса `URL`. Получить путь относительно пути поиска классов позволяет метод `getResource()` из класса `Class`. После загрузки FXML-файла необходимо выполнить приведение к типу класса контейнера:

```
BorderPane root = (BorderPane) FXMLLoader.load(
    getClass().getResource("Sample.fxml"));
```

Управлять оформлением различных компонентов можно с помощью каскадных таблиц стилей, как и в Web-приложениях. Для этого достаточно прописать стили в файле `application.css`. Давайте в качестве примера зададим цвет текста всех надписей (листинг 1.3).

Листинг 1.3. Содержимое файла `application.css`

```
Label { -fx-text-fill: #ff0000; }
```

Чтобы подключить этот файл к приложению, необходимо сначала получить ссылку на существующий список файлов (объект класса `ObservableList<E>`) с помощью метода `getStylesheets()` из класса `Scene`, а затем вызвать метод `add()` и передать ему путь к файлу в виде строки:

```
scene.getStylesheets().add(
    getClass().getResource("application.css").toExternalForm());
```


Полный текст программы для файла `Main.java` приведен в листинге 1.4.

Листинг 1.4. Содержимое файла `Main.java`

```
package application;

import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.BorderPane;
import javafx.fxml.FXMLLoader;

public class Main extends Application {
    @Override
    public void start(Stage primaryStage) throws Exception {
        BorderPane root = (BorderPane)FXMLLoader.load(
            getClass().getResource("Sample.fxml"));
        Scene scene = new Scene(root, 400, 150);
        scene.getStylesheets().add(
            getClass().getResource("application.css").toExternalForm());
        primaryStage.setTitle("Заголовок окна");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    public static void main(String[] args) {
        Application.launch(args);
    }
}
```

Содержимое файла `SampleController.java` приведено в листинге 1.5.

Листинг 1.5. Содержимое файла `SampleController.java`

```
package application;

public class SampleController {
}
```

Запуск JavaFX-приложения осуществляется точно так же, как и запуск обычного Java-приложения: в меню **Run** выбираем пункт **Run**, или нажимаем комбинацию клавиш `<Ctrl>+<F11>`, или нажимаем кнопку на панели инструментов. Однако сейчас после запуска мы получим следующее сообщение об ошибке:

```
Error: Could not find or load main class application.Main
Caused by: java.lang.NoClassDefFoundError: javafx/application/Application
```

Чтобы программа запускалась из редактора, нужно либо добавить в проект файл `module-info` и прописать в нем используемые модули, либо в настройках запуска добавить опцию `--add-modules`. Давайте попробуем оба эти способа на этом проекте, т. к. этом случае есть нюансы при создании модуля.

Начнем с создания модульного приложения. Чтобы добавить файл `module-info` в проект, в окне **Package Explorer** щелкаем правой кнопкой мыши на названии проекта и из контекстного меню выбираем пункт **Configure | Create module-info.java**. В открывшемся окне в поле **Module name** вводим название модуля, например `mymodule`, и нажимаем кнопку **Create**. Файл `module-info.java` будет добавлен в пакет по умолчанию и станет доступен для редактирования на отдельной вкладке. В файл `module-info.java` редактор вставит следующий код:

```
module mymodule {
    exports application;

    requires javafx.base;
    requires javafx.fxml;
    requires javafx.graphics;
}
```

Редактор опять добавил зависимости от модулей, вроде все хорошо, но это не так. Попытка запустить приложение приведет к ошибке (исключение `InvocationTargetException`). Причину такой ошибки я уже дважды описывал, но повторю и еще раз, чтобы обратить на это ваше внимание. При использовании FXML задействуется рефлексия, поэтому пакет нужно объявлять открытым с помощью инструкции `opens`, а не `exports`. Кроме того, внутри FXML мы используем компоненты, которые находятся в модуле `javafx.controls`, зависимость от которого также должна быть указана. Изменим содержимое файла `module-info.java`:

```
module mymodule {
    opens application to javafx.fxml, javafx.graphics, javafx.controls;
    requires transitive javafx.fxml;
    requires transitive javafx.controls;
}
```

Если мы сейчас запустим приложение, то получим окно почти как на рис. 1.1, но текст внутри окна будет красного цвета, а не черного, т. к. мы сами изменили цвет с помощью CSS в файле `application.css` (см. листинг 1.3).

Теперь рассмотрим второй способ, которым будем пользоваться в дальнейших примерах. Сначала удаляем файл `module-info.java` из проекта. Для этого в окне **Package Explorer** щелкаем правой кнопкой мыши на названии файла и из контекстного меню выбираем пункт **Delete**. В открывшемся окне подтверждаем удаление файла. Далее нам нужно добавить опцию `--add-modules` в командную строку для запуска приложения. Для этого открываем свойства проекта и в списке слева выбираем пункт **Run/Debug Settings**. Выделяем пункт с названием конфигурации запуска и нажимаем кнопку **Edit**. В открывшемся окне (см. рис. 1.27) переходим на вкладку **Arguments** и в поле **VM arguments** вводим следующую команду:

```
--add-modules ALL-MODULE-PATH
```

Сохраняем все изменения и запускаем программу.

ПРИМЕЧАНИЕ

Модуль `e(fx)clipse` версии 3.4 плохо работает с Java 11. Например, не работает дополнение в CSS. Возможно, в будущем это изменится, а пока, чтобы иметь возможность редактировать CSS-файлы, создайте новый проект и укажите в качестве компилятора и JRE версию 10, 9 или 8. После редактирования просто перенесите код в другой проект, работающий с Java 11. Использовать подсказки редактора гораздо удобнее, чем набирать код вручную.

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH01_02_Start`, приведенном в каталоге `CH01_first_steps` сопровождающего книгу электронного архива (см. *приложение*).

1.9. Указание идентификаторов компонентов

Чтобы при использовании FXML-файла иметь возможность управлять компонентом, необходимо указать *идентификатор* — уникальное имя компонента. Имя должно соответствовать требованиям по именованию переменных в языке Java. Указать идентификатор можно двумя способами:

- ❑ в программе Scene Builder выделяем компонент и справа на вкладке **Code** вводим имя компонента в поле **fx:id**;
- ❑ в FXML-файле для тега компонента вручную добавляем атрибут `fx:id`. Пример:

```
<Button fx:id="button1" text="Нажми меня"
        BorderPane.alignment="CENTER" />
```

Помимо атрибута `fx:id` существует также атрибут `id`, который задает идентификатор компонента для обращения к нему из CSS-файла. Не путайте эти атрибуты. Вот пример дополнительного указания атрибута `id`:

```
<Button fx:id="button1" id="my_button" text="Нажми меня"
        BorderPane.alignment="CENTER" />
```

Чтобы обратиться к компоненту из CSS-файла, необходимо вначале указать символ `#`, а затем имя идентификатора. Например, изменим цвет текста на кнопке:

```
#my_button { -fx-text-fill: #ff0000; }
```

Давайте создадим новый проект с названием `FX_CH01_09_id` по аналогии с предыдущим проектом, но вместо надписи добавим кнопку в центр окна. Для этого в программе Scene Builder в списке слева переходим на вкладку **Controls**, щелкаем левой кнопкой мыши на компоненте `Button` и, не отпуская кнопку мыши, перетаскиваем компонент на надпись **insert Center** контейнера `BorderPane`. Как только надпись будет обведена рамкой, отпускаем кнопку мыши. Кнопка отобразится в центральной части окна. На вкладке **Properties** в поле **Text** вводим `Нажми меня`, а в поле **id** — значение `my_button`. Затем на вкладке **Code** в поле **fx:id** вводим значение `button1`. Чтобы иметь возможность обработать нажатие кнопки, на вкладке **Code** в поле **On Action** вводим значение `onClick`. Сохраняем изменения.

В итоге описание кнопки в FXML-файле будет выглядеть следующим образом:

```
<Button id="my_button" fx:id="button1" mnemonicParsing="false"
        onAction="#onClick" text="Нажми меня"
        BorderPane.alignment="CENTER" />
```

Вся обработка событий осуществляется в классе контроллера `SampleController.java`. Во-первых, мы должны создать поле с именем, совпадающим с идентификатором, указанным в FXML-файле. Перед объявлением поля добавляется аннотация `@FXML`. Прежде чем использовать аннотацию, необходимо выполнить импорт с помощью инструкции:

```
import javafx.fxml.FXML;
```

Аннотация `@FXML` служит для обозначения защищенных и закрытых членов класса, связанных с FXML-файлом. Пример объявления поля:

```
@FXML
private Button button1;
```

Во-вторых, мы должны создать метод, который будет вызываться при наступлении события. В нашем случае при нажатии кнопки должен быть вызван метод `onClick()`. Формат метода:

```
@FXML
protected void onClick(ActionEvent event)
```

Метод принимает объект класса `ActionEvent`, через который доступна дополнительная информация о событии. Прежде чем использовать класс, необходимо выполнить импорт с помощью инструкции:

```
import javafx.event.ActionEvent;
```

Еще раз обратите внимание на то, что все названия пакетов начинаются с префикса `javafx`. Если удалить инструкцию импорта и навести указатель мыши на название класса, то редактор `Eclipse` предложит два варианта. Попытка выполнить импорт с помощью инструкции:

```
import java.awt.event.ActionEvent;
```

приведет к ошибке. Это очень частая ошибка!

Внутри метода `onClick()` выведем уведомление о нажатии кнопки в стандартный поток вывода. Этим способом удобно пользоваться на этапе отладки программы. Текст сообщения отобразится в окне консоли. Пример:

```
System.out.println("Нажата кнопка");
```

Чтобы продемонстрировать способ доступа к компоненту по идентификатору, изменим текст надписи при нажатии кнопки. Например, выведем количество нажатий с помощью метода `setText()`:

```
n++;
button1.setText("Вы нажали " + n + " раз(a)");
```

Редактор Eclipse позволяет автоматически сгенерировать класс контроллера, но при этом файл класса не должен существовать. Поэтому придется вначале удалить существующий файл `SampleController.java`. После этого переходим в FXML-файл, щелкаем правой кнопкой мыши в любом месте файла и из контекстного меню выбираем пункт **Source | Generate Controller**. В открывшемся окне проверяем настройки и нажимаем кнопку **ОК**. Файл `SampleController.java` будет добавлен в проект и внутри него будет создан класс и все необходимые объявления полей и методов. Содержимое файла `SampleController.java` приведено в листинге 1.6.

Листинг 1.6. Содержимое файла `SampleController.java`

```
package application;

import javafx.fxml.FXML;
import javafx.scene.control.Button;
import javafx.event.ActionEvent;

public class SampleController {
    @FXML
    private Button button1;

    private int n = 0;

    @FXML
    protected void onClick(ActionEvent event) {
        System.out.println("Нажата кнопка");
        n++;
        button1.setText("Вы нажали " + n + " раз(a)");
    }
}
```

Создать шаблон контроллера можно также в программе Scene Builder. Для этого в меню **View** выбираем пункт **Show Sample Controller Skeleton**. В результате текст шаблона контроллера будет доступен для копирования в открывшемся окне.

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH01_09_id`, приведенном в каталоге `CH01_first_steps` сопровождающего книгу электронного архива (см. *приложение*).

1.10. Доступ к главному окну приложения внутри класса контроллера

Очень часто внутри класса контроллера необходимо получить доступ к главному окну приложения. Например, чтобы узнать размеры окна или изменить текст в заголовке окна. Для этого нужно передать в класс контроллера ссылку на объект класса `Stage`. Давайте попробуем это сделать.

В редакторе Eclipse создаем копию проекта `FX_CH01_09_id` и называем новый проект `FX_CH01_10_Main`. Для этого в окне **Package Explorer** щелкаем правой кнопкой мыши на названии проекта `FX_CH01_09_id` и из контекстного меню выбираем пункт **Сору**. Далее щелкаем правой кнопкой мыши на свободном месте окна и из контекстного меню выбираем пункт **Paste**. В открывшемся окне вводим новое название в поле **Project name** и нажимаем кнопку **Сору**.

В предыдущем проекте мы загружали FXML-файл с помощью следующего кода:

```
BorderPane root = (BorderPane)FXMLLoader.load(  
    getClass().getResource("Sample.fxml"));
```

Чтобы передать ссылку на объект `Stage` в контроллер, необходимо вначале получить ссылку на класс контроллера. Для этого предназначен метод `getController()` из класса `FXMLLoader`. Давайте переделаем предыдущий код и сохраним ссылку на объект класса `FXMLLoader`, чтобы можно было вызвать метод `getController()` после загрузки FXML-файла:

```
FXMLLoader loader = new FXMLLoader();  
loader.setLocation(getClass().getResource("Sample.fxml"));  
BorderPane root = (BorderPane)loader.load();
```

Теперь можно передать ссылку на объект `Stage` в класс контроллера:

```
SampleController controller = loader.getController();  
controller.setPrimaryStage(primaryStage);
```

В этом примере мы воспользовались вспомогательным методом `setPrimaryStage()`, который необходимо реализовать внутри класса контроллера. Для этого в классе `SampleController` вначале объявляем поле `primaryStage`, в котором будем хранить ссылку:

```
private Stage primaryStage;
```

Далее внутри метода `setPrimaryStage()` присваиваем ссылку этому полю:

```
public void setPrimaryStage(Stage primaryStage) {  
    this.primaryStage = primaryStage;  
}
```

Чтобы через это поле получить доступ к объекту класса `Scene`, достаточно вызвать метод `getScene()`:

```
Scene scene = primaryStage.getScene();
```

В качестве примера внутри обработчика нажатия кнопки получим заголовок окна, его размеры, а также координаты левого верхнего угла сцены (листинг 1.7).

Листинг 1.7. Метод `onClick()` из класса `SampleController`

```
@FXML  
protected void onClick(ActionEvent event) {  
    // Получаем заголовок окна  
    System.out.println(primaryStage.getTitle());
```

```
// Получаем ширину и высоту окна
System.out.println(primaryStage.getWidth() + " : " +
    primaryStage.getHeight());
Scene scene = primaryStage.getScene();
// Получаем координаты сцены внутри окна
System.out.println(scene.getX() + " : " + scene.getY());
}
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH01_10_Main`, приведенном в каталоге `CH01_first_steps` сопровождающего книгу электронного архива (см. приложение).

Получить ссылку на объект окна можно также внутри обработчика события через параметр `event`. Для этого вначале нужно получить ссылку на источник события с помощью метода `getSource()` и выполнить приведение к классу `Node`, который наследуют все классы компонентов. После этого можно получить ссылку на объект класса `Scene` с помощью метода `getScene()`. Через объект класса `Scene` можно получить ссылку на объект класса `Window` с помощью метода `getWindow()`. Класс `Window` является базовым для класса `Stage`. В качестве примера переделаем код из листинга 1.7 и получим ссылку на окно через объект события (листинг 1.8).

Листинг 1.8. Метод `onClick()` из класса `SampleController`

```
@FXML
protected void onClick(ActionEvent event) {
    Scene scene = (Node) event.getSource().getScene();
    Window window = scene.getWindow();
    // Получаем заголовок окна
    System.out.println(((Stage) window).getTitle());
    // Получаем ширину и высоту окна
    System.out.println(window.getWidth() + " : " +
        window.getHeight());
    // Получаем координаты сцены внутри окна
    System.out.println(scene.getX() + " : " + scene.getY());
}
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH01_10_event`, приведенном в каталоге `CH01_first_steps` сопровождающего книгу электронного архива (см. приложение).

В некоторых случаях необходимо иметь ссылку на главное окно внутри метода `initialize()` (см. *разд. 1.11*). Чтобы передать ссылку, в этом случае можно воспользоваться следующим кодом:

```
FXMLLoader loader = new FXMLLoader();
loader.setLocation(getClass().getResource("DialogOK.fxml"));
```

```

loader.setControllerFactory(new Callback<Class<?>, Object>() {
    @Override
    public Object call(Class<?> param) {
        return new DialogOKController(DialogOK.this);
    }
});
GridPane root = (GridPane)loader.load();

```

В этом примере файл класса и FXML-файл имеют название `DialogOK`, а класс контроллера имеет название `DialogOKController`. Для передачи ссылки в этом случае используется конструктор класса `DialogOKController`:

```

public DialogOKController(DialogOK dialog) {
    this.dialog = dialog;
}

```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH01_10_DialogOK`, приведенном в каталоге `CH01_first_steps` сопровождающего книгу электронного архива (см. *приложение*).

Существует еще один очень простой способ доступа к главному окну из класса контроллера — достаточно сохранить ссылку на главное окно в статическом поле в классе `Main`:

```

public class Main extends Application {
    private static Stage pStage; // Сохранение ссылки в статическом поле

    @Override
    public void start(Stage primaryStage) throws Exception {
        Main.pStage = primaryStage; // Сохраняем ссылку на окно
        // ...
    }

    public static Stage getPrimaryStage() {
        return Main.pStage;
    }
    // ...
}

```

Обращение к главному окну внутри класса контроллера в этом случае будет выглядеть следующим образом:

```

Stage primaryStage = Main.getPrimaryStage();

```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH01_10_static`, приведенном в каталоге `CH01_first_steps` сопровождающего книгу электронного архива (см. *приложение*).

1.11. Метод *initialize()* и интерфейс *Initializable*

При использовании контроллера важно понимать, что внутри конструктора нельзя обращаться к компонентам, т. к. на этот момент они еще не существуют, и поля будут иметь значение `null`. Чтобы иметь возможность настраивать компоненты, следует внутри блока класса контроллера создать метод `initialize()`. Этот метод будет автоматически вызван сразу после создания компонентов. Пример использования метода `initialize()` приведен в листинге 1.9.

Листинг 1.9. Метод `initialize()`

```
package application;

import javafx.fxml.FXML;
import javafx.scene.control.Button;
import javafx.event.ActionEvent;

public class SampleController {
    @FXML
    private Button button1;

    public SampleController() {
        // Здесь компоненты еще не существуют!
        System.out.println(button1); // null
    }

    @FXML
    protected void initialize() {
        button1.setText("Текст на кнопке");
        System.out.println("Метод initialize()");
    }

    @FXML
    protected void onClick(ActionEvent event) {
        System.out.println("Нажата кнопка");
    }
}
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH01_11_initialize`, приведенном в каталоге `CH01_first_steps` сопровождающего книгу электронного архива (см. приложение).

Можно также реализовать интерфейс `Initializable`, который содержит объявление метода `initialize()`:

```
public void initialize(URL location, ResourceBundle resources)
```

Если класс реализует интерфейс `Initializable`, то метод `initialize()` без параметров вызван не будет. Давайте реализуем интерфейс, а заодно рассмотрим способы назначения обработчиков событий из программы (листинг 1.10).

Листинг 1.10. Интерфейс `Initializable`

```
package application;

import javafx.fxml.FXML;
import javafx.fxml.Initializable;
import javafx.scene.control.Button;
import java.net.URL;
import java.util.ResourceBundle;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;

public class SampleController implements Initializable {

    @FXML private Button button1;
    @FXML private Button button2;
    @FXML private Button button3;

    @Override
    public void initialize(URL location, ResourceBundle resources) {
        button1.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent event) {
                System.out.println("Нажата кнопка 1");
            }
        });
        button2.setOnAction(this::onClick);
        button3.setOnAction(event -> {
            System.out.println("Нажата кнопка 3");
        });
    }

    @FXML
    protected void onClick(ActionEvent event) {
        System.out.println("Нажата кнопка 2");
    }
}
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH01_11_Initializable`, приведенном в каталоге `CH01_first_steps` сопровождающего книгу электронного архива (см. *приложение*).

Итак, для назначения обработчика события нажатия кнопки используется метод `setOnAction()`. В этот метод мы можем передать:

- ❑ объект класса, реализующего функциональный интерфейс `EventHandler<T extends Event>`. Интерфейс содержит объявление метода `handle()`:

```
public void handle(T event)
```

- ❑ анонимный вложенный класс:

```
button1.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent event) {
        System.out.println("Нажата кнопка 1");
    }
});
```

- ❑ ссылку на метод:

```
button2.setOnAction(this::onClick);
```

- ❑ лямбда-выражение:

```
button3.setOnAction(event -> {
    System.out.println("Нажата кнопка 3");
});
```

1.12. Методы *init()* и *stop()*

Как вы уже знаете, основной класс JavaFX-приложения должен наследовать класс `Application` и реализовать метод `start()`, который будет вызван после создания объекта в потоке диспетчера обработки событий. Если необходимо выполнить какие-либо предварительные действия до вызова метода `start()`, то внутри класса следует реализовать метод `init()`. Обратите внимание: метод будет вызван в потоке `JavaFX-Launcher`, а не в потоке диспетчера обработки событий. Формат метода:

```
public void init() throws Exception
```

Чтобы выполнить какие-либо завершающие действия, можно реализовать метод `stop()`. Формат метода:

```
public void stop() throws Exception
```

Давайте реализуем эти методы, выведем последовательность их вызова и названия потоков (листинг 1.11).

Листинг 1.11. Методы `init()` и `stop()`

```
package application;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
```

```

import javafx.scene.layout.BorderPane;
import javafx.scene.text.Font;
import javafx.stage.Stage;

public class Main extends Application {

    public static void main(String[] args) {
        System.out.println("Метод main()" + " " +
            Thread.currentThread().getName());
        Application.launch(args);
    }
    @Override
    public void init() throws Exception {
        System.out.println("Метод init()" + " " +
            Thread.currentThread().getName());
    }
    @Override
    public void start(Stage stage) throws Exception {
        System.out.println("Метод start()" + " " +
            Thread.currentThread().getName());
        BorderPane root = new BorderPane();
        Label text = new Label("Привет, мир!");
        text.setFont(new Font(32));
        root.setCenter(text);
        Scene scene = new Scene(root, 400, 150);

        stage.setTitle("Заголовок окна");
        stage.setScene(scene);
        stage.show();
    }
    @Override
    public void stop() throws Exception {
        System.out.println("Метод stop()" + " " +
            Thread.currentThread().getName());
    }
}

```

Результат выполнения:

```

Метод main() main
Метод init() JavaFX-Launcher
Метод start() JavaFX Application Thread
Метод stop() JavaFX Application Thread

```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH01_12_init`, приведенном в каталоге `CH01_first_steps` сопровождающего книгу электронного архива (см. *приложение*).

1.13. Получение параметров из командной строки

При наличии метода `main()` необходимо вызвать статический метод `launch()` из класса `Application` и передать ему список значений, указанных в командной строке. Форматы метода:

```
public static void launch(String... args)
public static void launch(Class<? extends Application> appClass,
                        String... args)
```

Первый формат мы уже не раз использовали в предыдущих примерах. Во втором формате в первом параметре можно дополнительно указать класс:

```
public static void main(String[] args) {
    Application.launch(Main.class, args);
}
```

После вызова метода `launch()` управление вернется методу `main()` только после завершения работы приложения.

Чтобы получить список значений, переданных в командной строке, внутри JavaFX-приложения следует воспользоваться методом `getParameters()` из класса `Application`. Метод можно вызвать внутри метода `init()` или позже, но не раньше. Формат метода:

```
public final Application.Parameters getParameters()
```

Класс `Application.Parameters` содержит три основных метода:

❑ `getRaw()` — возвращает список всех параметров. Формат метода:

```
public List<String> getRaw()
```

❑ `getNamed()` — возвращает словарь с именованными параметрами (`--key=value`). Формат метода:

```
public Map<String, String> getNamed()
```

❑ `getUnnamed()` — возвращает список с неименованными параметрами. Формат метода:

```
public List<String> getUnnamed()
```

В качестве примера в каталоге `C:\book` создадим файл `Main.java` с кодом из листинга 1.12. Затем скомпилируем программу в командной строке и запустим на выполнение, передав различные параметры.

Листинг 1.12. Получение параметров из командной строки

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
```

```

import javafx.scene.layout.BorderPane;
import javafx.stage.Stage;

public class Main extends Application {

    public static void main(String[] args) {
        Application.launch(args);
    }
    @Override
    public void init() {
        System.out.println("getRaw(): " +
            this.getParameters().getRaw());
        System.out.println("getNamed(): " +
            this.getParameters().getNamed().keySet());
        System.out.println("getUnnamed(): " +
            this.getParameters().getUnnamed());
    }
    @Override
    public void start(Stage stage) throws Exception {
        BorderPane root = new BorderPane();
        root.setCenter(new Label("Привет, мир!"));
        Scene scene = new Scene(root, 400, 150);

        stage.setTitle("Заголовок окна");
        stage.setScene(scene);
        stage.show();
    }
}

```

Процесс компиляции и выполнения в командной строке:

```

C:\book>javac -encoding utf-8 --module-path C:\JavaSE\JavaFX\12\lib
--add-modules javafx.controls Main.java

C:\book>java --module-path C:\JavaSE\JavaFX\12\lib
--add-modules javafx.controls Main --param1=123 param2
getRaw(): [--param1=123, param2]
getNamed(): [param1]
getUnnamed(): [param2]

```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH01_13_getParameters`, приведенном в каталоге `CH01_first_steps` сопровождающего книгу электронного архива (см. *приложение*).

1.14. Завершение работы JavaFX-приложения

Работа JavaFX-приложения автоматически завершается после закрытия всех окон и завершения работы всех других пользовательских потоков. Если необходимо завершить работу JavaFX-приложения из программы, то следует воспользоваться статическим методом `exit()` из класса `Platform`. Формат метода:

```
public static void exit()
```

Можно также воспользоваться статическим методом `exit()` из класса `System`, но при этом метод `stop()` вызван не будет.

В качестве примера завершим работу приложения по нажатию кнопки (листинг 1.13).

Листинг 1.13. Завершение работы JavaFX-приложения

```
package application;

import javafx.application.Application;
import javafx.application.Platform;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.BorderPane;
import javafx.stage.Stage;

public class Main extends Application {

    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) throws Exception {
        BorderPane root = new BorderPane();
        Button button = new Button("Закрыть окно");
        root.setCenter(button);
        button.setOnAction(event -> Platform.exit());
        // button.setOnAction(event -> System.exit(0));
        Scene scene = new Scene(root, 400, 150);

        stage.setTitle("Заголовок окна");
        stage.setScene(scene);
        stage.show();
    }

    @Override
    public void stop() throws Exception {
        System.out.println("Метод stop()");
    }
}
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH01_14_exit`, приведенном в каталоге `CH01_first_steps` сопровождающего книгу электронного архива (см. приложение).

1.15. Выполнение длительных операций

Для взаимодействия с системой и запуска обработчиков предназначен основной цикл приложения. После вызова метода `start()` программа переходит в бесконечный цикл. Чтобы завершить работу приложения, необходимо вызвать метод `exit()` из класса `Platform`. Так как программа находится внутри цикла, то вызвать этот метод можно только при наступлении какого-либо события, — например, при нажатии пользователем кнопки. Цикл автоматически прерывается при нажатии пользователем кнопки **Заккрыть** в заголовке последнего окна.

После наступления события цикл прерывается и управление передается в обработчик. После завершения работы обработчика управление опять передается основному циклу приложения. Если внутри обработчика выполняется длительная операция, то программа перестает реагировать на события. В качестве примера изобразим длительный процесс с помощью метода `sleep()` из класса `Thread` (листинг 1.14).

Листинг 1.14. Выполнение длительных операций

```
package application;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.BorderPane;
import javafx.stage.Stage;

public class Main extends Application {

    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) throws Exception {
        BorderPane root = new BorderPane();
        Button button = new Button("Запустить процесс");
        root.setCenter(button);
        button.setOnAction(event -> {
            try {
                Thread.sleep(10000);
                System.out.println("stop");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });
        Scene scene = new Scene(root, 400, 150);
```



```

stage.setTitle("Заголовок окна");
stage.setScene(scene);
stage.show();
}
}

```

В этом примере при нажатии кнопки **Запустить процесс** внутри обработчика мы «засыпаем» на десять секунд и тем самым прерываем основной цикл приложения. Попробуйте нажать кнопку, и вы увидите, что окно перестает реагировать на любые события, и после перекрытия другим окном приложение не может выполнить перерисовку окна, пока не закончится выполнение процесса. Если в этот момент нажать кнопку **Заккрыть** в заголовке окна, то будет выведено окно, с помощью которого система сообщает, что программа не отвечает, и предлагает завершить ее.

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH01_15_NoThread`, приведенном в каталоге `CH01_first_steps` сопровождающего книгу электронного архива (см. *приложение*).

При выполнении длительной операции следует внутри обработчика создать отдельный поток и уже внутри него производить различные действия. В этом случае длительная операция станет выполняться параллельно с основным циклом приложения и не будет его блокировать. Переделаем код обработчика и создадим внутри него отдельный поток (листинг 1.15).

Листинг 1.15. Запуск длительной операции в отдельном потоке

```

button.setOnAction(event -> {
    (new Thread( () -> {
        try {
            System.out.println("start");
            Thread.sleep(10000);
            System.out.println("stop");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    })).start();
});

```

Обратите внимание: при завершении основного цикла JavaFX-приложения пользовательские потоки автоматически не завершаются. Кроме того, следует учитывать, что изменять свойства компонентов можно только из потока диспетчера задач, который называется `JavaFX Application Thread`. Чтобы изменить свойства из другого потока, нужно сформировать задачу (интерфейс `Runnable`) и передать ее статическому методу `runLater()` из класса `Platform` (см. *разд. 4.6.3*).

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH01_15_Thread`, приведенном в каталоге `CH01_first_steps` сопровождающего книгу электронного архива (см. *приложение*).

ГЛАВА 2



Управление окном приложения

Создать окно и управлять им позволяет класс `Stage`. Иерархия наследования:

```
Object — Window — Stage
```

Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.stage.Stage;
```

Класс `Stage` наследует класс `Window`, а значит, через объект класса `Stage` доступны все методы из класса `Window`. Класс `Window` является базовым также для класса `PopupWindow` (см. *разд. 15.6*), с помощью которого реализуются различные всплывающие окна, например подсказки, уведомления и контекстные меню.

Начиная с версии 8.40, в состав стандартной библиотеки входит класс `Dialog<R>`, с помощью которого реализуются различные диалоговые окна. Подробно этот класс и классы, наследующие класс `Dialog<R>`, мы рассмотрим в *главе 16*.

2.1. Создание и отображение окна

При создании приложения ссылка на окно верхнего уровня доступна через параметр метода `start()`. Если же необходимо создать дополнительное окно, то можно воспользоваться следующими конструкторами:

```
Stage()
Stage(StageStyle style)
```

Первый конструктор создает пустое окно с рамкой и заголовком, в котором расположены: значок, после щелчка на котором отображается меню окна (рис. 2.1), текст заголовка и кнопки **Свернуть**, **Развернуть** и **Заккрыть**. Второй конструктор позволяет указать стиль окна (см. *разд. 2.2*). В качестве примера по нажатию кнопки внутри главного окна создадим дополнительное окно и отобразим его (листинг 2.1).

Листинг 2.1. Создание и отображение окна

```
package application;

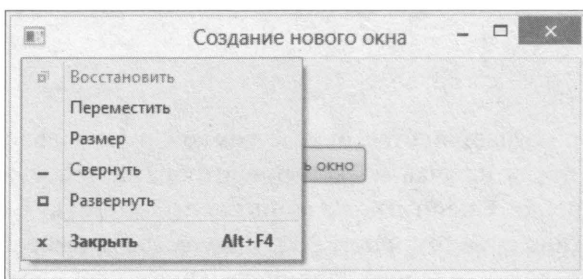
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.BorderPane;
import javafx.stage.Stage;

public class Main extends Application {

    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) throws Exception {
        BorderPane root = new BorderPane();
        Button button = new Button("Открыть окно");
        root.setCenter(button);
        button.setOnAction(event -> {
            newWindow(stage);
        });
        Scene scene = new Scene(root, 400, 150);
        stage.setTitle("Создание нового окна");
        stage.setScene(scene);
        stage.show();
    }

    public void newWindow(Stage parent) {
        Stage window = new Stage();    // Создаем окно
        window.setTitle("Новое окно");  // Указываем заголовок
        window.show();                 // Отображаем окно
    }
}
```

**Рис. 2.1. Меню окна**

Попробуйте нажать на кнопку несколько раз. В результате после каждого нажатия будет создано еще одно окно. Чтобы закрыть окно, необходимо нажать кнопку **Закрыть** в заголовке окна. Как только последнее окно будет закрыто, приложение прекратит работу.

Получить неизменяемый список всех окон приложения позволяет статический метод `getWindows()` из класса `Window`. Метод доступен, начиная с JavaFX 9. Формат метода:

```
public static ObservableList<Window> getWindows()
```

Пример вызова метода:

```
System.out.println(Window.getWindows());
```

Для изменения и получения текста в заголовке окна предназначены методы `setTitle()` и `getTitle()` из класса `Stage`. Форматы методов:

```
public final void setTitle(String value)
public final String getTitle()
```

Для управления заголовком окна можно также воспользоваться методом `titleProperty()` из класса `Stage`. Формат метода:

```
public final StringProperty titleProperty()
```

Метод возвращает свойство (объект класса `StringProperty`), связанное с заголовком окна. С помощью свойства мы можем задать текст заголовка (метод `setValue()`), получить текст (метод `getValue()`) или назначить обработчик изменения заголовка (метод `addListener()`). Изменение значения свойства сразу отразится на заголовке окна. Форматы методов:

```
public void setValue(String v)
public String getValue()
public void addListener(InvalidationListener listener)
public void addListener(ChangeListener<? super T> listener)
```

Пример:

```
window.titleProperty().addListener((obj, oldValue, newValue) -> {
    System.out.println(window.titleProperty().getValue());
    System.out.println(oldValue + " - " + newValue);
});
window.titleProperty().setValue("Новый заголовок");
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH02_01_titleProperty`, приведенном в каталоге `CH02_windows` сопровождающего книгу электронного архива (см. *приложение*).

После создания окна и настройки компонентов следует вызвать метод `show()`, который отобразит окно на экране. Формат метода:

```
public final void show()
```

Пример:

```
window.show();
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH02_01_show`, приведенном в каталоге `CH02_windows` сопровождающего книгу электронного архива (см. *приложение*).

Вместо метода `show()` можно воспользоваться методом `showAndWait()`. Формат метода:

```
public void showAndWait()
```

Метод `showAndWait()` отображает окно и ожидает его закрытия. При этом обработка остальных событий блокируется. Если внутри обработчика события нажатия кнопки после вызова метода `newWindow()` вставить инструкцию вывода сообщения, то инструкция будет выполнена только после закрытия или сокрытия окна. При использовании метода `show()` инструкция будет выполнена сразу после отображения окна, не дожидаясь его закрытия. Пример:

```
button.setOnAction(event -> {
    newWindow(stage);
    System.out.println("Окно закрыто");
});
// ... внутри метода newWindow()
window.showAndWait();           // Отображаем окно и ждем закрытия
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH02_01_showAndWait`, приведенном в каталоге `CH02_windows` сопровождающего книгу электронного архива (см. *приложение*).

Для сокрытия окна предназначен метод `hide()`. Обратите внимание: метод не закрывает окно, а всего лишь делает его невидимым. Формат метода:

```
public void hide()
```

Проверить, видимо окно в настоящее время или нет, позволяет метод `isShowing()`. Метод возвращает `true`, если окно видимо (даже если окно свернуто), и `false` — в противном случае. Формат метода:

```
public final boolean isShowing()
```

Получить состояние видимости окна, а также отслеживать изменение состояния, позволяет свойство `showing`. Получить ссылку на свойство можно с помощью метода `showingProperty()`. Формат метода:

```
public final ReadOnlyBooleanProperty showingProperty()
```

Пример:

```
stage.showingProperty().addListener(
    (obj, oldValue, newValue) -> {
```

```
System.out.println("Состояние окна изменилось: " +
    stage.showingProperty().getValue());
System.out.println(oldValue + " " + newValue);
});
```

Скроем или отобразим главное окно по нажатию кнопки в дополнительном окне (листинг 2.2).

Листинг 2.2. Соккрытие и отображение окна

```
public void newWindow(Stage parent) {
    Stage window = new Stage();
    window.setTitle("Новое окно");
    BorderPane pane = new BorderPane();
    Button btn = new Button("Скрыть или отобразить главное окно");
    pane.setCenter(btn);
    btn.setOnAction(event -> {
        if (parent.isShowing()) parent.hide();
        else parent.show();
    });
    window.setScene(new Scene(pane, 400, 150));
    window.show();
}
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH02_01_hide`, приведенном в каталоге `CH02_windows` сопровождающего книгу электронного архива (см. приложение).

2.2. Указание стиля окна

По умолчанию окно создается с заголовком, в котором расположены: значок, текст заголовка и кнопки **Свернуть**, **Развернуть** и **Заккрыть**. Указать другой тип создаваемого окна позволяет конструктор класса `Stage`, а также метод `initStyle()`. Обратите внимание на то, что метод `initStyle()` должен вызываться перед отображением окна. Форматы конструктора и метода:

```
import javafx.stage.StageStyle;
Stage(StageStyle style)
public final void initStyle(StageStyle style)
```

В качестве параметра `style` указываются следующие константы из перечисления `StageStyle`:

- `DECORATED` — значение по умолчанию. Окно белого цвета с рамкой и заголовком, в котором расположены: значок, текст заголовка и кнопки **Свернуть**, **Развернуть** и **Заккрыть**. При щелчке левой кнопкой мыши на значке отображается меню окна (см. рис. 2.1). Размеры окна можно изменять с помощью мыши, взявшись за границы окна;

- ❑ UNDECORATED — окно белого цвета без рамки и заголовка. Изменить размеры окна с помощью мыши нельзя;
- ❑ TRANSPARENT — прозрачное окно без рамки и заголовка. Изменить размеры окна с помощью мыши нельзя;
- ❑ UTILITY — окно белого цвета с рамкой и заголовком, в котором расположен только текст и кнопка **Заккрыть**. Размеры окна можно изменять с помощью мыши. Значок окна на панели задач не отображается;
- ❑ UNIFIED — окно белого цвета с рамкой и заголовком, в котором расположены: значок, текст заголовка и кнопки **Свернуть**, **Развернуть** и **Заккрыть**. При щелчке левой кнопкой мыши на значке отображается меню окна. Размеры окна можно изменять с помощью мыши, взявшись за границы окна. В отличие от стиля DECORATED, внутренняя граница рамки не выделяется. Если стиль UNIFIED не поддерживается платформой, то используется стиль DECORATED. Проверить возможность использования стиля UNIFIED позволяет следующий код:

```
// import javafx.application.ConditionalFeature;
// import javafx.application.Platform;
if (Platform.isSupported(
    ConditionalFeature.UNIFIED_WINDOW)) {
    System.out.println("Стиль UNIFIED поддерживается");
}
else {
    System.out.println("Нет поддержки");
}
```

Окна различных стилей показаны на рис. 2.2.

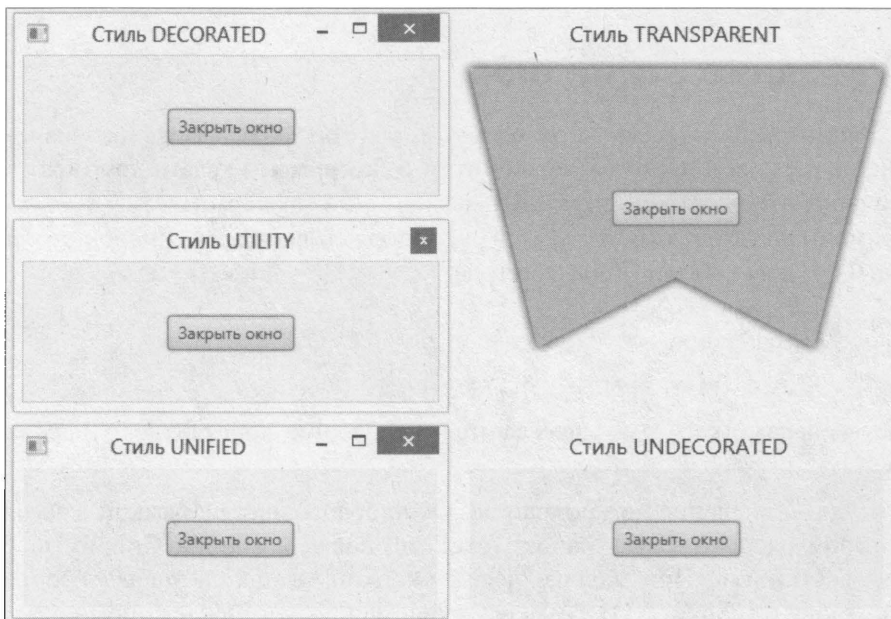


Рис. 2.2. Стили окна

Пример указания стиля в конструкторе класса:

```
Stage window = new Stage(StageStyle.DECORATED);
```

Пример указания стиля в методе `initStyle()`:

```
Stage window = new Stage();
window.initStyle(StageStyle.UTILITY);
// ...
window.show();
```

Получить текущий стиль окна позволяет метод `getStyle()`. Формат метода:

```
public final StageStyle getStyle()
```

ЭЛЕКТРОННЫЙ АРХИВ

Пример использования различных стилей содержится в проекте `FX_CH02_02_StageStyle`, приведенном в каталоге `CH02_windows` сопровождающего книгу электронного архива (см. приложение).

Часто у программистов возникает вопрос: можно ли убрать или сделать неактивными кнопки в заголовке окна? Если окно имеет фиксированные размеры, то кнопка **Развернуть** будет неактивной, если окно является модальным, то неактивной станет кнопка **Свернуть**, а если окно является и модальным, и с фиксированными размерами, то кнопок **Свернуть** и **Развернуть** вообще не будет. Если использовать стиль `UTILITY`, то кнопок **Свернуть** и **Развернуть** также не будет. При использовании класса `Dialog<R>` (см. *разд. 16.1*) кнопки **Свернуть** и **Развернуть** по умолчанию не отображаются. Что же касается кнопки **Заккрыть**, то с помощью метода `setOnCloseRequest()` можно назначить обработчик события и внутри этого обработчика вызвать метод `consume()` через объект события, тем самым запрещая закрытие окна с помощью кнопки **Заккрыть**. Пример:

```
primaryStage.setOnCloseRequest(new EventHandler<WindowEvent>() {
    @Override
    public void handle(WindowEvent event) {
        event.consume();
    }
});
```

2.3. Изменение и получение размеров окна

Для изменения размеров окна предназначены следующие методы:

❑ `setWidth()` — задает ширину окна. Формат метода:

```
public final void setWidth(double value)
```

❑ `setHeight()` — задает высоту окна. Формат метода:

```
public final void setHeight(double value)
```

❑ `setMinWidth()` — задает минимальную ширину окна. Значение по умолчанию `0.0`. Формат метода:

```
public final void setMinWidth(double value)
```


- ❑ `setMinHeight()` — задает минимальную высоту окна. Значение по умолчанию 0.0. **Формат метода:**

```
public final void setMinHeight(double value)
```

- ❑ `setMaxWidth()` — задает максимальную ширину окна. Значение по умолчанию `Double.MAX_VALUE`. **Формат метода:**

```
public final void setMaxWidth(double value)
```

- ❑ `setMaxHeight()` — задает максимальную высоту окна. Значение по умолчанию `Double.MAX_VALUE`. **Формат метода:**

```
public final void setMaxHeight(double value)
```

- ❑ `sizeToScene()` — подгоняет размеры окна под содержимое. **Формат метода:**

```
public void sizeToScene()
```

Получить размеры позволяют следующие методы:

- ❑ `getWidth()` — возвращает ширину окна (включая толщину границ). **Формат метода:**

```
public final double getWidth()
```

- ❑ `getHeight()` — возвращает высоту окна (включая высоту заголовка и толщину нижней границы). **Формат метода:**

```
public final double getHeight()
```

- ❑ `getMinWidth()` — возвращает минимальную ширину окна. **Формат метода:**

```
public final double getMinWidth()
```

- ❑ `getMinHeight()` — возвращает минимальную высоту окна. **Формат метода:**

```
public final double getMinHeight()
```

- ❑ `getMaxWidth()` — возвращает максимальную ширину окна. **Формат метода:**

```
public final double getMaxWidth()
```

- ❑ `getMaxHeight()` — возвращает максимальную высоту окна. **Формат метода:**

```
public final double getMaxHeight()
```

Изменять и получать размеры окна, а также отслеживать изменения значений, позволяют свойства. Получить ссылки на свойства можно с помощью следующих методов:

- ❑ `widthProperty()` — возвращает свойство, связанное с шириной. **Формат метода:**

```
public final ReadOnlyDoubleProperty widthProperty()
```

- ❑ `heightProperty()` — возвращает свойство, связанное с высотой. **Формат метода:**

```
public final ReadOnlyDoubleProperty heightProperty()
```

- ❑ `minWidthProperty()` — возвращает свойство, связанное с минимальной шириной. **Формат метода:**

```
public final DoubleProperty minWidthProperty()
```

- `minHeightProperty()` — возвращает свойство, связанное с минимальной высотой. Формат метода:

```
public final DoubleProperty minHeightProperty()
```

- `maxWidthProperty()` — возвращает свойство, связанное с максимальной шириной. Формат метода:

```
public final DoubleProperty maxWidthProperty()
```

- `maxHeightProperty()` — возвращает свойство, связанное с максимальной высотой. Формат метода:

```
public final DoubleProperty maxHeightProperty()
```

Пример отслеживания изменения ширины окна:

```
stage.widthProperty().addListener((obj, oldValue, newValue) -> {  
    System.out.println("Ширина изменилась: " + newValue);  
});
```

Если необходимо, чтобы пользователь не мог изменить размеры окна с помощью мыши, то следует вызвать метод `setResizable()` и передать ему значение `false`. Кнопка **Развернуть** в заголовке окна при этом станет неактивной. Формат метода:

```
public final void setResizable(boolean value)
```

Проверить возможность изменения размеров позволяет метод `isResizable()`. Метод возвращает значение `true`, если пользователь может изменить размеры окна, и `false` — в противном случае. Формат метода:

```
public final boolean isResizable()
```

Получить ссылку на свойство можно с помощью метода `resizableProperty()`. Формат метода:

```
public final BooleanProperty resizableProperty()
```

ЭЛЕКТРОННЫЙ АРХИВ

Пример изменения и получения размеров окна содержится в проекте `FX_CH02_03_size`, приведенном в каталоге `CH02_windows` сопровождающего книгу электронного архива (см. приложение).

2.4. Местоположение окна на экране

Задать местоположение окна на экране монитора позволяют следующие методы:

- `setX()` — задает положение левого верхнего угла окна по оси *x*. Формат метода:

```
public final void setX(double value)
```

- `setY()` — задает положение левого верхнего угла окна по оси *y*. Формат метода:

```
public final void setY(double value)
```

- ❑ `centerOnScreen()` — выравнивает окно примерно по центру экрана. Формат метода:

```
public void centerOnScreen()
```

ОБРАТИТЕ ВНИМАНИЕ!

Начало координат расположено в левом верхнем углу экрана. Положительная ось *X* направлена вправо, а положительная ось *Y* — вниз.

Получить позицию окна позволяют следующие методы:

- ❑ `getX()` — возвращает координату левого верхнего угла окна по оси *X*. Формат метода:

```
public final double getX()
```

- ❑ `getY()` — возвращает координату левого верхнего угла окна по оси *Y*. Формат метода:

```
public final double getY()
```

Получить ссылки на свойства позволяют следующие методы:

- ❑ `xProperty()` — координата левого верхнего угла окна по оси *X*. Формат метода:

```
public final ReadOnlyDoubleProperty xProperty()
```

- ❑ `yProperty()` — координата левого верхнего угла окна по оси *Y*. Формат метода:

```
public final ReadOnlyDoubleProperty yProperty()
```

Пример получения значений:

```
System.out.println("x: " + parent.getX() + " y: " +  
    parent.getY());  
System.out.println("x: " + parent.xProperty().get() + " y: " +  
    parent.yProperty().get());
```

Пример назначения обработчика события изменения положения окна:

```
stage.xProperty().addListener((obj, oldValue, newValue) -> {  
    System.out.println("Координата X изменилась: " + newValue);  
});  
stage.yProperty().addListener((obj, oldValue, newValue) -> {  
    System.out.println("Координата Y изменилась: " + newValue);  
});
```

2.4.1. Класс *Screen*: размеры экрана

Для отображения окна у правой или нижней границы экрана необходимо знать его размеры. Для получения доступа к экрану предназначены следующие статические методы из класса `Screen` (полный список методов смотрите в документации):

```
// Модуль javafx.graphics  
import javafx.stage.Screen;
```

- ❑ `getPrimary()` — возвращает ссылку на основной экран. Формат метода:

```
public static Screen getPrimary()
```

- ❑ `getScreens()` — возвращает список всех экранов. Формат метода:

```
public static ObservableList<Screen> getScreens()
```

Выведем количество экранов:

```
System.out.println("Экранов: " + Screen.getScreens().size());
```

Получить характеристики экрана позволяют следующие методы из класса `Screen`:

- ❑ `getDpi()` — возвращает разрешение экрана. Формат метода:

```
public final double getDpi()
```

- ❑ `getBounds()` — возвращает объект `Rectangle2D` с характеристиками всего экрана. Формат метода:

```
import javafx.geometry.Rectangle2D;  
public final Rectangle2D getBounds()
```

- ❑ `getVisualBounds()` — возвращает объект `Rectangle2D` с характеристиками доступной части экрана (без панели задач). Формат метода:

```
public final Rectangle2D getVisualBounds()
```

Пример:

```
Screen screen = Screen.getPrimary();  
System.out.println("getDpi(): " + screen.getDpi());  
Rectangle2D bounds = screen.getBounds();  
System.out.println("getMinX(): " + bounds.getMinX());  
System.out.println("getMinY(): " + bounds.getMinY());  
System.out.println("getMaxX(): " + bounds.getMaxX());  
System.out.println("getMaxY(): " + bounds.getMaxY());  
System.out.println("getWidth(): " + bounds.getWidth());  
System.out.println("getHeight(): " + bounds.getHeight());
```

ЭЛЕКТРОННЫЙ АРХИВ

Пример изменения и получения положения окна на экране содержится в проекте `FX_CH02_04_Screen`, приведенном в каталоге `CH02_windows` сопровождающего книгу электронного архива (см. приложение).

2.5. Разворачивание и сворачивание окна

В заголовке окна расположены кнопки **Свернуть** и **Развернуть**, с помощью которых можно свернуть окно в значок на панели задач или максимально развернуть окно. Выполнить подобные действия из программы позволяют следующие методы:

- ❑ `setIconified()` — сворачивает окно на панель задач (значение `true`) или отменяет режим (значение `false`). Эквивалентно нажатию кнопки **Свернуть** в заголовке окна. Формат метода:

```
public final void setIconified(boolean value)
```

- ❑ `isIconified()` — возвращает `true`, если окно свернуто, и `false` — в противном случае. Формат метода:

```
public final boolean isIconified()
```

- ❑ `iconifiedProperty()` — возвращает свойство, связанное с режимом сворачивания окна. Формат метода:

```
public final ReadOnlyBooleanProperty iconifiedProperty()
```

- ❑ `setMaximized()` — разворачивает окно до максимального размера (значение `true`) или отменяет режим (значение `false`). Эквивалентно нажатию кнопки **Развернуть** в заголовке окна. Формат метода:

```
public final void setMaximized(boolean value)
```

- ❑ `isMaximized()` — возвращает `true`, если окно раскрыто до максимальных размеров, и `false` — в противном случае. Формат метода:

```
public final boolean isMaximized()
```

- ❑ `maximizedProperty()` — возвращает свойство, связанное с режимом максимального раскрытия окна. Формат метода:

```
public final ReadOnlyBooleanProperty maximizedProperty()
```

- ❑ `setFullScreen()` — включает (значение `true`) или отключает (значение `false`) полноэкранный режим отображения окна. Окно отображается без заголовка и границ во весь экран. Для выхода из режима следует нажать клавишу <Esc>. Формат метода:

```
public final void setFullScreen(boolean value)
```

- ❑ `isFullScreen()` — возвращает `true`, если включен полноэкранный режим, и `false` — в противном случае. Формат метода:

```
public final boolean isFullScreen()
```

- ❑ `fullScreenProperty()` — возвращает свойство, связанное с полноэкранным режимом. Формат метода:

```
public final ReadOnlyBooleanProperty fullScreenProperty()
```

- ❑ `setFullScreenExitHint()` — устанавливает текст подсказки пользователю для полноэкранного режима. Формат метода:

```
public final void setFullScreenExitHint(String value)
```

- ❑ `getFullScreenExitHint()` — возвращает текст подсказки пользователю для полноэкранного режима или значение `null`. Формат метода:

```
public final String getFullScreenExitHint()
```

- ❑ `setFullScreenExitKeyCombination()` — позволяет задать комбинацию клавиш для выхода из полноэкранного режима. По умолчанию используется клавиша <Esc>. Формат метода:

```
import javafx.scene.input.KeyCombination;
public final void setFullScreenExitKeyCombination(
    KeyCombination keyCombination)
```

Пример назначения клавиши <Esc>:

```
window.setFullScreenExitKeyCombination(
    KeyCombination.valueOf("Esc"));
```

- `getFullScreenExitKeyCombination()` — возвращает комбинацию клавиш для выхода из полноэкранного режима или значение `null`. Формат метода:

```
public final KeyCombination getFullScreenExitKeyCombination()
```

ЭЛЕКТРОННЫЙ АРХИВ

Пример разворачивания и сворачивания окна содержится в проекте `FX_CH02_05_FullScreen`, приведенном в каталоге `CH02_windows` сопровождающего книгу электронного архива (см. приложение).

2.6. Управление фокусом ввода и положением окна по оси Z

После выполнения некоторых операций бывает необходимо передать фокус ввода другому окну (сделать окно активным) и переместить его на самый верх в стопке других окон, чтобы пользователь обратил на окно внимание. Для этого предназначены следующие методы:

- `requestFocus()` — устанавливает фокус ввода. В Windows, если окно было ранее свернуто в значок на панель задач, то при получении фокуса ввода оно автоматически не будет отображено на экране. В этом случае станет активным только значок на панели задач. Формат метода:

```
public final void requestFocus()
```

- `isFocused()` — возвращает `true`, если окно имеет фокус ввода, и `false` — в противном случае. Формат метода:

```
public final boolean isFocused()
```

- `focusedProperty()` — возвращает ссылку на свойство, связанное с фокусом ввода. Формат метода:

```
public final ReadOnlyBooleanProperty focusedProperty()
```

- `toFront()` — перемещает окно на передний план. Формат метода:

```
public void toFront()
```

- `toBack()` — перемещает окно на задний план. Формат метода:

```
public void toBack()
```

- `setAlwaysOnTop()` — включает (значение `true`) или отключает (значение `false`) режим отображения окна поверх всех остальных окон. Формат метода:

```
public final void setAlwaysOnTop(boolean value)
```

- `isAlwaysOnTop()` — возвращает `true`, если активен режим отображения окна поверх всех других окон, и `false` — в противном случае. Формат метода:

```
public final boolean isAlwaysOnTop()
```

- `alwaysOnTopProperty()` — возвращает ссылку на свойство, связанное с режимом отображения окна поверх всех других окон. Формат метода:

```
public final ReadOnlyBooleanProperty alwaysOnTopProperty()
```

ЭЛЕКТРОННЫЙ АРХИВ

Пример управления фокусом ввода и положением окна содержится в проекте `FX_CH02_06_Focus`, приведенном в каталоге `CH02_windows` сопровождающего книгу электронного архива (см. приложение).

2.7. Управление прозрачностью окна

Сделать окно и все его содержимое полупрозрачным позволяет метод `setOpacity()` из класса `Window`. Формат метода:

```
public final void setOpacity(double value)
```

В качестве параметра указывается вещественное число от 0.0 до 1.0. Число 0.0 соответствует полностью прозрачному окну, а число 1.0 — отсутствию прозрачности. Для получения степени непрозрачности окна из программы предназначен метод `getOpacity()`, который возвращает вещественное число от 0.0 до 1.0. Формат метода:

```
public final double getOpacity()
```

Получить ссылку на свойство позволяет метод `opacityProperty()`. Формат метода:

```
public final DoubleProperty opacityProperty()
```

Выведем окно со степенью непрозрачности 0.7, а при нажатии кнопки изменим степень прозрачности и обработаем событие изменения прозрачности (листинг 2.3).

Листинг 2.3. Полупрозрачное окно

```
public void newWindow() {
    Stage window = new Stage();
    window.setOpacity(0.7);
    BorderPane pane = new BorderPane();
    Button btn = new Button("Изменить прозрачность");
    pane.setCenter(btn);
    btn.setOnAction(event -> {
        if (window.getOpacity() > 0.6) window.setOpacity(0.5);
        else window.setOpacity(0.7);
    });
    window.opacityProperty().addListener(
        (obj, oldValue, newValue) -> {
```

```
        System.out.println("Изменение прозрачности " + newValue);
    });
    window.setScene(new Scene(pane, 400, 250));
    window.setTitle("Полупрозрачное окно");
    window.show();
}
```

Полупрозрачные окна поддерживаются не всеми платформами. Проверить возможность изменения прозрачности окна позволяет следующий код:

```
// import javafx.application.ConditionalFeature;
// import javafx.application.Platform;
if (Platform.isSupported(ConditionalFeature.TRANSPARENT_WINDOW)) {
    System.out.println("Полупрозрачные окна поддерживаются");
}
else {
    System.out.println("Нет поддержки");
}
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH02_07_opacity`, приведенном в каталоге `CH02_windows` сопровождающего книгу электронного архива (см. приложение).

2.8. Модальные окна

Модальным называется окно, которое не позволяет взаимодействовать с другими окнами в том же приложении. Пока модальное окно не будет закрыто, сделать активным другое окно нельзя. Например, если в программе Microsoft Word выбрать пункт меню **Файл | Сохранить как**, то откроется модальное диалоговое окно, позволяющее выбрать путь и название файла. Пока это окно не будет закрыто, вы не сможете взаимодействовать с главным окном приложения.

Указать, что окно является модальным, позволяет метод `initModality()` из класса `Stage`. Формат метода:

```
public final void initModality(Modality modality)
```

В качестве параметра могут быть указаны следующие константы из перечисления `Modality`:

```
import javafx.stage.Modality;
```

- ☐ `NONE` — окно не является модальным;
- ☐ `WINDOW_MODAL` — окно блокирует только родительские окна в пределах иерархии. При использовании этого значения необходимо указать ссылку на родительское окно с помощью метода `initOwner()`;
- ☐ `APPLICATION_MODAL` — окно блокирует все окна приложения.

Получить текущее значение позволяет метод `getModality()`. Формат метода:

```
public final Modality getModality()
```

С помощью метода `initOwner()` можно указать ссылку на родительское окно. Если ссылка указана, то окно всегда будет отображаться поверх родительского окна. Значение по умолчанию: `null`. Формат метода:

```
public final void initOwner(Window owner)
```

Получить ссылку на родительское окно позволяет метод `getOwner()`. При отсутствии родительского окна метод возвращает значение `null`. Формат метода:

```
public final Window getOwner()
```

Создадим метод, с помощью которого можно открыть различные модальные и не модальные окна (листинг 2.4).

Листинг 2.4. Модальные окна

```
public void newWindow(Stage parent, Modality modality) {
    Stage window = new Stage();
    window.initModality(modality);
    window.initOwner(parent);
    BorderPane pane = new BorderPane();
    Button btn = new Button("Закрыть окно");
    pane.setCenter(btn);
    btn.setOnAction(event -> {
        window.close();
    });
    window.setScene(new Scene(pane, 400, 250));
    window.setTitle("Новое окно");
    window.show();
}
```

Итак, метод `newWindow()` в качестве параметров принимает ссылку на родительское окно и константу из перечисления `Modality`. Если в первом параметре указать значение `null`, а во втором — `NONE`, то окно будет не модальным окном верхнего уровня:

```
newWindow(null, Modality.NONE);
```

Если в первом параметре вместо значения `null` указать ссылку на родительское окно, то окно будет всегда расположено поверх родительского окна:

```
newWindow(stage, Modality.NONE);
```

Чтобы окно блокировало только родительское окно, в первом параметре передаем ссылку на родительское окно, а во втором параметре — константу `WINDOW_MODAL`:

```
newWindow(stage, Modality.WINDOW_MODAL);
```

При этом получить доступ к родительскому окну можно только при закрытии модального окна, а другие окна заблокированы не будут. Если ссылку на родительское окно не указать, то окно не будет модальным.

Чтобы блокировать все окна приложения, следует во втором параметре указать константу `APPLICATION_MODAL`:

```
newWindow(null, Modality.APPLICATION_MODAL);
```

Модальные окна в большинстве случаев являются диалоговыми. Для работы с диалоговыми окнами, начиная с версии 8.40, предназначен класс `Dialog<R>`. Подробно этот класс и классы, наследующие класс `Dialog<R>`, мы рассмотрим в главе 16.

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH02_08_modality`, приведенном в каталоге `CH02_windows` сопровождающего книгу электронного архива (см. приложение).

2.9. Смена значка в заголовке окна

По умолчанию в левом верхнем углу окна отображается стандартный значок. Чтобы сменить значок в заголовке окна, необходимо выполнить следующие действия:

1. Создать в редакторе значок размером 16 на 16 или 32 на 32 пиксела в формате PNG, BMP, GIF или JPEG и поместить его, например, в каталог `src\img`, как мы и будем делать в примере.
2. Получить путь к значку с помощью метода `getResource()` или создать поток с помощью метода `getResourceAsStream()`.
3. Передать путь или поток конструктору класса `Image`.
4. Вызвать метод `getIcons()` объекта окна и добавить объект `Image` в список с помощью метода `add()`. Форматы методов:

```
public final ObservableList<Image> getIcons()  
public boolean add(E e)
```

В качестве примера создадим значок в формате PNG размером 32 на 32 пиксела с названием `blue32x32.png` и сохраним его в каталоге `src\img`, а затем установим этот значок для окна (листинг 2.5).

Листинг 2.5. Смена значка в заголовке окна

```
package application;  
  
import javafx.application.Application;  
import javafx.scene.image.Image;  
import javafx.stage.Stage;  
  
public class Main extends Application {  
    public static void main(String[] args) {
```

```

        Application.launch(args);
    }

    @Override
    public void start(Stage stage) throws Exception {
        stage.setTitle("Смена значка в заголовке окна");
        try {
            stage.getIcons().add(new Image(
                getClass().getResourceAsStream("/img/blue32x32.png")));
            /*stage.getIcons().add(new Image(
                getClass().getResource(
                    "/img/blue32x32.png").toExternalForm()));*/
        } catch (Exception e) {
            System.out.println("Не удалось загрузить значок");
        }
        stage.show();
    }
}

```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH02_09_icons`, приведенном в каталоге `CH02_windows` сопровождающего книгу электронного архива (см. приложение).

2.10. Изменение цвета фона

Под изменением цвета фона окна подразумевается изменение цвета фона контейнера, отображаемого внутри окна. Вначале идет контейнер верхнего уровня, реализуемый классом `Scene`, а затем над ним другие контейнеры, внутри которых размещаются компоненты.

ПРИМЕЧАНИЕ

Подробно изменение фона контейнеров и компонентов мы рассмотрим в разд. 3.5.

Изменить цвет фона обычного контейнера проще всего с помощью стиля `-fx-background-color`. Указать стиль можно в отдельном файле, а затем подключить его с помощью метода `getStylesheets()` из класса `Scene`. Пример указания стиля:

```

.root {
    -fx-background-color: green;
}

```

Можно также указать стиль с помощью метода `setStyle()`. Формат метода:

```
public final void setStyle(String value)
```

Пример изменения цвета фона контейнера:

```

VBox root = new VBox(15);
root.setStyle("-fx-background-color: green;");

```

Если необходимо сделать цвет фона контейнера прозрачным, то можно воспользоваться следующим кодом:

```
// import javafx.scene.layout.Background;
root.setBackground(Background.EMPTY);
```

Указать цвет фона контейнера, реализуемого классом `Scene`, можно с помощью конструкторов класса или с помощью метода `setFill()`. Форматы конструкторов и метода `setFill()`:

```
public Scene(Parent root, double width, double height, Paint fill)
public Scene(Parent root, Paint fill)
public final void setFill(Paint value)
```

Пример указания цвета в конструкторе класса `Scene`:

```
// import javafx.scene.paint.Color;
Scene scene = new Scene(pane, 400, 250, Color.BLACK);
```

Пример использования метода `setFill()`:

```
scene.setFill(Color.BLUE);
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH02_10_background`, приведенном в каталоге `CH02_windows` сопровождающего книгу электронного архива (см. приложение).

2.11. Использование изображения в качестве фона

В качестве фона контейнера можно использовать изображение. Для этого применяется стиль `-fx-background-image`. Пример указания стиля:

```
-fx-background-image: url(/img/bg_image.png);
```

По умолчанию изображение повторяется по горизонтали и вертикали, чтобы заполнить всю поверхность контейнера. Изменить это поведение можно с помощью стиля `-fx-background-repeat`. Пример запрета повтора фонового изображения:

```
-fx-background-repeat: no-repeat;
```

Пример указания стилей (изображение `bg_image.png` должно быть расположено в каталоге `src\img` проекта):

```
VBox root = new VBox(15);
root.setStyle(
    "-fx-background-image: url(/img/bg_image.png); " +
    "-fx-background-repeat: no-repeat;");
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH02_11_backgroundImage`, приведенном в каталоге `CH02_windows` сопровождающего книгу электронного архива (см. приложение).

2.12. Создание окна произвольной формы

Чтобы создать окно произвольной формы, следует выполнить следующие шаги:

1. При создании окна использовать стиль `TRANSPARENT`:

```
Stage window = new Stage(StageStyle.TRANSPARENT);
```

2. Создать объект сцены с прозрачным фоном (`Color.TRANSPARENT`):

```
window.setScene(new Scene(pane, 400, 300, Color.TRANSPARENT));
```

3. После создания контейнера сделать его фон прозрачным с помощью следующего кода:

```
pane.setBackground(Background.EMPTY);
```

4. Нарисовать фон (например, с помощью SVG-графики или на основе изображения) и добавить его в контейнер.

В качестве примера создадим окно произвольной формы с кнопкой, с помощью которой можно закрыть окно (листинг 2.6).

Листинг 2.6. Создание окна произвольной формы

```
public void newWindowTransparent() {
    Stage window = new Stage(StageStyle.TRANSPARENT);
    SVGPath svg = new SVGPath();
    svg.setContent("M 50 50 L 350 50 L 300 250 L 200 200 L 100 250 z");
    svg.setFill(new Color(0.0, 1.0, 0.5, 1.0));
    svg.setEffect(new DropShadow());
    window.setTitle("Окно произвольной формы");
    StackPane pane = new StackPane();
    pane.setBackground(Background.EMPTY);
    Button btn = new Button("Закрыть окно");
    pane.getChildren().addAll(svg, btn);
    btn.setOnAction(event -> {
        window.close();
    });
    window.setScene(new Scene(pane, 400, 300, Color.TRANSPARENT));
    window.show();
}
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH02_12_freeForm`, а пример создания окна произвольной формы с помощью масок — в проекте `FX_CH05_07_clip`, приведенных в каталогах `CH02_windows` и `CH05_graphics` сопровождающего книгу электронного архива (см. приложение).

2.13. Заккрытие окна из программы

Заккрыть окно из программы позволяет метод `close()` из класса `Stage`. Формат метода:

```
public void close()
```

Мы уже много раз использовали этот метод для закрытия окна при нажатии кнопки, например, в листинге 2.6:

```
window.close();
```

Пользователь может закрыть окно, нажав кнопку **Заккрыть** в заголовке окна. Если необходимо перехватить и обработать закрытие окна этим способом, то внутри программы следует назначить обработчик события с помощью метода `setOnCloseRequest()`. Формат метода:

```
public final void setOnCloseRequest(EventHandler<WindowEvent> value)
```

Обработчик принимает один параметр, имеющий тип `WindowEvent`. Если внутри обработчика через объект события вызвать метод `consume()`, то окно закрыто не будет. В качестве примера создадим окно с надписью и кнопкой (листинг 2.7). По нажатию этой кнопки закроем окно с помощью метода `close()`. По нажатию кнопки **Заккрыть** в заголовке окна перехватим это событие и выведем диалоговое окно с запросом закрытия окна. Если пользователь нажмет кнопку **ОК**, то закроем окно, а если кнопку **Cancel**, то оставим окно открытым и изменим текст надписи.

Листинг 2.7. Заккрытие окна из программы

```
@Override
public void start(Stage stage) throws Exception {
    VBox root = new VBox(15);
    root.setAlignment(Pos.CENTER);
    Button button = new Button("Заккрыть окно");
    Label label = new Label(
        "Нажмите кнопку Заккрыть в заголовке окна");
    root.getChildren().addAll(label, button);
    button.setOnAction(event -> {
        stage.close();
    });
    stage.setOnCloseRequest(event -> {
        Alert dialog = new Alert(AlertType.CONFIRMATION,
            "Вы действительно хотите закрыть окно?");
        dialog.setTitle("Заккрытие окна");
        dialog.setHeaderText(null);
        Optional<ButtonType> result = dialog.showAndWait();
        if (result.isPresent() && result.get() == ButtonType.OK) {
            System.out.println("Окно будет закрыто");
        }
    });
}
```

```
        else {
            label.setText("Окно не будет закрыто!");
            event.consume();
        }
    });
    stage.setTitle("Закрытие окна");
    stage.setScene(new Scene(root, 400, 150));
    stage.show();
}
```

В этом примере мы воспользовались классом `Alert` для вывода модального диалогового окна с двумя кнопками: **ОК** и **Cancel**. Обратите внимание: класс `Alert` доступен только с версии 8.40. Подробно диалоговые окна мы рассмотрим в *главе 16*.

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH02_13_close`, приведенном в каталоге `CH02_windows` сопровождающего книгу электронного архива (см. *приложение*).

ГЛАВА 3



Размещение компонентов в окне

При размещении в окне нескольких компонентов обычно возникает вопрос взаимного их расположения, а также их минимальных размеров. Следует помнить, что по умолчанию размеры окна можно изменить, взявшись мышью на границу окна, а значит, необходимо перехватывать событие изменения размеров и производить перерасчет позиции и размера каждого компонента. JavaFX избавляет нас от лишних проблем и предоставляет множество *компонентов-контейнеров*, которые производят перерасчет автоматически. Все, что от нас требуется, — это выбрать нужный контейнер, добавить туда компоненты в определенном порядке, а затем поместить контейнер на сцену или в другой контейнер.

3.1. Класс *Scene*

Прежде чем использовать контейнеры, необходимо создать контейнер самого верхнего уровня, который мы будем называть *сценой*. Реализуется сцена с помощью класса *Scene*. Инструкция импорта класса:

```
// Модуль javafx.graphics
import javafx.scene.Scene;
```

3.1.1. Создание объекта

Создать объект класса *Scene* позволяют следующие конструкторы:

```
Scene(Parent root)
Scene(Parent root, double width, double height)
Scene(Parent root, double width, double height, Paint fill)
Scene(Parent root, Paint fill)
Scene(Parent root, double width, double height, boolean depthBuffer)
Scene(Parent root, double width, double height, boolean depthBuffer,
      SceneAntialiasing antiAliasing)
```

Параметр *root* задает ссылку на контейнер верхнего уровня. Класс *Parent* наследуют не только классы контейнеров, но и классы компонентов, поэтому вместо кон-

тейнера можно добавить и компонент, например кнопку. В этом случае кнопка будет иметь размер, равный внутреннему размеру окна. Так как можно указать ссылку только на один компонент, добавить другие компоненты в этом случае будет нельзя. Поэтому в качестве элемента верхнего уровня используются именно контейнеры. Пример:

```
BorderPane pane = new BorderPane();
Button btn = new Button("Закрыть окно");
pane.setCenter(btn);
Scene scene = new Scene(pane);
```

Изменить ссылку на контейнер верхнего уровня после создания объекта сцены позволяет метод `setRoot()`. Формат метода:

```
public final void setRoot(Parent value)
```

Получить ссылку на контейнер верхнего уровня можно с помощью метода `getRoot()`. Формат метода:

```
public final Parent getRoot()
```

Параметры `width` и `height` задают начальные размеры сцены. Если размеры окна и сцены не указаны, то размеры окна и сцены будут соответствовать внутреннему содержимому окна. Пример указания ширины и высоты сцены:

```
Scene scene = new Scene(pane, 400, 250);
```

Параметр `fill` задает параметры фона сцены. В качестве значения можно указать объекты классов `Color` (сплошной цвет), `ImagePattern` (фоновое изображение), `LinearGradient` (линейный градиент) и `RadialGradient` (радиальный градиент). Обратите внимание: чтобы увидеть результат, у контейнера верхнего уровня должен быть прозрачный фон. Установить прозрачный фон позволяет следующий код:

```
pane.setBackground(Background.EMPTY);
```

Пример указания красного цвета фона:

```
Scene scene = new Scene(pane, 300, 150, Color.RED);
```

Если в параметре `depthBuffer` указано значение `true`, то объект сцены будет иметь буфер глубины. Этот параметр используется для работы с трехмерным пространством. Проверить наличие буфера позволяет метод `isDepthBuffer()`. Формат метода:

```
public final boolean isDepthBuffer()
```

Параметр `antiAliasing` задает уровень сглаживания. Значение по умолчанию: `SceneAntialiasing.DISABLED`. Получить значение уровня сглаживания позволяет метод `getAntiAliasing()`. Формат метода:

```
import javafx.scene.SceneAntialiasing;
public final SceneAntialiasing getAntiAliasing()
```

После создания сцены ее необходимо прикрепить к окну с помощью метода `setScene()`. Формат метода:

```
public final void setScene(Scene value)
```

Получить ссылку на сцену через объект окна позволяет метод `getScene()` из класса `Window`. Формат метода:

```
public final Scene getScene()
```

Чтобы получить ссылку на окно через объект сцены, можно воспользоваться методом `getWindow()` из класса `Scene`. Формат метода:

```
public final Window getWindow()
```

3.1.2. Размеры сцены

Начальные размеры сцены задаются с помощью параметров `width` и `height` в конструкторе класса `Scene`. Если размеры окна и сцены не указаны, то размеры окна и сцены будут соответствовать внутреннему содержимому окна. Получить текущие размеры сцены позволяют следующие методы:

- ❑ `getWidth()` — возвращает ширину сцены. Формат метода:

```
public final double getWidth()
```

- ❑ `getHeight()` — возвращает высоту сцены. Формат метода:

```
public final double getHeight()
```

- ❑ `widthProperty()` — возвращает ссылку на свойство, которое связано с шириной сцены. Формат метода:

```
public final ReadOnlyDoubleProperty widthProperty()
```

- ❑ `heightProperty()` — возвращает ссылку на свойство, которое связано с высотой сцены. Формат метода:

```
public final ReadOnlyDoubleProperty heightProperty()
```

3.1.3. Местоположение сцены

Получить координаты левого верхнего угла сцены относительно левого верхнего угла окна позволяют следующие методы:

- ❑ `getX()` — возвращает координату по оси `x`. Благодаря этой координате мы можем узнать толщину границы окна. Формат метода:

```
public final double getX()
```

- ❑ `getY()` — возвращает координату по оси `y`. Благодаря этой координате мы можем узнать высоту заголовка окна. Формат метода:

```
public final double getY()
```

- ❑ `xProperty()` — возвращает ссылку на свойство, которое связано с координатой `x`. Формат метода:

```
public final ReadOnlyDoubleProperty xProperty()
```

- ❑ `yProperty()` — возвращает ссылку на свойство, которое связано с координатой `y`. Формат метода:

```
public final ReadOnlyDoubleProperty yProperty()
```

ОБРАТИТЕ ВНИМАНИЕ!

Координаты будут доступны только после отображения окна. До этого момента они равны 0.0.

3.1.4. Фон сцены

Указать характеристики фона сцены можно через параметр `fill` в конструкторе класса. После создания объекта используются следующие методы:

- ❑ `setFill()` — задает характеристики фона. Формат метода:

```
public final void setFill(Paint value)
```

- ❑ `getFill()` — возвращает характеристики фона. Формат метода:

```
public final Paint getFill()
```

- ❑ `fillProperty()` — возвращает ссылку на свойство, которое связано с характеристиками фона. Формат метода:

```
public final ObjectProperty<Paint> fillProperty()
```

В качестве параметра метода `setFill()` можно указать:

- ❑ объект класса `Color` — задает сплошной цвет фона. Пример:

```
newWindow(Color.RED);
```

- ❑ объект класса `ImagePattern` — устанавливает фоновое изображение. Пример:

```
try {
    ImagePattern im = new ImagePattern(new Image(
        getClass().getResourceAsStream("/img/bg_image.png")));
    newWindow(im);
} catch (Exception e) {
    System.out.println("Не удалось загрузить изображение");
}
```

- ❑ объект класса `LinearGradient` — заливка линейным градиентом. Пример:

```
Stop[] stops = new Stop[] {
    new Stop(0, Color.BLACK), new Stop(1, Color.WHITE)
};
LinearGradient lg = new LinearGradient(0, 0, 1, 0, true,
    CycleMethod.NO_CYCLE, stops);
newWindow(lg);
```

- ❑ объект класса `RadialGradient` — радиальный градиент. Пример:

```
RadialGradient rg = RadialGradient.valueOf(
    "center 100px 100px, radius 300px, red 0%, " +
    "blue 50%, black 100%");
newWindow(rg);
```

Обратите внимание: чтобы увидеть результат, у контейнера верхнего уровня должен быть прозрачный фон. Установить прозрачный фон позволяет следующий код:

```
pane.setBackground(Background.EMPTY);
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH03_01_Scene`, приведенном в каталоге `CH03_containers` сопровождающего книгу электронного архива (см. приложение).

3.2. Размеры контейнеров и компонентов

После создания сцены и прикрепления контейнера верхнего уровня можно размещать внутри сцены компоненты и другие контейнеры. Все классы контейнеров и компонентов являются наследниками класса `Node`, поэтому в дальнейшем мы их будем называть одним словом — *узлы*. Большинство классов являются также наследниками классов `Parent` и `Region`. Пример иерархии наследования для контейнера `HBox`:

```
Object — Node — Parent — Region — Pane — HBox
```

Пример иерархии для компонента `Button` (командная кнопка):

```
Object — Node — Parent — Region — Control — Labeled —  
      ButtonBase — Button
```

Прежде чем мы начнем рассматривать контейнеры и компоненты, необходимо изучить некоторые общие характеристики. Начнем с размеров узлов. Указать размеры позволяют следующие методы из класса `Region`:

- ❑ `setPrefWidth()` — задает предпочтительную ширину узла. Значение по умолчанию: `Region.USE_COMPUTED_SIZE` (равно `-1.0`). Формат метода:

```
public final void setPrefWidth(double value)
```
- ❑ `setPrefHeight()` — задает предпочтительную высоту узла. Значение по умолчанию: `Region.USE_COMPUTED_SIZE` (равно `-1.0`). Формат метода:

```
public final void setPrefHeight(double value)
```
- ❑ `setPrefSize()` — задает одновременно предпочтительную ширину и высоту. Формат метода:

```
public void setPrefSize(double prefWidth, double prefHeight)
```
- ❑ `setMinWidth()` — задает минимальную ширину узла. Значение по умолчанию: `Region.USE_COMPUTED_SIZE` (равно `-1.0`). Формат метода:

```
public final void setMinWidth(double value)
```
- ❑ `setMinHeight()` — задает минимальную высоту узла. Значение по умолчанию: `Region.USE_COMPUTED_SIZE` (равно `-1.0`). Формат метода:

```
public final void setMinHeight(double value)
```

- ❑ `setMinSize()` — задает одновременно минимальную ширину и высоту. Формат метода:

```
public void setMinSize(double minWidth, double minHeight)
```

- ❑ `setMaxWidth()` — задает максимальную ширину узла. Значение по умолчанию: `Region.USE_COMPUTED_SIZE` (равно `-1.0`). Формат метода:

```
public final void setMaxWidth(double value)
```

- ❑ `setMaxHeight()` — задает максимальную высоту узла. Значение по умолчанию: `Region.USE_COMPUTED_SIZE` (равно `-1.0`). Формат метода:

```
public final void setMaxHeight(double value)
```

- ❑ `setMaxSize()` — задает одновременно максимальную ширину и высоту. Формат метода:

```
public void setMaxSize(double maxWidth, double maxHeight)
```

В качестве значения можно указать вещественное число или одну из статических констант из класса `Region`:

- ❑ `USE_COMPUTED_SIZE` — указывает, что значение не задано. Формат:

```
public static final double USE_COMPUTED_SIZE
```

Пример вывода значения константы:

```
System.out.println(Region.USE_COMPUTED_SIZE); // -1.0
```

- ❑ `USE_PREF_SIZE` — указывает, что должны использоваться предпочтительные размеры. Применяется только при указании минимального или максимального размера. Формат:

```
public static final double USE_PREF_SIZE
```

Пример вывода значения константы:

```
System.out.println(Region.USE_PREF_SIZE); // -Infinity
```

При указании максимальных значений можно задать значение `Double.MAX_VALUE`. Например, если это значение указать для максимальной ширины кнопки, то кнопка займет всю доступную ширину контейнера. Пример:

```
button.setMaxWidth(Double.MAX_VALUE);
```

При использовании программы **Scene Builder** размеры узлов задаются на вкладке **Layout** в разделе **Size**. Можно также выделить узел и изменять его размеры, взявшись мышью за маркеры, расположенные по углам и посередине всех сторон. При этом будут автоматически изменяться значения предпочтительных размеров на вкладке **Layout** в разделе **Size**.

Пример указания размеров контейнера и компонента в FXML-файле:

```
<VBox maxHeight="-Infinity" maxWidth="-Infinity" minHeight="-Infinity"
  minWidth="-Infinity" prefHeight="200.0" prefWidth="300.0"
  xmlns="http://javafx.com/javafx/10.0.1"
  xmlns:fx="http://javafx.com/fxml/1">
```

```
<children>
  <Button maxWidth="1.7976931348623157E308"
    mnemonicParsing="false" prefWidth="200.0" text="OK" />
</children>
</VBox>
```

Получить размеры позволяют следующие методы из класса `Region`:

- ❑ `getWidth()` и `getHeight()` — возвращают текущую ширину и высоту соответственно. Эти размеры вычисляются менеджерами компоновки и до момента отображения окна имеют значение 0.0. Форматы методов:

```
public final double getWidth()
public final double getHeight()
```

- ❑ `getPrefWidth()` и `getPrefHeight()` — возвращают предпочтительную ширину и высоту соответственно. Форматы методов:

```
public final double getPrefWidth()
public final double getPrefHeight()
```

- ❑ `getMinWidth()` и `getMinHeight()` — возвращают минимальную ширину и высоту соответственно. Форматы методов:

```
public final double getMinWidth()
public final double getMinHeight()
```

- ❑ `getMaxWidth()` и `getMaxHeight()` — возвращают максимальную ширину и высоту соответственно. Форматы методов:

```
public final double getMaxWidth()
public final double getMaxHeight()
```

Получить ссылки на свойства можно с помощью следующих методов из класса `Region`:

- ❑ `widthProperty()` и `heightProperty()` — текущая ширина и высота соответственно. Эти размеры вычисляются менеджерами компоновки и до момента отображения окна имеют значение 0.0. Форматы методов:

```
public final ReadOnlyDoubleProperty widthProperty()
public final ReadOnlyDoubleProperty heightProperty()
```

- ❑ `prefWidthProperty()` и `prefHeightProperty()` — предпочтительная ширина и высота соответственно. Форматы методов:

```
public final DoubleProperty prefWidthProperty()
public final DoubleProperty prefHeightProperty()
```

- ❑ `minWidthProperty()` и `minHeightProperty()` — минимальная ширина и высота соответственно. Форматы методов:

```
public final DoubleProperty minWidthProperty()
public final DoubleProperty minHeightProperty()
```

- ❑ `maxWidthProperty()` и `maxHeightProperty()` — максимальная ширина и высота соответственно. Форматы методов:

```
public final DoubleProperty maxWidthProperty()  
public final DoubleProperty maxHeightProperty()
```

Получить размеры области, в которую вписан узел, позволяют также свойства `layoutBounds`, `boundsInLocal` и `boundsInParent` из класса `Node`. Описание этих свойств и методов доступа к ним можно найти в *разд. 5.1.4*.

Помните, что в приложении могут использоваться разные стили оформления, в том числе и характеристики шрифта. Подогнав размеры под один стиль, можно прийти в ужас при виде приложения, в котором использован другой стиль, где размер шрифта в два раза больше. Поэтому лучше вообще отказаться от указания фиксированных размеров или задавать размер и название шрифта для каждого компонента. Кроме того, приложение может поддерживать несколько языков интерфейса. Длина слов в разных языках различается, что также станет причиной искажения компонентов. Старайтесь указывать ограничения там, где это действительно необходимо.

ЭЛЕКТРОННЫЙ АРХИВ

Пример установки и получения размеров содержится в проекте `FX_CH03_02_size`, приведенном в каталоге `CH03_containers` сопровождающего книгу электронного архива (см. *приложение*).

3.3. Внутренние отступы

При добавлении узлов в контейнер, поддерживающий автоматическое выравнивание, узлы будут соприкасаться с границами контейнера. Для управления внутренними отступами между границами контейнера и расположенными внутри контейнера узлами предназначены следующие методы из класса `Region`:

- ❑ `setPadding()` — задает внутренние отступы. Формат метода:

```
public final void setPadding(Insets value)
```
- ❑ `getPadding()` — возвращает внутренние отступы. Формат метода:

```
public final Insets getPadding()
```
- ❑ `paddingProperty()` — возвращает ссылку на свойство. Формат метода:

```
public final ObjectProperty<Insets> paddingProperty()
```

3.3.1. Класс *Insets*

Размер отступов задается с помощью класса `Insets`. Создать экземпляр класса позволяют следующие конструкторы:

```
import javafx.geometry.Insets;  
Insets(double topRightBottomLeft)  
Insets(double top, double right, double bottom, double left)
```

Первый конструктор задает одинаковое значение для всех четырех сторон:

```
Insets ins = new Insets(20.0);
System.out.println(ins);
// Insets [top=20.0, right=20.0, bottom=20.0, left=20.0]
```

Второй конструктор позволяет указать отступ от каждой границы по отдельности (первый параметр задает отступ от верхней границы и далее по часовой стрелке):

```
Insets ins = new Insets(10.0, 20.0, 30.0, 40.0);
System.out.println(ins);
// Insets [top=10.0, right=20.0, bottom=30.0, left=40.0]
```

Получить отдельные значения после создания объекта позволяют методы `getTop()` (отступ сверху), `getRight()` (отступ справа), `getBottom()` (отступ снизу) и `getLeft()` (отступ слева):

```
Insets ins = new Insets(10.0, 20.0, 30.0, 40.0);
System.out.println(ins.getTop()); // 10.0
System.out.println(ins.getRight()); // 20.0
System.out.println(ins.getBottom()); // 30.0
System.out.println(ins.getLeft()); // 40.0
```

Создать объект с нулевыми значениями позволяет статическая константа `EMPTY`:

```
public static final Insets EMPTY
```

Пример:

```
Insets ins = Insets.EMPTY;
System.out.println(ins);
// Insets [top=0.0, right=0.0, bottom=0.0, left=0.0]
```

ЭЛЕКТРОННЫЙ АРХИВ

Пример установки и получения значений внутренних отступов содержится в проекте `FX_CH03_03_padding`, приведенном в каталоге `CH03_containers` сопровождающего книгу электронного архива (см. *приложение*).

При использовании программы **Scene Builder** внутренние отступы задаются на вкладке **Layout** в разделе **Internal** (параметр **Padding**). Если в первое поле ввести значение и нажать на значок в форме стрелки, указывающей вправо, то остальные поля будут автоматически заполнены тем же самым значением.

В **FXML**-файле внутренние отступы задаются тегом `<padding>`, внутри которого вложен тег `<Insets>`. Пример указания внутренних отступов в **FXML**-файле:

```
<VBox prefHeight="200.0" prefWidth="300.0"
  xmlns="http://javafx.com/javafx/10.0.1"
  xmlns:fx="http://javafx.com/fxml/1">
  <children>
    <Button mnemonicParsing="false" text="Button">
      <padding>
        <Insets bottom="5.0" left="5.0" right="5.0" top="5.0" />
      </padding>
    </Button>
  </children>
</VBox>
```



```

        </Button>
    </children>
    <padding>
        <Insets bottom="30.0" left="40.0" right="20.0" top="10.0" />
    </padding>
</VBox>

```

3.4. Местоположение узлов внутри контейнера

Если мы используем контейнеры, которые не содержат менеджер компоновки (например, классы `Group` или `Pane`), то должны указать местоположение узла внутри контейнера, иначе после добавления они будут расположены друг над другом в левом верхнем углу контейнера. Указать местоположение позволяют следующие методы из класса `Node`:

- ❑ `setLayoutX()` — задает координату по оси *x* относительно левого верхнего угла контейнера. Формат метода:

```
public final void setLayoutX(double value)
```

- ❑ `setLayoutY()` — задает координату по оси *y*. Формат метода:

```
public final void setLayoutY(double value)
```

- ❑ `relocate()` — позволяет указать сразу две координаты. Формат метода:

```
public void relocate(double x, double y)
```

Получить координаты позволяют следующие методы из класса `Node`:

- ❑ `getLayoutX()` — возвращает координату по оси *x*. Формат метода:

```
public final double getLayoutX()
```

- ❑ `getLayoutY()` — возвращает координату по оси *y*. Формат метода:

```
public final double getLayoutY()
```

Получить ссылки на свойства позволяют следующие методы из класса `Node`:

- ❑ `layoutXProperty()` — координата по оси *x*. Формат метода:

```
public final DoubleProperty layoutXProperty()
```

- ❑ `layoutYProperty()` — координата по оси *y*. Формат метода:

```
public final DoubleProperty layoutYProperty()
```

Пример отслеживания изменения координаты по оси *x*:

```

button.layoutXProperty().addListener((obj, oldValue, newValue) -> {
    System.out.println("Изменение X " + newValue);
});

```

Временно сместить узел на некоторое расстояние относительно текущего местоположения позволяют методы `setTranslateX()` и `setTranslateY()`. Эти методы очень удобно использовать для создания анимации. Значения смещений прибавляются

к исходным координатам, заданным методами `setLayoutX()`, `setLayoutY()` или `relocate()`, и не изменяют исходные координаты. Чтобы вернуть узел на первоначальное место, достаточно присвоить смещению значение 0.0. Форматы методов:

```
public final void setTranslateX(double value)
public final void setTranslateY(double value)
```

Получить текущее значение смещения позволяют методы `getTranslateX()` и `getTranslateY()`. Изначально методы возвращают значение 0.0. Форматы методов:

```
public final double getTranslateX()
public final double getTranslateY()
```

Получить ссылки на свойства можно с помощью методов `translateXProperty()` и `translateYProperty()`. Форматы методов:

```
public final DoubleProperty translateXProperty()
public final DoubleProperty translateYProperty()
```

Получить координаты области, в которую вписан узел, позволяют также свойства `boundsInLocal` и `boundsInParent` из класса `Node`. Описание этих свойств и методов доступа к ним можно найти в *разд. 5.1.4*.

Для преобразования координат можно воспользоваться следующими методами из класса `Node`:

❑ `localToParent()` — преобразует локальные координаты узла в координаты родительского узла. Форматы метода:

```
public Point2D localToParent(double localX, double localY)
public Point2D localToParent(Point2D localPoint)
public Point3D localToParent(double x, double y, double z)
public Point3D localToParent(Point3D localPoint)
public Bounds localToParent(Bounds localBounds)
```

❑ `localToScene()` — преобразует локальные координаты узла в координаты сцены. Форматы метода:

```
public Point2D localToScene(double localX, double localY)
public Point2D localToScene(double x, double y,
                             boolean rootScene)
public Point2D localToScene(Point2D localPoint)
public Point2D localToScene(Point2D localPoint,
                             boolean rootScene)
public Point3D localToScene(double x, double y, double z)
public Point3D localToScene(double x, double y, double z,
                             boolean rootScene)
public Point3D localToScene(Point3D localPoint)
public Point3D localToScene(Point3D localPoint,
                             boolean rootScene)
public Bounds localToScene(Bounds localBounds)
public Bounds localToScene(Bounds localBounds, boolean rootScene)
```

- ❑ `localToScreen()` — преобразует локальные координаты узла в координаты экрана. **Форматы метода:**

```
public Point2D localToScreen(double localX, double localY)
public Point2D localToScreen(Point2D localPoint)
public Point2D localToScreen(double localX, double localY,
                             double localZ)
public Point2D localToScreen(Point3D localPoint)
public Bounds localToScreen(Bounds localBounds)
```

- ❑ `parentToLocal()` — преобразует координаты родителя в локальные координаты узла. **Форматы метода:**

```
public Point2D parentToLocal(double parentX, double parentY)
public Point2D parentToLocal(Point2D parentPoint)
public Point3D parentToLocal(double parentX, double parentY,
                             double parentZ)
public Point3D parentToLocal(Point3D parentPoint)
public Bounds parentToLocal(Bounds parentBounds)
```

- ❑ `sceneToLocal()` — преобразует координаты сцены в локальные координаты узла. **Форматы метода:**

```
public Point2D sceneToLocal(double sceneX, double sceneY)
public Point2D sceneToLocal(double x, double y,
                             boolean rootScene)
public Point2D sceneToLocal(Point2D scenePoint)
public Point2D sceneToLocal(Point2D point, boolean rootScene)
public Point3D sceneToLocal(double sceneX, double sceneY,
                             double sceneZ)
public Point3D sceneToLocal(Point3D scenePoint)
public Bounds sceneToLocal(Bounds sceneBounds)
public Bounds sceneToLocal(Bounds bounds, boolean rootScene)
```

- ❑ `screenToLocal()` — преобразует координаты экрана в локальные координаты узла. **Форматы метода:**

```
public Point2D screenToLocal(double screenX, double screenY)
public Point2D screenToLocal(Point2D screenPoint)
public Bounds screenToLocal(Bounds screenBounds)
```

При использовании программы Scene Builder координаты задаются на вкладке **Layout** в разделе **Position** (поля **Layout X** и **Layout Y**). Можно также просто переместить компонент на форме с помощью мыши, и координаты автоматически добавятся в соответствующие поля. Значения смещений задаются на вкладке **Layout** в разделе **Transforms** (поля **Translate X** и **Translate Y**). Пример указания координат в FXML-файле:

```
<Button layoutX="110.0" layoutY="175.0"
        mnemonicParsing="false" text="Button" />
```

ЭЛЕКТРОННЫЙ АРХИВ

Пример изменения и получения координат содержится в проекте `FX_CH03_04_relocate`, приведенном в каталоге `CH03_containers` сопровождающего книгу электронного архива (см. приложение).

3.5. Фон контейнеров и компонентов

Изменить цвет фона проще всего с помощью стиля `-fx-background-color`. Указать стиль можно в отдельном файле, а затем подключить его с помощью метода `getStylesheets()` из класса `Scene`. Пример указания стиля:

```
.root {  
    -fx-background-color: green;  
}
```

Можно также указать стиль с помощью метода `setStyle()`. Формат метода:

```
public final void setStyle(String value)
```

Пример изменения цвета фона контейнера:

```
VBox root = new VBox(15);  
root.setStyle("-fx-background-color: green;");
```

Управлять фоном позволяют следующие методы из класса `Region`:

❑ `setBackground()` — задает характеристики фона. Формат метода:

```
public final void setBackground(Background value)
```

❑ `getBackground()` — возвращает характеристики фона. Формат метода:

```
public final Background getBackground()
```

❑ `backgroundProperty()` — возвращает ссылку на свойство, связанное с фоном. Формат метода:

```
public final ObjectProperty<Background> backgroundProperty()
```

3.5.1. Класс *Background*

Характеристики фона задаются с помощью класса `Background`. Инструкция импорта:

```
// Модуль javafx.graphics  
import javafx.scene.layout.Background;
```

Создать объект позволяют следующие конструкторы:

```
Background(BackgroundFill... fills)  
Background(BackgroundImage... images)  
Background(BackgroundFill[] fills, BackgroundImage[] images)  
Background(List<BackgroundFill> fills, List<BackgroundImage> images)
```

Первый конструктор задает сплошной цвет заливки (или градиент) с помощью класса `BackgroundFill`. Можно указать несколько цветов, разделяя объекты класса

`BackgroundFill` запятыми. При этом последующие цвета будут накладываться на предыдущие. Пример указания красного цвета:

```
newWindow(new Background(new BackgroundFill(Color.RED, null, null)));
```

Пример указания трех цветов:

```
newWindow(new Background(
    new BackgroundFill(Color.RED, null, null),
    new BackgroundFill(Color.YELLOW, null, new Insets(20)),
    new BackgroundFill(Color.GREEN, new CornerRadii(10),
        new Insets(40))
));
```

Второй конструктор позволяет указать фоновое изображение с помощью класса `BackgroundImage`. Можно указать несколько изображений, разделяя объекты класса `BackgroundImage` запятыми. При этом последующие изображения будут накладываться на предыдущие. Пример использования изображения в качестве фона (если размер фона больше размера изображения, то изображение будет повторяться по горизонтали и вертикали, заполняя фон целиком):

```
try {
    Image im = new Image(
        getClass().getResourceAsStream("/img/bg_image.png"));
    newWindow(new Background(
        new BackgroundImage(im, BackgroundRepeat.REPEAT,
            BackgroundRepeat.REPEAT, BackgroundPosition.DEFAULT,
            BackgroundSize.DEFAULT)
    ));
} catch (Exception e) {
    System.out.println("Не удалось загрузить изображение");
}
```

Третий и четвертый конструкторы позволяют указать сразу и цвет заливки, и фоновое изображение в виде массива или списка. При этом вначале выводятся сплошные цвета, а затем поверх цветов фоновые изображения. Пример использования списков:

```
try {
    Image im = new Image(
        getClass().getResourceAsStream("/img/bg_image.png"));
    List<BackgroundFill> listFill = new ArrayList<BackgroundFill>();
    listFill.add(new BackgroundFill(Color.RED, null, null));
    listFill.add(new BackgroundFill(Color.GREEN, new CornerRadii(10),
        new Insets(40)));
    List<BackgroundImage> listImage = new ArrayList<BackgroundImage>();
    listImage.add(new BackgroundImage(im, BackgroundRepeat.NO_REPEAT,
        BackgroundRepeat.NO_REPEAT, BackgroundPosition.CENTER,
        BackgroundSize.DEFAULT));
    newWindow(new Background(listFill, listImage));
}
```

```
} catch (Exception e) {  
    System.out.println("Не удалось загрузить изображение");  
}
```

Если необходимо сделать цвет фона прозрачным, то можно воспользоваться статической константой `EMPTY`:

```
public static final Background EMPTY
```

Пример указания прозрачного фона для контейнера верхнего уровня:

```
root.setBackground(Background.EMPTY);
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH03_05_Background`, приведенном в каталоге `CH03_containers` сопровождающего книгу электронного архива (см. приложение).

3.5.2. Класс *BackgroundFill*: заливка сплошным цветом, градиентом или текстурой

Заливка фона сплошным цветом, градиентом или текстурой реализуется с помощью класса `BackgroundFill`. Инструкция импорта:

```
// Модуль javafx.graphics  
import javafx.scene.layout.BackgroundFill;
```

Создать объект позволяет следующий конструктор:

```
BackgroundFill(Paint fill, CornerRadii radii, Insets insets)
```

Параметр `fill` задает характеристики заливки фона. В качестве значения можно указать объекты классов `Color` (сплошной цвет), `ImagePattern` (заливка текстурой), `LinearGradient` (линейный градиент) и `RadialGradient` (радиальный градиент). Если параметр имеет значение `null`, то фон будет прозрачным. Пример заливки красным цветом:

```
newWindow(new Background(  
    new BackgroundFill(Color.RED, null, null)));
```

Параметр `radii` задает радиус скругления углов в виде объекта класса `CornerRadii` (см. *разд. 3.5.3*). Если параметр имеет значение `null`, то скругление не производится (используется значение `CornerRadii.EMPTY`).

Параметр `insets` задает отступ от краев в виде объекта класса `Insets` (см. *разд. 3.3.1*). Если параметр имеет значение `null`, то отступы равны нулю (используется значение `Insets.EMPTY`). Пример:

```
newWindow(new Background(  
    new BackgroundFill(Color.RED, CornerRadii.EMPTY, Insets.EMPTY),  
    new BackgroundFill(Color.YELLOW, null, new Insets(20)),  
    new BackgroundFill(Color.GREEN, new CornerRadii(10),  
        new Insets(40))  
));
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH03_05_BackgroundFill`, приведенном в каталоге `CH03_containers` сопровождающего книгу электронного архива (см. приложение).

3.5.3. Класс *CornerRadii*: радиус скругления углов

Радиус скругления углов задается с помощью класса `CornerRadii`. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.layout.Cornerradii;
```

Создать объект позволяют следующие конструкторы:

```
CornerRadii(double radius)
CornerRadii(double radius, boolean asPercent)
CornerRadii(double topLeft, double topRight,
             double bottomRight, double bottomLeft, boolean asPercent)
CornerRadii(
double topLeftHorizontalRadius, double topLeftVerticalRadius,
double topRightVerticalRadius, double topRightHorizontalRadius,
double bottomRightHRadius, double bottomRightVRadius,
double bottomLeftVRadius, double bottomLeftHRadius,
boolean topLeftHRadiusAsPercent, boolean topLeftVRadiusAsPercent,
boolean topRightVRadiusAsPercent, boolean topRightHRadiusAsPercent,
boolean bottomRightHRadiusAsPercent, boolean bottomRightVRadiusAsPercent,
boolean bottomLeftVRadiusAsPercent, boolean bottomLeftHRadiusAsPercent)
```

Первый конструктор задает одно значение для всех углов. Второй конструктор дополнительно принимает параметр `asPercent`. Если в параметре `asPercent` указано значение `true`, то значение интерпретируется в процентах. В первом конструкторе параметр `asPercent` имеет значение `false`. Третий конструктор задает радиус для каждого угла по отдельности, а четвертый — все характеристики по отдельности.

Пример:

```
newWindow(new CornerRadii(10.0, 30.0, 10.0, 30.0, false));
```

Если необходимо использовать значение по умолчанию, то можно воспользоваться статической константой `EMPTY`:

```
public static final CornerRadii EMPTY
```

Пример:

```
new BackgroundFill(Color.ANTIQUEWHITE,
                  CornerRadii.EMPTY, Insets.EMPTY)
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH03_05_CornerRadii`, приведенном в каталоге `CH03_containers` сопровождающего книгу электронного архива (см. приложение).

3.5.4. Класс *BackgroundImage*: фоновое изображение

Задать характеристики фонового изображения позволяет класс `BackgroundImage`. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.layout.BackgroundImage;
```

Формат конструктора класса:

```
BackgroundImage(
    Image image, BackgroundRepeat repeatX, BackgroundRepeat repeatY,
    BackgroundPosition position, BackgroundSize size)
```

Параметр `image` задает фоновое изображение, а параметры `repeatX` и `repeatY` — повторяемость фонового изображения по горизонтали и вертикали. Можно указать константы `NO_REPEAT` (без повторения), `REPEAT` (с повторением), `ROUND` и `SPACE` из перечисления `BackgroundRepeat`. Если есть бесшовная текстура, то, указав константу `REPEAT`, можно заполнить всю область фона с помощью следующего кода:

```
newWindow(new BackgroundImage(im, BackgroundRepeat.REPEAT,
    BackgroundRepeat.REPEAT, BackgroundPosition.DEFAULT,
    BackgroundSize.DEFAULT));
```

Аналогичную операцию можно выполнить и с помощью стилей:

```
root.setStyle(
    "-fx-background-image: url(/img/bg_image.jpg); " +
    "-fx-background-repeat: repeat;"
);
```

Параметр `position` задает положение фонового рисунка с помощью класса `BackgroundPosition`. Формат конструктора:

```
import javafx.scene.layout.BackgroundPosition;
import javafx.geometry.Side;
BackgroundPosition(Side horizontalSide, double horizontalPosition,
    boolean horizontalAsPercentage,
    Side verticalSide, double verticalPosition,
    boolean verticalAsPercentage)
```

Параметр `horizontalSide` может принимать константы `LEFT` или `RIGHT` из перечисления `Side`, а параметр `verticalSide` — `TOP` или `BOTTOM`. Размер отступа от указанной границы задается в параметрах `horizontalPosition` и `verticalPosition`. Если в параметрах `horizontalAsPercentage` и `verticalAsPercentage` задано значение `true`, то значение соответствующего параметра задается в процентах. Пример:

```
newWindow(new BackgroundImage(im, BackgroundRepeat.NO_REPEAT,
    BackgroundRepeat.NO_REPEAT,
    new BackgroundPosition(Side.LEFT, 20.0, false,
        Side.BOTTOM, 10.0, false),
    BackgroundSize.DEFAULT));
```


Вместо указания конкретного значения можно воспользоваться следующими статическими константами из класса `BackgroundPosition`:

❑ `DEFAULT` — позиция по умолчанию. Формат:

```
public static final BackgroundPosition DEFAULT
```

❑ `CENTER` — позиция по центру. Формат:

```
public static final BackgroundPosition CENTER
```

Пример вывода фонового изображения по центру области:

```
newWindow(new BackgroundImage(im, BackgroundRepeat.NO_REPEAT,  
    BackgroundRepeat.NO_REPEAT, BackgroundPosition.CENTER,  
    BackgroundSize.DEFAULT));
```

Параметр `size` указывает, как должны изменяться размеры фонового изображения при изменении размеров области. В качестве значения задается экземпляр класса `BackgroundSize`. Формат конструктора:

```
import javafx.scene.layout.BackgroundSize;  
BackgroundSize(double width, double height,  
    boolean widthAsPercentage, boolean heightAsPercentage,  
    boolean contain, boolean cover)
```

Параметры `width` и `height` задают ширину и высоту соответственно. Вместо конкретного значения можно указать статическую константу `BackgroundSize.AUTO`. Если в параметрах `widthAsPercentage` и `heightAsPercentage` указано значение `true`, то значение соответствующего размера задается в процентах. Если в параметре `contain` указано значение `true`, то при масштабировании сохраняется процентное соотношение сторон, чтобы занять всю ширину или высоту (что больше) области. Если в параметре `cover` указано значение `true`, то при масштабировании сохраняется процентное соотношение сторон, чтобы занять всю область и по ширине, и по высоте. Пример:

```
newWindow(new BackgroundImage(im, BackgroundRepeat.NO_REPEAT,  
    BackgroundRepeat.NO_REPEAT, BackgroundPosition.DEFAULT,  
    new BackgroundSize(BackgroundSize.AUTO, BackgroundSize.AUTO,  
        false, false, true, false)));
```

Вместо создания экземпляра класса `BackgroundSize` можно воспользоваться статической константой `DEFAULT`. Формат:

```
public static final BackgroundSize DEFAULT
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH03_05_BackgroundImage`, приведенном в каталоге `CH03_containers` сопровождающего книгу электронного архива (см. *приложение*).

3.6. Рамка

Помимо фона узел может иметь одну или несколько рамок. Управлять рамками позволяют следующие методы из класса `Region`:

❑ `setBorder()` — задает характеристики рамки. Формат метода:

```
public final void setBorder(Border value)
```

❑ `getBorder()` — возвращает характеристики рамки. Формат метода:

```
public final Border getBorder()
```

❑ `borderProperty()` — возвращает ссылку на свойство. Формат метода:

```
public final ObjectProperty<Border> borderProperty()
```

Характеристики рамки задаются с помощью класса `Border`. Инструкция импорта:

```
import javafx.scene.layout.Border;
```

Создать объект позволяют следующие конструкторы:

```
import javafx.scene.layout.BorderStroke;
import javafx.scene.layout.BorderImage;
Border(BorderStroke... strokes)
Border(BorderImage... images)
Border(BorderStroke[] strokes, BorderImage[] images)
Border(List<BorderStroke> strokes, List<BorderImage> images)
```

Первый конструктор задает сплошной цвет рамки (или градиент) с помощью класса `BorderStroke`. Можно указать несколько объектов, разделяя объекты класса `BorderStroke` запятыми. При этом последующие рамки будут накладываться на предыдущие. Пример указания трех рамок разного стиля и цвета:

```
newWindow(new Border(
    new BorderStroke(Color.RED, BorderStrokeStyle.SOLID,
        CornerRadii.EMPTY, new BorderWidths(3.0), new Insets(20.0)),
    new BorderStroke(Color.BLUE, BorderStrokeStyle.DASHED,
        CornerRadii.EMPTY, new BorderWidths(3.0), new Insets(40.0)),
    new BorderStroke(Color.GREEN, BorderStrokeStyle.DOTTED,
        CornerRadii.EMPTY, new BorderWidths(3.0), new Insets(60.0))
));
```

Второй конструктор позволяет использовать изображение рамки с помощью класса `BorderImage`. Можно указать несколько изображений, разделяя объекты класса `BorderImage` запятыми. Пример использования изображения в качестве рамки:

```
newWindow(new Border(
    new BorderImage(im, new BorderWidths(25.0),
        new Insets(20.0), BorderWidths.DEFAULT,
        false, BorderRepeat.REPEAT, BorderRepeat.REPEAT)
));
```

Третий и четвертый конструкторы позволяют указать сразу и цвет рамки, и изображение в виде массива или списка.

Конструкторы класса `BorderStroke`:

```
BorderStroke(Paint stroke, BorderStrokeStyle style,
             CornerRadii radii, BorderWidths widths)
BorderStroke(Paint stroke, BorderStrokeStyle style,
             CornerRadii radii, BorderWidths widths, Insets insets)
BorderStroke(Paint topStroke, Paint rightStroke, Paint bottomStroke,
             Paint leftStroke, BorderStrokeStyle topStyle,
             BorderStrokeStyle rightStyle, BorderStrokeStyle bottomStyle,
             BorderStrokeStyle leftStyle, CornerRadii radii,
             BorderWidths widths, Insets insets)
```

Параметр `stroke` задает цвет рамки. В качестве значения можно указать объекты классов `Color` (сплошной цвет), `ImagePattern` (фоновое изображение), `LinearGradient` (линейный градиент) и `RadialGradient` (радиальный градиент). Если параметр имеет значение `null`, то рамка будет черного цвета.

Параметр `style` задает стиль линии рамки. Можно создать объект класса `BorderStrokeStyle` с помощью конструктора или воспользоваться следующими статическими константами из класса `BorderStrokeStyle`:

```
public static final BorderStrokeStyle NONE
public static final BorderStrokeStyle SOLID
public static final BorderStrokeStyle DASHED
public static final BorderStrokeStyle DOTTED
```

Параметр `radii` задает радиус скругления углов в виде объекта класса `CornerRadii` (см. *разд. 3.5.3*). Если параметр имеет значение `null`, то скругление не производится (используется значение `CornerRadii.EMPTY`).

Параметр `widths` позволяет указать толщину линии рамки в виде объекта класса `BorderWidths`. Основные конструкторы:

```
BorderWidths(double width)
BorderWidths(double top, double right, double bottom, double left)
```

Первый конструктор задает одно значение для всех границ рамки, а второй конструктор позволяет задать ширину для каждой границы по отдельности.

Параметр `insets` задает отступ от краев в виде объекта класса `Insets` (см. *разд. 3.3.1*). Если параметр имеет значение `null`, то отступы равны нулю (используется значение `Insets.EMPTY`).

Третий конструктор класса `BorderStroke` позволяет указать характеристики каждой границы рамки по отдельности. Пример:

```
BorderStroke bs = new BorderStroke(
    Color.RED, Color.BLUE, Color.GREEN, Color.BLACK,
    BorderStrokeStyle.SOLID, BorderStrokeStyle.DASHED,
    BorderStrokeStyle.DOTTED, BorderStrokeStyle.DOTTED,
    CornerRadii.EMPTY, new BorderWidths(3.0), new Insets(20.0));
```

Если необходимо убрать рамку, то можно воспользоваться статической константой `EMPTY` из класса `Border`. Формат:

```
public static final Border EMPTY
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH03_06_Border`, приведенном в каталоге `CH03_containers` сопровождающего книгу электронного архива (см. приложение).

3.7. Изменение прозрачности узла

Управлять прозрачностью узла позволяют следующие методы из класса `Node`:

- ❑ `setOpacity()` — задает степень непрозрачности: вещественное число от 0.0 до 1.0. Формат метода:

```
public final void setOpacity(double value)
```

- ❑ `getOpacity()` — возвращает степень непрозрачности. Формат метода:

```
public final double getOpacity()
```

- ❑ `opacityProperty()` — возвращает ссылку на свойство. Формат метода:

```
public final DoubleProperty opacityProperty()
```

При использовании программы `Scene Builder` степень непрозрачности задается на вкладке **Properties** в разделе **Node** (параметр **Opacity**). Пример указания степени непрозрачности в FXML-файле:

```
<Button mnemonicParsing="false" opacity="0.7" text="Button" />
```

В качестве примера изменим степень непрозрачности надписи при изменении положения ползунка (листинг 3.1).

Листинг 3.1. Изменение прозрачности

```
package application;

import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.Slider;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class Main extends Application {

    public static void main(String[] args) {
        Application.launch(args);
    }
}
```

```

@Override
public void start(Stage stage) throws Exception {
    VBox root = new VBox(30);
    root.setAlignment(Pos.CENTER);
    Label label = new Label("Измените значение с помощью ползунка");
    label.setStyle("-fx-font-size: 12pt;");
    Slider slider = new Slider(0.0, 1.0, 0.7);
    slider.setShowTickLabels(true);
    slider.setMajorTickUnit(0.1);
    root.getChildren().addAll(label, slider);
    label.opacityProperty().bindBidirectional(slider.valueProperty());
    Scene scene = new Scene(root, 400, 150);
    stage.setTitle("Изменение прозрачности");
    stage.setScene(scene);
    stage.show();
}
}

```

Обратите внимание на инструкцию:

```
label.opacityProperty().bindBidirectional(slider.valueProperty());
```

Эта инструкция демонстрирует возможность связывания значений свойств различных узлов. Вместо того чтобы назначать обработчик изменения значения ползунка и внутри обработчика изменять степень непрозрачности надписи, мы можем все автоматизировать с помощью одной инструкции. При изменении значения ползунка автоматически будет меняться степень непрозрачности надписи. Более подробно свойства и обработку событий мы рассмотрим далее в этой книге.

3.8. Управление видимостью узла

Управлять видимостью узла позволяют следующие методы из класса `Node`:

❑ `setVisible()` — позволяет изменить значение. Формат метода:

```
public final void setVisible(boolean value)
```

❑ `isVisible()` — возвращает значение `true`, если узел видим, и `false` — в противном случае. Формат метода:

```
public final boolean isVisible()
```

❑ `visibleProperty()` — возвращает ссылку на свойство. Формат метода:

```
public final BooleanProperty visibleProperty()
```

Не путайте понятия видимости узла и прозрачности узла. Узел может быть полностью прозрачным, но при этом видимым.

При использовании программы **Scene Builder** значение видимости узла задается на вкладке **Properties** в разделе **Node** (флажок **Visible**). Пример отключения видимости надписи в FXML-файле:

```
<Label text="Label" visible="false" />
```

ЭЛЕКТРОННЫЙ АРХИВ

Пример управления видимостью узла содержится в проекте `FX_CH03_08_visible`, приведенном в каталоге `CH03_containers` сопровождающего книгу электронного архива (см. приложение).

3.9. Управление доступностью узла

В некоторых случаях необходимо сделать некоторые компоненты или все узлы внутри контейнера недоступными. Например, при нажатии кнопки осуществляется длительная операция. Чтобы пользователь много раз не нажимал кнопку, думая, что программа зависла, следует сразу после нажатия кнопки сделать ее недоступной, а после завершения выполнения операции снова вернуть ей доступность. Если узел недоступен, то он не принимает событий, а следовательно, при попытке нажать недоступную кнопку обработчик вызван не будет.

Управлять доступностью узла позволяют следующие методы из класса `Node`:

- ❑ `setDisable()` — позволяет изменить значение. Если указать значение `true`, то узел станет недоступным. Формат метода:

```
public final void setDisable(boolean value)
```

- ❑ `isDisable()` — возвращает значение `true`, если для узла установлен статус недоступности, и `false` — в противном случае. Если сделать недоступным контейнер, то все узлы внутри контейнера станут недоступными, однако статус недоступности для конкретного узла внутри контейнера изменен не будет. Чтобы проверить текущий статус, следует использовать метод `isDisabled()`. Формат метода:

```
public final boolean isDisable()
```

- ❑ `isDisabled()` — возвращает значение `true`, если узел недоступен, и `false` — в противном случае. Формат метода:

```
public final boolean isDisabled()
```

- ❑ `disableProperty()` — возвращает ссылку на свойство `disable`. Формат метода:

```
public final BooleanProperty disableProperty()
```

- ❑ `disabledProperty()` — возвращает ссылку на свойство `disabled`. Формат метода:

```
public final ReadOnlyBooleanProperty disabledProperty()
```

При использовании программы `Scene Builder` значение доступности узла задается на вкладке **Properties** в разделе **Node** (флажок **Disable**). Пример отключения доступности кнопки в FXML-файле:

```
<Button disable="true" mnemonicParsing="false" text="Button" />
```

ЭЛЕКТРОННЫЙ АРХИВ

Пример управления доступностью узла содержится в проекте `FX_CH03_09_disable`, приведенном в каталоге `CH03_containers` сопровождающего книгу электронного архива (см. приложение).

3.10. Абсолютное позиционирование

Прежде чем изучать компоненты-контейнеры, рассмотрим возможность *абсолютного позиционирования* компонентов в окне. Итак, если мы используем классы `Group` или `Pane`, то при добавлении компонента он выводится в позицию с координатами $(0, 0)$. Иными словами, если мы добавим несколько компонентов, то все они отобразятся на одной и той же позиции. Последний добавленный компонент окажется на вершине этой кучи, а остальные компоненты станут видны лишь частично или вообще не видны. Размеры добавляемых компонентов будут соответствовать их содержимому. Для изменения местоположения компонента следует воспользоваться методами из *разд. 3.4*, а для изменения размеров — методами из *разд. 3.2*.

Абсолютное позиционирование имеет следующие недостатки:

- ❑ при изменении размеров окна необходимо пересчитывать и изменять характеристики всех компонентов вручную;
- ❑ при указании фиксированных размеров компонентов возможны проблемы. Помните, что в приложении могут использоваться разные стили оформления, в том числе и характеристики шрифта. Подогнав размеры под один стиль, можно прийти в ужас при виде приложения, в котором использован другой стиль, где размер шрифта в два раза больше. Поэтому лучше вообще отказаться от указания фиксированных размеров или задавать размер и название шрифта для каждого компонента. Кроме того, приложение может поддерживать несколько языков интерфейса. Длина слов в разных языках различается, что также станет причиной искажения компонентов.

3.10.1. Класс *Group*

Класс `Group` позволяет группировать узлы. Любые преобразования или эффекты, примененные к группе, используются ко всем узлам внутри группы. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.Group;
```

Иерархия наследования:

`Object` — `Node` — `Parent` — `Group`

Обратите внимание: класс `Group` не наследует класс `Region`, а, следовательно, группа не имеет фона (фон прозрачный). Кроме того, нельзя указать размеры группы с помощью методов из *разд. 3.2*.

Создать экземпляры класса `Group` позволяют следующие конструкторы:

```
Group()
Group(Node... children)
Group(Collection<Node> children)
```

Добавить узлы в группу можно с помощью конструкторов класса или метода `getChildren()`. Метод `getChildren()` возвращает список узлов группы в виде объекта класса `ObservableList<E>`. Формат метода:

```
public ObservableList<Node> getChildren()
```

Класс `ObservableList<E>` реализует интерфейсы `Collection<E>` и `List<E>`, поэтому для добавления или удаления узлов можно воспользоваться методами из этих интерфейсов. Например, добавить один узел позволяет метод `add()`:

```
Group root = new Group();  
Button button = new Button("Кнопка");  
root.getChildren().add(button);
```

Положение узла внутри списка может иметь значение, поэтому не следует забывать, что метод `add()` в первом параметре позволяет указать индекс позиции:

```
public void add(int index, E element)
```

Для добавления сразу нескольких узлов используется метод `addAll()`. Форматы метода:

```
public boolean addAll(E... elements)  
public boolean addAll(Collection<? extends E> c)  
public boolean addAll(int index, Collection<? extends E> c)
```

Пример добавления нескольких узлов в конец списка:

```
Group root = new Group();  
Button button1 = new Button("Кнопка 1");  
Button button2 = new Button("Кнопка 2");  
root.getChildren().addAll(button1, button2);  
// Изменяем местоположение  
button1.relocate(100.0, 50.0);  
button2.relocate(100.0, 100.0);
```

При использовании программы *Scene Builder* компонент `Group` можно найти на вкладке **Miscellaneous**. Пример описания группы в `FXML`-файле:

```
<Group>  
  <children>  
    <Button mnemonicParsing="false" text="Button" />  
    <Label layoutX="100.0" layoutY="50.0" text="Label" />  
  </children>  
</Group>
```

ЭЛЕКТРОННЫЙ АРХИВ

Пример использования класса `Group` содержится в проекте `FX_CH03_10_Group`, приведенном в каталоге `CH03_containers` сопровождающего книгу электронного архива (см. приложение).

3.10.2. Класс *Pane*

Класс `Pane` является базовым классом для большинства компонентов-контейнеров. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.layout.Pane;
```

Иерархия наследования:

`Object` — `Node` — `Parent` — `Region` — `Pane`

Создать экземпляр класса `Pane` позволяют следующие конструкторы:

```
Pane()
Pane(Node... children)
```

Добавить узлы можно с помощью конструктора класса или метода `getChildren()`. Метод `getChildren()` возвращает список узлов в виде объекта класса `ObservableList<E>`. Формат метода:

```
public ObservableList<Node> getChildren()
```

Класс `ObservableList<E>` реализует интерфейсы `Collection<E>` и `List<E>`, поэтому для добавления или удаления узлов можно воспользоваться методами из этих интерфейсов. Например, добавить один узел позволяет метод `add()`:

```
Pane root = new Pane();
Button button = new Button("Кнопка");
root.getChildren().add(button);
```

Положение узла внутри списка может иметь значение, поэтому не следует забывать, что метод `add()` в первом параметре позволяет указать индекс позиции:

```
public void add(int index, E element)
```

Для добавления сразу нескольких узлов используется метод `addAll()`. Форматы метода:

```
public boolean addAll(E... elements)
public boolean addAll(Collection<? extends E> c)
public boolean addAll(int index, Collection<? extends E> c)
```

Пример добавления нескольких узлов в конец списка:

```
Pane root = new Pane();
Button button1 = new Button("Кнопка 1");
Button button2 = new Button("Кнопка 2");
root.getChildren().addAll(button1, button2);
// Изменяем местоположение
button1.relocate(100.0, 50.0);
button2.relocate(100.0, 100.0);
// Задаем предпочтительные размеры панели
root.setPrefSize(400.0, 300.0);
```

При использовании программы Scene Builder компонент `Pane` можно найти на вкладке **Containers**. Пример описания панели в FXML-файле:

```
<Pane prefHeight="200.0" prefWidth="200.0">
  <children>
    <Button mnemonicParsing="false" text="Button" />
    <Label layoutX="120.0" layoutY="50.0" text="Label" />
  </children>
</Pane>
```

ЭЛЕКТРОННЫЙ АРХИВ

Пример использования класса `Pane` содержится в проекте `FX_CH03_10_Pane`, приведенном в каталоге `CH03_containers` сопровождающего книгу электронного архива (см. *приложение*).

3.11. Горизонтальное и вертикальное выравнивание

Компоненты-контейнеры (их еще называют *менеджерами компоновки* или *менеджерами геометрии*) лишены недостатков абсолютного позиционирования. При изменении размеров окна производится автоматическое изменение характеристик всех компонентов, добавленных в контейнер. Настройки шрифта при этом также учитываются, поэтому изменение размеров шрифта в два раза приведет только к увеличению компонентов и размеров окна. Для автоматического выравнивания компонентов по горизонтали и вертикали задействуются классы `HBox` и `VBox`.

3.11.1. Класс *HBox*: выравнивание по горизонтали

Класс `HBox` выстраивает все добавляемые компоненты по горизонтали (по умолчанию слева направо). Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.layout.HBox;
```

Иерархия наследования:

`Object` — `Node` — `Parent` — `Region` — `Pane` — `HBox`

Создать экземпляры класса `HBox` позволяют следующие конструкторы:

```
HBox()
HBox(double spacing)
HBox(Node... children)
HBox(double spacing, Node... children)
```

Параметр `spacing` задает расстояние между узлами по горизонтали. Если параметр не указан, то расстояние равно нулю. Управлять расстоянием позволяют также следующие методы:

□ `setSpacing()` — задает расстояние между узлами. Формат метода:

```
public final void setSpacing(double value)
```

❑ `getSpacing()` — возвращает текущее расстояние. Формат метода:

```
public final double getSpacing()
```

❑ `spacingProperty()` — возвращает ссылку на свойство. Формат метода:

```
public final DoubleProperty spacingProperty()
```

При использовании программы Scene Builder расстояние между узлами задается на вкладке **Layout** в разделе **Internal** (поле **Spacing**). Пример указания значения в FXML-файле:

```
<HBox prefHeight="400.0" prefWidth="600.0" spacing="15.0">
  <children>
    <Button mnemonicParsing="false" text="Button" />
    <Button mnemonicParsing="false" text="Button" />
  </children>
</HBox>
```

Добавить узлы можно с помощью конструкторов класса или метода `getChildren()`. Пример:

```
HBox pane = new HBox(15.0);
Button button1 = new Button("Кнопка 1");
Button button2 = new Button("Кнопка 2");
pane.getChildren().addAll(button1, button2);
```

С помощью свойства `fillHeight` можно указать, будут ли узлы заполнять всю доступную область по вертикали. Изменить или получить значение позволяют следующие методы:

❑ `setFillHeight()` — задает значение. Формат метода:

```
public final void setFillHeight(boolean value)
```

❑ `isFillHeight()` — возвращает текущее значение. Формат метода:

```
public final boolean isFillHeight()
```

❑ `fillHeightProperty()` — возвращает ссылку на свойство. Формат метода:

```
public final BooleanProperty fillHeightProperty()
```

При использовании программы Scene Builder значение задается на вкладке **Layout** в разделе **Specific** (флажок **Fill Height**). Пример указания значения в FXML-файле:

```
<HBox prefHeight="400.0" prefWidth="600.0" fillHeight="false">
  <children>
    <Button mnemonicParsing="false" text="Button" />
    <Button mnemonicParsing="false" text="Button" />
  </children>
</HBox>
```

Изменить выравнивание узлов внутри контейнера позволяют следующие методы:

❑ `setAlignment()` — задает выравнивание. В качестве параметра указываются следующие константы из перечисления `javafx.geometry.Pos`: `BASELINE_CENTER`,

BASELINE_LEFT, BASELINE_RIGHT, BOTTOM_CENTER, BOTTOM_LEFT, BOTTOM_RIGHT, CENTER, CENTER_LEFT, CENTER_RIGHT, TOP_CENTER, TOP_LEFT и TOP_RIGHT. **Формат метода:**

```
public final void setAlignment(Pos value)
```

❑ `getAlignment()` — возвращает текущее значение. **Формат метода:**

```
public final Pos getAlignment()
```

❑ `alignmentProperty()` — возвращает ссылку на свойство. **Формат метода:**

```
public final ObjectProperty<Pos> alignmentProperty()
```

При использовании программы Scene Builder значение задается на вкладке **Properties** в разделе **Node** (список **Alignment**). Пример указания значения в FXML-файле:

```
<HBox prefHeight="400.0" prefWidth="600.0" alignment="CENTER">
  <children>
    <Button mnemonicParsing="false" text="Button" />
    <Button mnemonicParsing="false" text="Button" />
  </children>
</HBox>
```

По умолчанию размеры окна можно изменять с помощью мыши. Если необходимо, чтобы какой-либо компонент занимал всю доступную область и растягивался при увеличении ширины окна, то для этого компонента следует назначить приоритет с помощью статического метода `setHgrow()`. **Формат метода:**

```
public static void setHgrow(Node child, Priority value)
```

Причем в большинстве случаев нужно дополнительно указать значение `Double.MAX_VALUE` для максимальной ширины узла:

```
button.setMaxWidth(Double.MAX_VALUE);
```

В параметре `value` указываются следующие константы из перечисления `javafx.scene.layout.Priority`:

- ❑ **NEVER** — размеры узла не изменяются;
- ❑ **ALWAYS** — размеры узла всегда изменяются. Если существуют несколько компонентов с приоритетом **ALWAYS**, то они будут делить пространство поровну;
- ❑ **SOMETIMES** — размеры узла изменяются, только если нет других узлов с приоритетом **ALWAYS**.

Получить текущий приоритет узла позволяет статический метод `getHgrow()`. **Формат метода:**

```
public static Priority getHgrow(Node child)
```

При использовании программы Scene Builder приоритет задается на вкладке **Layout** в разделе **HBox Constraints** (список **Hgrow**). Пример указания значения в FXML-файле:

```
<Button maxWidth="1.7976931348623157E308" mnemonicParsing="false"
    text="Button" HBox.hgrow="ALWAYS" />
```

При использовании класса `HBox` мы можем указать расстояние между узлами (метод `setSpacing()`), внутренний отступ от границ контейнера до узлов (метод `setPadding()`), а также внешний отступ для каждого узла по отдельности с помощью статического метода `setMargin()`. Формат метода:

```
public static void setMargin(Node child, Insets value)
```

Размер отступа задается с помощью класса `Insets` (см. *разд. 3.3.1*). Пример:

```
HBox.setMargin(button1, new Insets(20.0));
```

Получить размер внешнего отступа позволяет статический метод `getMargin()`. Формат метода:

```
public static Insets getMargin(Node child)
```

При использовании программы `Scene Builder` внешний отступ задается на вкладке **Layout** в разделе **HBox Constraints** (параметр **Margin**). Если в первое поле ввести значение и нажать на значок в форме стрелки, указывающей вправо, то остальные поля будут автоматически заполнены тем же самым значением. Пример указания значения в `FXML`-файле:

```
<Button mnemonicParsing="false" text="Button">
    <HBox.margin>
        <Insets bottom="20.0" left="20.0" right="20.0" top="20.0" />
    </HBox.margin>
</Button>
```

ЭЛЕКТРОННЫЙ АРХИВ

Пример использования класса `HBox` содержится в проекте `FX_CH03_11_HBox`, приведенном в каталоге `CH03_containers` сопровождающего книгу электронного архива (см. *приложение*).

3.11.2. Класс `VBox`: выравнивание по вертикали

Класс `VBox` выстраивает все добавляемые компоненты по вертикали (сверху вниз). Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.layout.VBox;
```

Иерархия наследования:

```
Object — Node — Parent — Region — Pane — VBox
```

Создать экземпляр класса `VBox` позволяют следующие конструкторы:

```
VBox()
VBox(double spacing)
VBox(Node... children)
VBox(double spacing, Node... children)
```

Параметр `spacing` задает расстояние между узлами по вертикали. Если параметр не указан, то расстояние равно нулю. Управлять расстоянием позволяют также следующие методы:

- ❑ `setSpacing()` — задает расстояние между узлами. Формат метода:
`public final void setSpacing(double value)`
- ❑ `getSpacing()` — возвращает текущее расстояние. Формат метода:
`public final double getSpacing()`
- ❑ `spacingProperty()` — возвращает ссылку на свойство. Формат метода:
`public final DoubleProperty spacingProperty()`

При использовании программы Scene Builder расстояние между узлами задается на вкладке **Layout** в разделе **Internal** (поле **Spacing**). Пример указания значения в FXML-файле:

```
<VBox prefHeight="400.0" prefWidth="600.0" spacing="15.0">
  <children>
    <Button mnemonicParsing="false" text="Button" />
    <Button mnemonicParsing="false" text="Button" />
  </children>
</VBox>
```

Добавить узлы можно с помощью конструкторов класса или метода `getChildren()`. Пример:

```
VBox pane = new VBox(15.0);
Button button1 = new Button("Кнопка 1");
Button button2 = new Button("Кнопка 2");
pane.getChildren().addAll(button1, button2);
```

С помощью свойства `fillWidth` можно указать, будут ли узлы заполнять всю доступную область по горизонтали. Изменить или получить значение позволяют следующие методы:

- ❑ `setFillWidth()` — задает значение. Формат метода:
`public final void setFillWidth(boolean value)`
- ❑ `isFillWidth()` — возвращает текущее значение. Формат метода:
`public final boolean isFillWidth()`
- ❑ `fillWidthProperty()` — возвращает ссылку на свойство. Формат метода:
`public final BooleanProperty fillWidthProperty()`

При использовании программы Scene Builder значение задается на вкладке **Layout** в разделе **Specific** (флажок **Fill Width**). Пример указания значения в FXML-файле:

```
<VBox prefHeight="400.0" prefWidth="600.0" fillHeight="false">
  <children>
    <Button mnemonicParsing="false" text="Button" />
    <Button mnemonicParsing="false" text="Button" />
  </children>
```

```
</children>
</VBox>
```

Изменить выравнивание узлов внутри контейнера позволяют следующие методы:

- ❑ `setAlignment()` — задает выравнивание. В качестве параметра указываются следующие константы из перечисления `javafx.geometry.Pos`: `BASELINE_CENTER`, `BASELINE_LEFT`, `BASELINE_RIGHT`, `BOTTOM_CENTER`, `BOTTOM_LEFT`, `BOTTOM_RIGHT`, `CENTER`, `CENTER_LEFT`, `CENTER_RIGHT`, `TOP_CENTER`, `TOP_LEFT` и `TOP_RIGHT`. Формат метода:

```
public final void setAlignment(Pos value)
```

- ❑ `getAlignment()` — возвращает текущее значение. Формат метода:

```
public final Pos getAlignment()
```

- ❑ `alignmentProperty()` — возвращает ссылку на свойство. Формат метода:

```
public final ObjectProperty<Pos> alignmentProperty()
```

При использовании программы **Scene Builder** значение задается на вкладке **Properties** в разделе **Node** (список **Alignment**). Пример указания значения в FXML-файле:

```
<VBox prefHeight="400.0" prefWidth="600.0" alignment="CENTER">
    <children>
        <Button mnemonicParsing="false" text="Button" />
        <Button mnemonicParsing="false" text="Button" />
    </children>
</VBox>
```

По умолчанию размеры окна можно изменять с помощью мыши. Если необходимо, чтобы какой-либо компонент занимал всю доступную область и растягивался при увеличении высоты окна, то для этого компонента следует назначить приоритет с помощью статического метода `setVgrow()`. Формат метода:

```
public static void setVgrow(Node child, Priority value)
```

Причем в большинстве случаев нужно дополнительно указать значение `Double.MAX_VALUE` для максимальной высоты узла:

```
button.setMaxHeight(Double.MAX_VALUE)
```

В параметре `value` указываются следующие константы из перечисления `javafx.scene.layout.Priority`:

- ❑ `NEVER` — размеры узла не изменяются;
- ❑ `ALWAYS` — размеры узла всегда изменяются. Если существуют несколько компонентов с приоритетом `ALWAYS`, то они будут делить пространство поровну;
- ❑ `SOMETIMES` — размеры узла изменяются, только если нет других узлов с приоритетом `ALWAYS`.

Получить текущий приоритет узла позволяет статический метод `getVgrow()`. Формат метода:

```
public static Priority getVgrow(Node child)
```

При использовании программы Scene Builder приоритет задается на вкладке **Layout** в разделе **VBox Constraints** (список **Vgrow**). Пример указания значения в FXML-файле:

```
<Button maxHeight="1.7976931348623157E308" mnemonicParsing="false"
    text="Button" VBox.vgrow="ALWAYS" />
```

Используя класс `VBox`, мы можем указать расстояние между узлами (метод `setSpacing()`), внутренний отступ от границ контейнера до узлов (метод `setPadding()`), а также внешний отступ для каждого узла по отдельности с помощью статического метода `setMargin()`. Формат метода:

```
public static void setMargin(Node child, Insets value)
```

Размер отступа задается с помощью класса `Insets` (см. *разд. 3.3.1*). Пример:

```
VBox.setMargin(button1, new Insets(20.0));
```

Получить размер внешнего отступа позволяет статический метод `getMargin()`. Формат метода:

```
public static Insets getMargin(Node child)
```

При использовании программы Scene Builder внешний отступ задается на вкладке **Layout** в разделе **VBox Constraints** (параметр **Margin**). Если в первое поле ввести значение и нажать на значок в форме стрелки, указывающей вправо, то остальные поля будут автоматически заполнены тем же самым значением. Пример указания значения в FXML-файле:

```
<Button mnemonicParsing="false" text="Button">
    <VBox.margin>
        <Insets bottom="20.0" left="20.0" right="20.0" top="20.0" />
    </VBox.margin>
</Button>
```

ЭЛЕКТРОННЫЙ АРХИВ

Пример использования класса `VBox` содержится в проекте `FX_CH03_11_VBox`, приведенном в каталоге `CH03_containers` сопровождающего книгу электронного архива (см. *приложение*).

3.12. Класс *GridPane*: размещение узлов внутри ячеек таблицы

Помимо выравнивания компонентов по горизонтали или вертикали существует возможность размещения узлов внутри ячеек таблицы. Для этого предназначен класс `GridPane`. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.layout.GridPane;
```

Иерархия наследования:

Object — Node — Parent — Region — Pane — GridPane

Создать экземпляр класса можно следующим образом:

```
GridPane pane = new GridPane();
```

При использовании программы Scene Builder компонент `GridPane` можно найти на вкладке **Containers**. После добавления компонента на форму таблица будет состоять из двух столбцов и трех строк. По сторонам таблицы расположены ярлыки. Если щелкнуть левой кнопкой мыши на ярлыке, то отобразится вкладка **Layout**, на которой задаются значения различных свойств для строки или столбца. Если на ярлыке щелкнуть правой кнопкой мыши, то отобразится контекстное меню, с помощью которого можно удалить строку (или столбец), добавить новую строку (или столбец) или переместить строку (или столбец).

Чтобы добавить компонент в таблицу, достаточно перетащить его с помощью мыши в нужную ячейку. Для изменения свойств отдельной ячейки надо выделить компонент, расположенный внутри ячейки, и отредактировать значения свойств на вкладке **Layout** в разделе **Grid Pane Constraints**.

Чтобы изменить свойства всей таблицы, нужно щелкнуть левой кнопкой мыши на свободном месте таблицы и изменить значения свойств на вкладках **Layout** и **Properties**.

На рис. 3.1 показан процесс создания формы идентификации пользователя в программе Scene Builder, а результат после запуска можно увидеть на рис. 3.2.

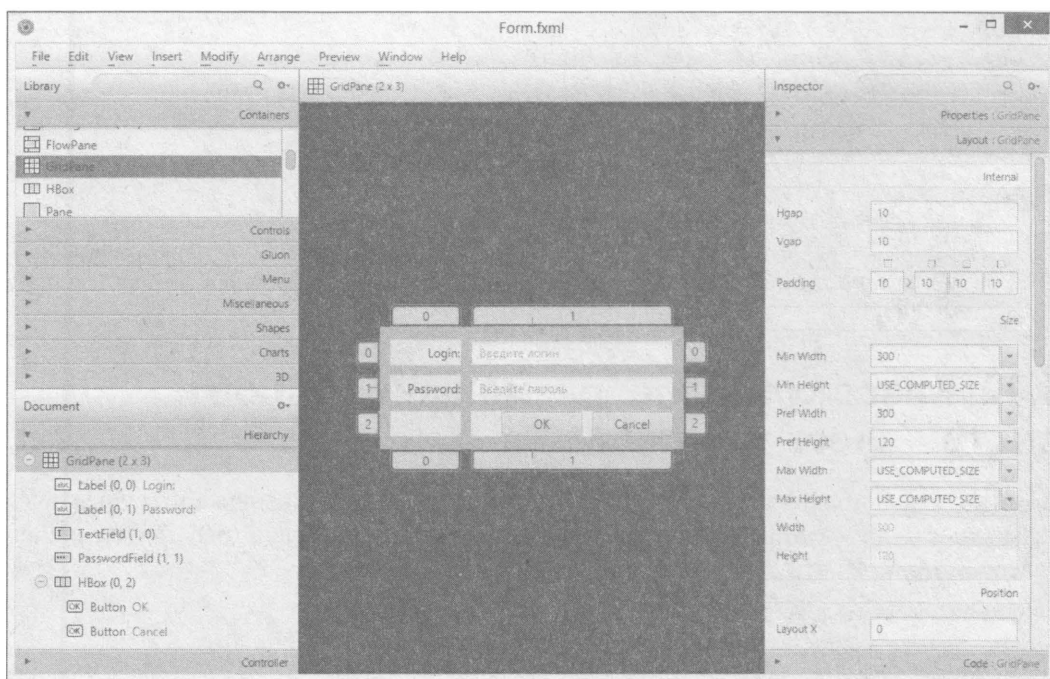


Рис. 3.1. Процесс создания формы идентификации пользователя в программе Scene Builder

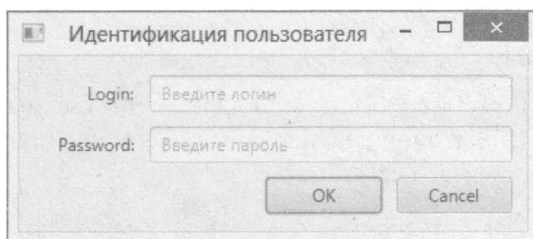


Рис. 3.2. Результат после запуска программы

ЭЛЕКТРОННЫЙ АРХИВ

Пример использования класса `GridPane` содержится в проекте `FX_CH03_12_GridPane`, приведенном в каталоге `CH03_containers` сопровождающего книгу электронного архива (см. приложение).

3.12.1. Класс *ColumnConstraints*: описание столбцов

Свойства каждого столбца таблицы описываются с помощью класса `ColumnConstraints`. Инструкция импорта:

```
import javafx.scene.layout.ColumnConstraints;
```

Иерархия наследования:

Object — ConstraintsBase — ColumnConstraints

Создать экземпляр класса `ColumnConstraints` позволяют следующие конструкторы:

```
ColumnConstraints()
ColumnConstraints(double width)
ColumnConstraints(double minWidth, double prefWidth, double maxWidth)
ColumnConstraints(double minWidth, double prefWidth, double maxWidth,
                  Priority hgrow, HPos halignment, boolean fillWidth)
```

Первый конструктор создает объект без указания каких-либо свойств. Второй конструктор позволяет указать ширину столбца. При этом параметры, определяющие минимальную ширину и максимальную ширину, будут иметь значение `Region.USE_PREF_SIZE`. Третий конструктор задает минимальную, предпочтительную и максимальную ширину столбца. Управлять шириной столбца после создания объекта позволяют следующие методы:

```
// Предпочтительный размер
public final void setPrefWidth(double value)
public final double getPrefWidth()
public final DoubleProperty prefWidthProperty()
// Минимальный размер
public final void setMinWidth(double value)
public final double getMinWidth()
public final DoubleProperty minWidthProperty()
// Максимальный размер
public final void setMaxWidth(double value)
```

```
public final double getMaxWidth()
public final DoubleProperty maxWidthProperty()
```

Ширину столбца можно также указать в процентах (по умолчанию параметр имеет значение `-1`, означающее, что процентное соотношение не применяется). Для этого используются следующие методы:

```
public final void setPercentWidth(double value)
public final double getPercentWidth()
public final DoubleProperty percentWidthProperty()
```

Если задано значение в процентах, то значения параметров `minWidth`, `prefWidth`, `maxWidth` и `hgrow` игнорируются.

Четвертый конструктор дополнительно принимает три параметра:

- ❑ `hgrow` — задает приоритет изменения ширины столбца при изменении размеров панели. В качестве значения можно указать константы: `ALWAYS`, `SOMETIMES` или `NEVER` из перечисления `Priority`. Эти константы мы уже рассматривали в *разд. 3.11.1*. Управлять приоритетом после создания объекта позволяют методы `setHgrow()`, `getHgrow()` и `hgrowProperty()`. Форматы методов:

```
public final void setHgrow(Priority value)
public final Priority getHgrow()
public final ObjectProperty<Priority> hgrowProperty()
```

- ❑ `halignment` — задает горизонтальное выравнивание узлов внутри ячеек столбца. В качестве значения можно указать константы: `LEFT`, `CENTER` или `RIGHT` из перечисления `javafx.geometry.HPos`. Управлять выравниванием после создания объекта позволяют следующие методы:

```
public final void setHalignment(HPos value)
public final HPos getHalignment()
public final ObjectProperty<HPos> halignmentProperty()
```

- ❑ `fillWidth` — если указано значение `true`, то узел будет заполнять всю ширину ячейки. Для некоторых узлов необходимо дополнительно указать значение `Double.MAX_VALUE` для максимального размера. Управлять параметром после создания объекта позволяют следующие методы:

```
public final void setFillWidth(boolean value)
public final boolean isFillWidth()
public final BooleanProperty fillWidthProperty()
```

После создания объекта нужно вызвать метод `getColumnConstraints()` из класса `GridPane` и добавить объект в список. Формат метода:

```
public final ObservableList<ColumnConstraints> getColumnConstraints()
```

Пример:

```
GridPane pane = new GridPane();
ColumnConstraints col = new ColumnConstraints(
    20, 150, Double.MAX_VALUE);
```

```
col.setHgrow(Priority.ALWAYS);  
col.setFillWidth(true);  
col.setHalignment(HPos.CENTER);  
pane.getColumnConstraints().add(col);
```

3.12.2. Класс *RowConstraints*: описание строк

Свойства каждой строки таблицы описываются с помощью класса `RowConstraints`. Инструкция импорта:

```
import javafx.scene.layout.RowConstraints;
```

Иерархия наследования:

```
Object — ConstraintsBase — RowConstraints
```

Создать экземпляр класса `RowConstraints` позволяют следующие конструкторы:

```
RowConstraints()  
RowConstraints(double height)  
RowConstraints(double minHeight, double prefHeight, double maxHeight)  
RowConstraints(double minHeight, double prefHeight, double maxHeight,  
                Priority vgrow, VPos valignment, boolean fillHeight)
```

Первый конструктор создает объект без указания каких-либо свойств. Второй конструктор позволяет указать высоту строки. При этом параметры, определяющие минимальную высоту и максимальную высоту, будут иметь значение `Region.USE_PREF_SIZE`. Третий конструктор задает минимальную, предпочтительную и максимальную высоту строки. Управлять высотой строки после создания объекта позволяют следующие методы:

```
// Предпочтительный размер  
public final void setPrefHeight(double value)  
public final double getPrefHeight()  
public final DoubleProperty prefHeightProperty()  
// Минимальный размер  
public final void setMinHeight(double value)  
public final double getMinHeight()  
public final DoubleProperty minHeightProperty()  
// Максимальный размер  
public final void setMaxHeight(double value)  
public final double getMaxHeight()  
public final DoubleProperty maxHeightProperty()
```

Высоту строки можно также указать в процентах (по умолчанию параметр имеет значение `-1`, означающее, что процентное соотношение не применяется). Для этого используются следующие методы:

```
public final void setPercentHeight(double value)  
public final double getPercentHeight()  
public final DoubleProperty percentHeightProperty()
```

Если задано значение в процентах, то значения параметров `minHeight`, `prefHeight`, `maxHeight` и `vgrow` игнорируются.

Четвертый конструктор дополнительно принимает три параметра:

- ❑ `vgrow` — задает приоритет изменения высоты строки при изменении размеров панели. В качестве значения можно указать константы: `ALWAYS`, `SOMETIMES` или `NEVER` из перечисления `Priority`. Эти константы мы уже рассматривали в разд. 3.11.1. Управлять приоритетом после создания объекта позволяют следующие методы:

```
public final void setVgrow(Priority value)
public final Priority getVgrow()
public final ObjectProperty<Priority> vgrowProperty()
```

- ❑ `valignment` — задает вертикальное выравнивание узлов внутри ячеек строки. В качестве значения можно указать константы: `TOP`, `BASELINE`, `CENTER` или `BOTTOM` из перечисления `javafx.geometry.VPos`. Управлять выравниванием после создания объекта позволяют следующие методы:

```
public final void setValignment(VPos value)
public final VPos getValignment()
public final ObjectProperty<VPos> valignmentProperty()
```

- ❑ `fillHeight` — если указано значение `true`, то узел будет заполнять всю высоту ячейки. Для некоторых узлов необходимо дополнительно указать значение `Double.MAX_VALUE` для максимального размера. Управлять параметром после создания объекта позволяют следующие методы:

```
public final void setFillHeight(boolean value)
public final boolean isFillHeight()
public final BooleanProperty fillHeightProperty()
```

После создания объекта нужно вызвать метод `getRowConstraints()` из класса `GridPane` и добавить объект в список. Формат метода:

```
public final ObservableList<RowConstraints> getRowConstraints()
```

Пример:

```
GridPane pane = new GridPane();
RowConstraints row = new RowConstraints(20, 50, Double.MAX_VALUE);
row.setFillHeight(true);
row.setVgrow(Priority.ALWAYS);
row.setValignment(VPos.CENTER);
pane.getRowConstraints().add(row);
```

Пример указания значений в FXML-файле:

```
<GridPane prefHeight="400.0" prefWidth="600.0">
  <columnConstraints>
    <ColumnConstraints fillWidth="false" halignment="LEFT"
      hgrow="SOMETIMES" maxWidth="1.7976931348623157E308"
      minWidth="10.0" prefWidth="100.0" />
  </columnConstraints>
```

```
<rowConstraints>
<RowConstraints fillHeight="false" maxHeight="1.7976931348623157E308"
  minHeight="10.0" prefHeight="100.0" valignment="TOP"
  vgrow="SOMETIMES" />
</rowConstraints>
<children>
  <Button mnemonicParsing="false" text="Button" />
</children>
</GridPane>
```

3.12.3. Добавление узла в ячейку таблицы

Добавить узлы в ячейки таблицы можно несколькими способами. Если строки и столбцы описаны с помощью классов `ColumnConstraints` и `RowConstraints`, то достаточно воспользоваться методом `add()`. Форматы метода:

```
public void add(Node child, int columnIndex, int rowIndex)
public void add(Node child, int columnIndex, int rowIndex,
  int colspan, int rowspan)
```

В первом параметре указывается ссылка на добавляемый узел, во втором — индекс столбца (нумерация начинается с 0), а в третьем — индекс строки (нумерация начинается с 0). Пример:

```
GridPane pane = new GridPane();
ColumnConstraints col1 = new ColumnConstraints(100);
ColumnConstraints col2 = new ColumnConstraints(100);
ColumnConstraints col3 = new ColumnConstraints(100);
pane.getColumnConstraints().addAll(col1, col2, col3);

RowConstraints row1 = new RowConstraints(50);
RowConstraints row2 = new RowConstraints(50);
RowConstraints row3 = new RowConstraints(50);
pane.getRowConstraints().addAll(row1, row2, row3);

for (int i = 0, c = pane.getColumnConstraints().size(); i < c; i++) {
    for (int j = 0, r = pane.getRowConstraints().size(); j < r; j++) {
        pane.add(new Label("label_" + i + "_" + j), i, j);
    }
}
```

Начиная с JavaFX 9, для получения количества столбцов и строк внутри таблицы можно воспользоваться методами `getColumnCount()` и `getRowCount()`:

```
for (int i = 0, c = pane.getColumnCount(); i < c; i++) {
    for (int j = 0, r = pane.getRowCount(); j < r; j++) {
        pane.add(new Label("label_" + i + "_" + j), i, j);
    }
}
```

Метод `add()` можно использовать, и не описывая столбцы и строки, но в этом случае характеристики столбцов и строк не будут заданы. Пример:

```
GridPane pane = new GridPane();
pane.add(new Label("label 1"), 0, 0);
pane.add(new Label("label 2"), 2, 2);
```

Параметр `colspan` задает количество объединяемых ячеек по горизонтали (число от 1 и далее), а параметр `rowspan` — количество объединяемых ячеек по вертикали (число от 1 и далее). Пример объединения двух ячеек по горизонтали:

```
GridPane pane = new GridPane();
pane.setGridLinesVisible(true); // Отображение сетки
Label label = new Label("label 1");
pane.add(label, 0, 0, 2, 1);
GridPane.setHalignment(label, HPos.CENTER);
pane.add(new Label("label 2"), 0, 2);
pane.add(new Label("label 3"), 1, 2);
pane.add(new Label("label 4"), 2, 2);
```

В этом примере мы воспользовались двумя дополнительными методами, чтобы результат был нагляднее. Метод `setGridLinesVisible()` служит для отображения границ таблицы на этапе отладки. Значение `true` включает отображение границ, а значение `false` — отключает. Формат метода:

```
public final void setGridLinesVisible(boolean value)
```

Проверить текущее значение и получить ссылку на свойство позволяют следующие методы:

```
public final boolean isGridLinesVisible()
public final BooleanProperty gridLinesVisibleProperty()
```

Статический метод `setHalignment()` задает горизонтальное выравнивание узла внутри ячейки. В примере мы указали значение `HPos.CENTER` для выравнивания надписи по центру объединенных ячеек, чтобы было видно, что ячейки объединены.

С помощью метода `addRow()` можно вставить строку в таблицу и сразу заполнить все ячейки узлами. Формат метода:

```
public void addRow(int rowIndex, Node... children)
```

Метод `addColumn()` позволяет вставить столбец в таблицу и сразу заполнить все ячейки узлами. Формат метода:

```
public void addColumn(int columnIndex, Node... children)
```

Пример:

```
GridPane pane = new GridPane();
pane.addRow(0, new Label("0"), new Label("1"), new Label("2"));
pane.addColumn(3, new Label("4"));
```

Помимо рассмотренных способов существует еще один: задать характеристики для каждого узла, а затем добавить узлы в таблицу с помощью метода `getChildren()`.

Пример:

```
GridPane pane = new GridPane();
pane.setGridLinesVisible(true);
Label label1 = new Label("0");
Label label2 = new Label("1");
Label label3 = new Label("2");
Label label4 = new Label("3");
GridPane.setColumnIndex(label1, 0);
GridPane.setRowIndex(label1, 0);
GridPane.setConstraints(label2, 1, 0);
GridPane.setConstraints(label3, 2, 0);
GridPane.setConstraints(label4, 3, 0);
Label label = new Label("label_0_1");
GridPane.setConstraints(label, 0, 1, 3, 1, HPos.CENTER, VPos.CENTER);
pane.getChildren().addAll(label1, label2, label3, label4, label);
```

Управлять характеристиками узла позволяют следующие статические методы:

- ❑ `setColumnIndex()` — задает индекс столбца (нумерация начинается с 0). Формат метода:

```
public static void setColumnIndex(Node child, Integer value)
```

- ❑ `getColumnIndex()` — возвращает индекс столбца. Формат метода:

```
public static Integer getColumnIndex(Node child)
```

- ❑ `setRowIndex()` — задает индекс строки (нумерация начинается с 0). Формат метода:

```
public static void setRowIndex(Node child, Integer value)
```

- ❑ `getRowIndex()` — возвращает индекс строки. Формат метода:

```
public static Integer getRowIndex(Node child)
```

- ❑ `setConstraints()` — позволяет указать сразу несколько характеристик узла.
Форматы метода:

[illegible]


```
public static void setConstraints(Node child,
                                int columnIndex, int rowIndex,
                                int colspan, int rowspan,
                                HPos halignment, VPos valignment,
                                Priority hgrow, Priority vgrow,
                                Insets margin)
```

В параметре `child` указывается ссылка на узел, в параметрах `columnIndex` и `rowIndex` задаются индексы столбца и строки соответственно, в параметрах `colspan` и `rowspan` — количество объединяемых ячеек по горизонтали и вертикали соответственно, в параметрах `halignment` и `valignment` — выравнивание по горизонтали и вертикали соответственно. Приоритеты растяжения можно задать в параметрах `hgrow` и `vgrow`. Параметр `margin` позволяет указать внешний отступ.

3.12.4. Объединение нескольких ячеек

Один узел может занимать сразу несколько ячеек. В предыдущих разделах мы познакомились с методами `add()` и `setConstraints()`, которые позволяют указать количество объединяемых ячеек в параметрах `colspan` и `rowspan`. Пример объединения двух ячеек по горизонтали:

```
GridPane pane = new GridPane();
Label label = new Label("label 1");
pane.add(label, 0, 0, 2, 1);
```

Помимо методов `add()` и `setConstraints()` можно воспользоваться следующими методами:

- ❑ `setColumnSpan()` — задает количество объединяемых ячеек по горизонтали. **Формат метода:**

```
public static void setColumnSpan(Node child, Integer value)
```

- ❑ `getColumnSpan()` — возвращает количество объединенных ячеек по горизонтали. **Формат метода:**

```
public static Integer getColumnSpan(Node child)
```

- ❑ `setRowSpan()` — задает количество объединяемых ячеек по вертикали. **Формат метода:**

```
public static void setRowSpan(Node child, Integer value)
```

- ❑ `getRowSpan()` — возвращает количество объединенных ячеек по вертикали. **Формат метода:**

```
public static Integer getRowSpan(Node child)
```

В качестве значения параметра `value` можно указать статическую константу `REMAINING`, которая означает, что узел занимает все оставшиеся строки или столбцы. **Формат:**

```
public static final int REMAINING
```

Выведем значение константы:

```
System.out.println(GridPane.REMAINING); // 2147483647
```

Пример указания значения константы `REMAINING` в FXML-файле:

```
<Button text="Button" GridPane.rowSpan="2147483647" />
```

Пример объединения двух ячеек по горизонтали:

```
GridPane pane = new GridPane();
Label label = new Label("label 1");
pane.add(label, 0, 0);
GridPane.setColumnSpan(label, 2);
```

3.12.5. Изменение свойств ячейки

Указать характеристики сразу всех ячеек внутри столбца или строки позволяют классы `ColumnConstraints` и `RowConstraints`. Для управления свойствами отдельных ячеек предназначены следующие методы:

- `setHalignment()` и `getHalignment()` — управляют горизонтальным выравниванием внутри ячейки. В параметре `value` можно указать константы: `LEFT`, `CENTER` или `RIGHT` из перечисления `javafx.geometry.HPos`. Форматы методов:

```
public static void setHalignment(Node child, HPos value)
public static HPos getHalignment(Node child)
```

- `setValignment()` и `getValignment()` — управляют вертикальным выравниванием внутри ячейки. В параметре `value` можно указать константы: `TOP`, `BASELINE`, `CENTER` или `BOTTOM` из перечисления `javafx.geometry.VPos`. Форматы методов:

```
public static void setValignment(Node child, VPos value)
public static VPos getValignment(Node child)
```

Пример:

```
Label label = new Label("label");
pane.add(label, 0, 0);
GridPane.setHalignment(label, HPos.LEFT);
GridPane.setValignment(label, VPos.TOP);
```

- `setHgrow()` и `getHgrow()` — управляют приоритетом изменения ширины ячейки при изменении размеров панели. В параметре `value` можно указать константы: `ALWAYS`, `SOMETIMES` или `NEVER` из перечисления `Priority`. Эти константы мы уже рассматривали в *разд. 3.11.1*. Если ширина столбца задана в процентах, то приоритет игнорируется. Форматы методов:

```
public static void setHgrow(Node child, Priority value)
public static Priority getHgrow(Node child)
```

- `setVgrow()` и `getVgrow()` — управляют приоритетом изменения высоты ячейки при изменении размеров панели. В параметре `value` можно указать константы: `ALWAYS`, `SOMETIMES` или `NEVER` из перечисления `Priority`. Эти константы мы уже

рассматривали в *разд. 3.11.1*. Если высота строки задана в процентах, то приоритет игнорируется. Форматы методов:

```
public static void setVgrow(Node child, Priority value)
public static Priority getVgrow(Node child)
```

- ❑ `setFillWidth()` и `isFillWidth()` — если в параметре `value` указано значение `true`, то узел будет заполнять всю ширину ячейки. Для некоторых узлов необходимо дополнительно указать значение `Double.MAX_VALUE` для максимального размера. Форматы методов:

```
public static void setFillWidth(Node child, Boolean value)
public static Boolean isFillWidth(Node child)
```

- ❑ `setFillHeight()` и `isFillHeight()` — если в параметре `value` указано значение `true`, то узел будет заполнять всю высоту ячейки. Для некоторых узлов необходимо дополнительно указать значение `Double.MAX_VALUE` для максимального размера. Форматы методов:

```
public static void setFillHeight(Node child, Boolean value)
public static Boolean isFillHeight(Node child)
```

- ❑ `setMargin()` и `getMargin()` — управляют внешними отступами. Форматы методов:

```
public static void setMargin(Node child, Insets value)
public static Insets getMargin(Node child)
```

Пример указания отступов в FXML-файле:

```
<TextField GridPane.rowIndex="1">
  <GridPane.margin>
    <Insets bottom="10.0" left="10.0" right="10.0" top="10.0" />
  </GridPane.margin>
</TextField>
```

ПРИМЕЧАНИЕ

Задать свойства ячейки позволяет также метод `setConstraints()` (см. *разд. 3.12.3*).

3.12.6. Изменение свойств таблицы

Управлять выравниванием таблицы внутри контейнера позволяют следующие методы:

- ❑ `setAlignment()` — задает выравнивание. В качестве параметра указываются следующие константы из перечисления `javafx.geometry.Pos`: `BASELINE_CENTER`, `BASELINE_LEFT`, `BASELINE_RIGHT`, `BOTTOM_CENTER`, `BOTTOM_LEFT`, `BOTTOM_RIGHT`, `CENTER`, `CENTER_LEFT`, `CENTER_RIGHT`, `TOP_CENTER`, `TOP_LEFT` и `TOP_RIGHT`. Формат метода:

```
public final void setAlignment(Pos value)
```

- ❑ `getAlignment()` — возвращает текущее значение. Формат метода:

```
public final Pos getAlignment()
```

- `alignmentProperty()` — возвращает ссылку на свойство. Формат метода:

```
public final ObjectProperty<Pos> alignmentProperty()
```

Управлять отображением границ таблицы на этапе отладки позволяют следующие методы:

- `setGridLinesVisible()` — значение `true` включает отображение границ, а значение `false` — отключает. Формат метода:

```
public final void setGridLinesVisible(boolean value)
```

- `isGridLinesVisible()` — возвращает текущее значение. Формат метода:

```
public final boolean isGridLinesVisible()
```

- `gridLinesVisibleProperty()` — возвращает ссылку на свойство. Формат метода:

```
public final BooleanProperty gridLinesVisibleProperty()
```

С помощью следующих методов можно управлять толщиной границ внутри таблицы:

- `setHgap()` — задает толщину вертикальных границ. Формат метода:

```
public final void setHgap(double value)
```

- `getHgap()` — возвращает текущую толщину вертикальных границ. Формат метода:

```
public final double getHgap()
```

- `hgapProperty()` — возвращает ссылку на свойство `hgap`. Формат метода:

```
public final DoubleProperty hgapProperty()
```

- `setVgap()` — задает толщину горизонтальных границ. Формат метода:

```
public final void setVgap(double value)
```

- `getVgap()` — возвращает текущую толщину горизонтальных границ. Формат метода:

```
public final double getVgap()
```

- `vgapProperty()` — возвращает ссылку на свойство `vgap`. Формат метода:

```
public final DoubleProperty vgapProperty()
```

Начиная с JavaFX 9, доступны следующие методы:

- `getColumnCount()` — возвращает количество столбцов таблицы. Формат метода:

```
public final int getColumnCount()
```

- `getRowCount()` — возвращает количество строк таблицы. Формат метода:

```
public final int getRowCount()
```

- `getCellBounds()` — возвращает информацию о положении и размерах ячейки с указанными индексами. Информация доступна только после отображения таблицы. Формат метода:

```
public final Bounds getCellBounds(int columnIndex, int rowIndex)
```

3.13. Класс *FlowPane*

Контейнер, описываемый классом `FlowPane`, позволяет выравнивать узлы по горизонтали или по вертикали. При выравнивании по горизонтали, в отличие от класса `HBox`, если узлы не помещаются на одной строке, то они будут «перетекать» на следующую строку. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.layout.FlowPane;
```

Иерархия наследования:

Object — Node — Parent — Region — Pane — FlowPane

Создать экземпляр класса позволяют следующие конструкторы:

```
FlowPane()
FlowPane(Node... children)
FlowPane(double hgap, double vgap)
FlowPane(double hgap, double vgap, Node... children)
FlowPane(Orientation orientation)
FlowPane(Orientation orientation, Node... children)
FlowPane(Orientation orientation, double hgap, double vgap)
FlowPane(Orientation orientation, double hgap, double vgap,
        Node... children)
```

Первый конструктор создает контейнер с выравниванием по горизонтали и отступами между узлами, равными нулю. Второй конструктор позволяет сразу добавить узлы в контейнер с горизонтальной ориентацией. Параметры `hgap` и `vgap` задают расстояние между узлами по горизонтали и вертикали соответственно. Управлять значениями этих параметров после создания объекта позволяют следующие методы:

```
public final void setHgap(double value)
public final double getHgap()
public final DoubleProperty hgapProperty()
public final void setVgap(double value)
public final double getVgap()
public final DoubleProperty vgapProperty()
```

Параметр `orientation` задает направление выравнивания узлов внутри контейнера. В качестве значения можно указать константы: `HORIZONTAL` или `VERTICAL` из перечисления `javafx.geometry.Orientation`. Формат:

```
public static final Orientation HORIZONTAL
public static final Orientation VERTICAL
```

Управлять направлением выравнивания после создания объекта позволяют следующие методы:

```
public final void setOrientation(Orientation value)
public final Orientation getOrientation()
public final ObjectProperty<Orientation> orientationProperty()
```

Для управления предпочтительным размером по ширине (для горизонтальной ориентации) или по высоте (для вертикальной ориентации) предназначены такие методы:

```
public final void setPrefWrapLength(double value)
public final double getPrefWrapLength()
public final DoubleProperty prefWrapLengthProperty()
```

Управлять выравниванием узлов внутри контейнера позволяют следующие методы:

- `setAlignment()` — задает выравнивание. В качестве параметра указываются следующие константы из перечисления `javafx.geometry.Pos`: `BASELINE_CENTER`, `BASELINE_LEFT`, `BASELINE_RIGHT`, `BOTTOM_CENTER`, `BOTTOM_LEFT`, `BOTTOM_RIGHT`, `CENTER`, `CENTER_LEFT`, `CENTER_RIGHT`, `TOP_CENTER`, `TOP_LEFT` и `TOP_RIGHT`. Формат метода:

```
public final void setAlignment(Pos value)
```

- `getAlignment()` — возвращает текущее значение. Формат метода:

```
public final Pos getAlignment()
```

- `alignmentProperty()` — возвращает ссылку на свойство. Формат метода:

```
public final ObjectProperty<Pos> alignmentProperty()
```

Для управления горизонтальным выравниванием внутри столбца предназначены такие методы:

- `setColumnHalignment()` — в параметре `value` можно указать константы: `LEFT`, `CENTER` или `RIGHT` из перечисления `javafx.geometry.HPos`. Формат метода:

```
public final void setColumnHalignment(HPos value)
```

- `getColumnHalignment()` — возвращает текущее значение. Формат метода:

```
public final HPos getColumnHalignment()
```

- `columnHalignmentProperty()` — возвращает ссылку на свойство. Формат метода:

```
public final ObjectProperty<HPos> columnHalignmentProperty()
```

Управлять вертикальным выравниванием внутри строки позволяют такие методы:

- `setRowValignment()` — в параметре `value` можно указать константы: `TOP`, `BASELINE`, `CENTER` или `BOTTOM` из перечисления `javafx.geometry.VPos`. Формат метода:

```
public final void setRowValignment(VPos value)
```

- `getRowValignment()` — возвращает текущее значение. Формат метода:

```
public final VPos getRowValignment()
```

- `rowValignmentProperty()` — возвращает ссылку на свойство. Формат метода:

```
public final ObjectProperty<VPos> rowValignmentProperty()
```

При использовании класса `FlowPane` мы можем указать внутренний отступ от границ контейнера до узлов (метод `setPadding()`), а также внешний отступ для каж-

дого узла по отдельности с помощью статического метода `setMargin()`. Формат метода:

```
public static void setMargin(Node child, Insets value)
```

Размер отступа задается с помощью класса `Insets` (см. *разд. 3.3.1*). Пример:

```
FlowPane.setMargin(button1, new Insets(20.0));
```

Получить размер внешнего отступа позволяет статический метод `getMargin()`. Формат метода:

```
public static Insets getMargin(Node child)
```

Добавить узлы в контейнер позволяют конструкторы класса, а также метод `getChildren()`. Пример добавления трех кнопок в контейнер с вертикальным выравниванием:

```
Button button1 = new Button("Кнопка 1");
Button button2 = new Button("Кнопка 2");
Button button3 = new Button("Кнопка 3");
FlowPane pane = new FlowPane();
pane.getChildren().addAll(button1, button2, button3);
```

Пример описания свойств контейнера в FXML-файле:

```
<FlowPane alignment="CENTER" columnHalignment="RIGHT"
    orientation="VERTICAL" prefHeight="300.0"
    prefWidth="300.0" rowValignment="TOP" vgap="10.0">
    <children>
        <Button mnemonicParsing="false" text="Button 1" />
        <Button mnemonicParsing="false" text="Button 2" />
        <Button mnemonicParsing="false" text="Button 3" />
    </children>
</FlowPane>
```

ЭЛЕКТРОННЫЙ АРХИВ

Пример использования класса `FlowPane` содержится в проекте `FX_CH03_13_FlowPane`, приведенном в каталоге `CH03_containers` сопровождающего книгу электронного архива (см. *приложение*).

3.14. Класс *BorderPane*

Класс `BorderPane` описывает контейнер, внутри которого узлы могут быть прикреплены к одной из четырех сторон или помещены в центр (рис. 3.3). Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.layout.BorderPane;
```

Иерархия наследования:

```
Object — Node — Parent — Region — Pane — BorderPane
```

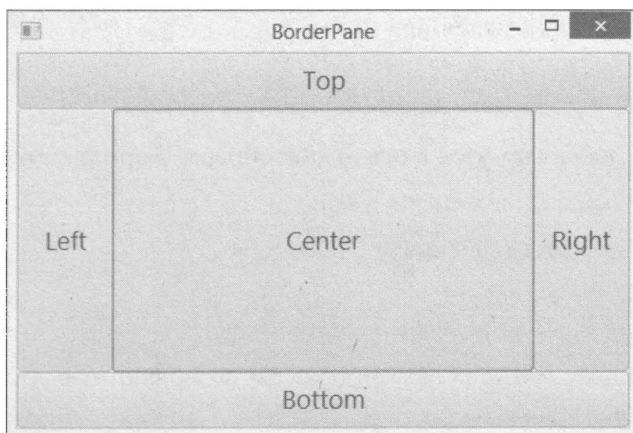


Рис. 3.3. Схема размещения компонентов в контейнере BorderLayout

Создать экземпляры класса позволяют следующие конструкторы:

```
BorderPane()
```

```
BorderPane(Node center)
```

```
BorderPane(Node center, Node top, Node right, Node bottom, Node left)
```

Первый конструктор создает пустой контейнер, второй конструктор позволяет добавить узел в центр контейнера, а третий конструктор — сразу несколько узлов в пять областей контейнера. Добавить узлы после создания объекта позволяют следующие методы:

❑ `setLeft()` — добавляет узел в область слева. Формат метода:

```
public final void setLeft(Node value)
```

Методы для получения значения:

```
public final Node getLeft()
```

```
public final ObjectProperty<Node> leftProperty()
```

❑ `setTop()` — добавляет узел в область сверху. Формат метода:

```
public final void setTop(Node value)
```

Методы для получения значения:

```
public final Node getTop()
```

```
public final ObjectProperty<Node> topProperty()
```

❑ `setRight()` — добавляет узел в область справа. Формат метода:

```
public final void setRight(Node value)
```

Методы для получения значения:

```
public final Node getRight()
```

```
public final ObjectProperty<Node> rightProperty()
```

❑ `setBottom()` — добавляет узел в область снизу. Формат метода:

```
public final void setBottom(Node value)
```


Методы для получения значения:

```
public final Node getBottom()
public final ObjectProperty<Node> bottomProperty()
```

- ❑ **setCenter()** — добавляет узел в центр контейнера. Формат метода:

```
public final void setCenter(Node value)
```

Методы для получения значения:

```
public final Node getCenter()
public final ObjectProperty<Node> centerProperty()
```

Управлять выравниванием узла внутри области позволяют следующие статические методы из класса `BorderPane`:

- ❑ **setAlignment()** — задает выравнивание для узла внутри области. В качестве параметра указываются следующие константы из перечисления `javafx.geometry.Pos`: `BASELINE_CENTER`, `BASELINE_LEFT`, `BASELINE_RIGHT`, `BOTTOM_CENTER`, `BOTTOM_LEFT`, `BOTTOM_RIGHT`, `CENTER`, `CENTER_LEFT`, `CENTER_RIGHT`, `TOP_CENTER`, `TOP_LEFT` и `TOP_RIGHT`. Формат метода:

```
public static void setAlignment(Node child, Pos value)
```

- ❑ **getAlignment()** — возвращает текущее значение. Формат метода:

```
public static Pos getAlignment(Node child)
```

При использовании класса `BorderPane` мы можем указать внутренний отступ от границ контейнера до узлов (метод `setPadding()`), а также внешний отступ для каждого узла по отдельности с помощью статического метода `setMargin()`. Формат метода:

```
public static void setMargin(Node child, Insets value)
```

Размер отступа задается с помощью класса `Insets` (см. *разд. 3.3.1*). Пример:

```
BorderPane.setMargin(button2, new Insets(20.0));
```

Получить размер внешнего отступа позволяет статический метод `getMargin()`. Формат метода:

```
public static Insets getMargin(Node child)
```

При использовании программы `Scene Builder` компонент `BorderPane` можно найти на вкладке **Containers**. Пример описания свойств контейнера в FXML-файле:

```
<BorderPane prefHeight="400.0" prefWidth="600.0">
  <top>
    <Button text="Button" BorderPane.alignment="CENTER">
      <BorderPane.margin>
        <Insets bottom="20.0" left="20.0" right="20.0" top="20.0" />
      </BorderPane.margin>
    </Button>
  </top>
```

```
<left>
    <Button text="Button" BorderPane.alignment="CENTER" />
</left>
<center>
    <Button text="Button" BorderPane.alignment="CENTER" />
</center>
<right>
    <Button text="Button" BorderPane.alignment="CENTER" />
</right>
<bottom>
    <Button text="Button" BorderPane.alignment="CENTER" />
</bottom>
</BorderPane>
```

ЭЛЕКТРОННЫЙ АРХИВ

Пример использования класса `BorderPane` содержится в проекте `FX_CH03_14_BorderPane`, приведенном в каталоге `CH03_containers` сопровождающего книги электронного архива (см. приложение).

3.15. Класс *StackPane*

Класс `StackPane` реализует контейнер, в котором узлы расположены друг над другом. Первый добавленный узел будет расположен в самом низу стопки, а последний узел — на самом верху. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.layout.StackPane;
```

Иерархия наследования:

`Object` — `Node` — `Parent` — `Region` — `Pane` — `StackPane`

Создать экземпляр класса позволяют следующие конструкторы:

```
StackPane()
StackPane(Node... children)
```

Первый конструктор создает пустой контейнер, а второй конструктор позволяет сразу добавить узлы в контейнер. Добавить узлы в контейнер после создания объекта можно с помощью метода `getChildren()`. Пример:

```
StackPane pane = new StackPane();
Rectangle rect1 = new Rectangle(150.0, 140.0);
rect1.setFill(Color.DODGERBLUE);
Rectangle rect2 = new Rectangle(240.0, 100.0);
rect2.setFill(Color.LIGHTGREEN);
pane.getChildren().addAll(rect1, rect2);
```

По умолчанию узлы внутри контейнера выравниваются по центру. Изменить выравнивание всех узлов позволяют следующие методы:

```
public final void setAlignment(Pos value)
public final Pos getAlignment()
public final ObjectProperty<Pos> alignmentProperty()
```

Для изменения выравнивания только одного узла можно воспользоваться такими статическими методами из класса `StackPane`:

```
public static void setAlignment(Node child, Pos value)
public static Pos getAlignment(Node child)
```

С помощью статического метода `setMargin()` из класса `StackPane` можно указать внешние отступы для отдельного узла. Формат метода:

```
public static void setMargin(Node child, Insets value)
```

Получить размеры отступов позволяет статический метод `getMargin()`. Формат метода:

```
public static Insets getMargin(Node child)
```

При использовании программы `Scene Builder` компонент `StackPane` можно найти на вкладке **Containers**. Пример описания свойств контейнера в `FXML`-файле:

```
<StackPane prefHeight="400.0" prefWidth="600.0">
  <children>
    <Rectangle arcHeight="5.0" arcWidth="5.0" fill="DODGERBLUE"
      height="141.0" stroke="BLACK" strokeType="INSIDE"
      width="157.0" />
    <Rectangle arcHeight="5.0" arcWidth="5.0" fill="#ff1f54"
      height="102.0" stroke="BLACK" strokeType="INSIDE"
      width="247.0" StackPane.alignment="CENTER">
      <StackPane.margin>
        <Insets top="10.0" />
      </StackPane.margin>
    </Rectangle>
  </children>
</StackPane>
```

ЭЛЕКТРОННЫЙ АРХИВ

Пример использования класса `StackPane` содержится в проекте `FX_CH03_15_StackPane`, приведенном в каталоге `CH03_containers` сопровождающего книгу электронного архива (см. приложение).

3.16. Класс *TilePane*

Контейнер, описываемый классом `TilePane`, позволяет выравнивать узлы по горизонтали или по вертикали. Каждый узел находится внутри прямоугольной области (плитки). Все плитки имеют одинаковый размер. При выравнивании по горизонтали, если узлы не помещаются на одной строке, то они будут «перетекать» на следующую строку. При выравнивании по вертикали, если узлы не помещаются

в одном столбце, то они будут «перетекать» в соседний столбец. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.layout.TilePane;
```

Иерархия наследования:

Object – Node – Parent – Region – Pane – TilePane

Создать экземпляр класса позволяют следующие конструкторы:

```
TilePane()
TilePane(Node... children)
TilePane(double hgap, double vgap)
TilePane(double hgap, double vgap, Node... children)
TilePane(Orientation orientation)
TilePane(Orientation orientation, Node... children)
TilePane(Orientation orientation, double hgap, double vgap)
TilePane(Orientation orientation, double hgap, double vgap,
        Node... children)
```

Первый конструктор создает контейнер с выравниванием по горизонтали и отступами между плитками, равными нулю. Второй конструктор позволяет сразу добавить узлы в контейнер с горизонтальной ориентацией. Параметры `hgap` и `vgap` задают расстояние между плитками по горизонтали и вертикали соответственно. Управлять значениями этих параметров после создания объекта позволяют следующие методы:

```
public final void setHgap(double value)
public final double getHgap()
public final DoubleProperty hgapProperty()
public final void setVgap(double value)
public final double getVgap()
public final DoubleProperty vgapProperty()
```

Параметр `orientation` задает направление выравнивания узлов внутри контейнера. В качестве значения можно указать константы: `HORIZONTAL` или `VERTICAL` из перечисления `javafx.geometry.Orientation`. Формат:

```
public static final Orientation HORIZONTAL
public static final Orientation VERTICAL
```

Управлять направлением выравнивания после создания объекта позволяют следующие методы:

```
public final void setOrientation(Orientation value)
public final Orientation getOrientation()
public final ObjectProperty<Orientation> orientationProperty()
```

Управлять предпочтительным количеством столбцов и рядов (по умолчанию равно 5) позволяют такие методы:

```
public final void setPrefColumns(int value)
public final int getPrefColumns()
public final IntegerProperty prefColumnsProperty()
public final void setPrefRows(int value)
public final int getPrefRows()
public final IntegerProperty prefRowsProperty()
```

Для управления предпочтительными размерами плитки (значение по умолчанию: `Region.USE_COMPUTED_SIZE`, равное `-1.0`) предназначены такие методы:

```
public final void setPrefTileWidth(double value)
public final double getPrefTileWidth()
public final DoubleProperty prefTileWidthProperty()
public final void setPrefTileHeight(double value)
public final double getPrefTileHeight()
public final DoubleProperty prefTileHeightProperty()
```

Получить текущие размеры плитки позволяют следующие методы:

```
public final double getTileWidth()
public final ReadOnlyDoubleProperty tileWidthProperty()
public final double getTileHeight()
public final ReadOnlyDoubleProperty tileHeightProperty()
```

Управлять выравниванием всех плиток внутри контейнера можно с помощью следующих методов:

```
public final void setAlignment(Pos value)
public final Pos getAlignment()
public final ObjectProperty<Pos> alignmentProperty()
```

В параметре `value` указываются следующие константы из перечисления `javafx.geometry.Pos`: `BASELINE_CENTER`, `BASELINE_LEFT`, `BASELINE_RIGHT`, `BOTTOM_CENTER`, `BOTTOM_LEFT`, `BOTTOM_RIGHT`, `CENTER`, `CENTER_LEFT`, `CENTER_RIGHT`, `TOP_CENTER`, `TOP_LEFT` и `TOP_RIGHT`.

Для управления выравниванием узла внутри одной плитки предназначены такие статические методы из класса `TilePane`:

```
public static void setAlignment(Node node, Pos value)
public static Pos getAlignment(Node node)
```

Пример:

```
TilePane.setAlignment(button, Pos.CENTER);
```

Управлять выравниванием узлов внутри сразу всех плиток можно с помощью следующих методов:

```
public final void setTileAlignment(Pos value)
public final Pos getTileAlignment()
public final ObjectProperty<Pos> tileAlignmentProperty()
```

При использовании класса `TilePane` мы можем указать внутренний отступ от границ контейнера до плиток (метод `setPadding()`), а также внешний отступ для каж-

ного узла по отдельности с помощью статического метода `setMargin()`. Формат метода:

```
public static void setMargin(Node node, Insets value)
```

Получить размер внешнего отступа позволяет статический метод `getMargin()`. Формат метода:

```
public static Insets getMargin(Node node)
```

Добавить узлы в контейнер позволяют конструкторы класса, а также метод `getChildren()`. Пример добавления трех кнопок в контейнер с вертикальным выравниванием:

```
Button button1 = new Button("Кнопка 1");
Button button2 = new Button("Кнопка 2");
Button button3 = new Button("Кнопка 3");
TilePane pane = new TilePane();
pane.getChildren().addAll(button1, button2, button3);
```

При использовании программы Scene Builder компонент `TilePane` можно найти на вкладке **Containers**. Пример описания свойств контейнера в FXML-файле:

```
<TilePane alignment="CENTER" hgap="10.0" orientation="VERTICAL"
    prefHeight="300.0" prefTileHeight="50.0"
    prefTileWidth="100.0" prefWidth="300.0" vgap="10.0">
  <children>
    <Button text="Кнопка 1" TilePane.alignment="TOP_RIGHT" />
    <Button text="Кнопка 2">
      <TilePane.margin>
        <Insets right="30.0" />
      </TilePane.margin>
    </Button>
    <Button text="Кнопка 3" />
  </children>
</TilePane>
```

ЭЛЕКТРОННЫЙ АРХИВ

Пример использования класса `TilePane` содержится в проекте `FX_CH03_16_TilePane`, приведенном в каталоге `CH03_containers` сопровождающего книгу электронного архива (см. приложение).

3.17. Класс *AnchorPane*: привязка узла к сторонам панели

Класс `AnchorPane` позволяет указать привязку узла к сторонам панели. Если задана привязка к двум противоположным сторонам панели, то будет изменяться размер узла. Благодаря этому можно, например, «растянуть» узел на всю ширину панели.

Если привязки не указаны, то используется абсолютное позиционирование. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.layout.AnchorPane;
```

Иерархия наследования:

Object — Node — Parent — Region — Pane — AnchorPane

Создать экземпляр класса позволяют следующие конструкторы:

```
AnchorPane()
AnchorPane(Node... children)
```

Добавить узел после создания объекта можно с помощью метода `getChildren()`.

Пример:

```
Button button = new Button("Кнопка");
AnchorPane pane = new AnchorPane();
pane.getChildren().add(button);
```

После добавления все узлы будут помещены в позицию с координатами (0, 0). Последний добавленный узел будет расположен на вершине этой стопки. Чтобы распределить узлы по поверхности панели, можно воспользоваться методами: `setLayoutX()`, `setLayoutY()` или `relocate()`. Все эти методы мы уже рассматривали в *разд. 3.4*.

Вместо указания абсолютного местоположения можно выполнить привязку узла к одной из сторон контейнера. Управлять привязкой позволяют следующие статические методы из класса `AnchorPane`:

```
// К верхней границе
public static void setTopAnchor(Node child, Double value)
public static Double getTopAnchor(Node child)
// К правой границе
public static void setRightAnchor(Node child, Double value)
public static Double getRightAnchor(Node child)
// К нижней границе
public static void setBottomAnchor(Node child, Double value)
public static Double getBottomAnchor(Node child)
// К левой границе
public static void setLeftAnchor(Node child, Double value)
public static Double getLeftAnchor(Node child)
```

Пример привязки ко всем четырем сторонам контейнера:

```
Button button = new Button("Кнопка");
AnchorPane pane = new AnchorPane();
pane.getChildren().add(button);
AnchorPane.setTopAnchor(button, 5.0);
AnchorPane.setRightAnchor(button, 5.0);
AnchorPane.setBottomAnchor(button, 5.0);
AnchorPane.setLeftAnchor(button, 5.0);
```

Чтобы удалить привязку, следует передать в соответствующий метод значение `null`. Удалить сразу все привязки позволяет статический метод `clearConstraints()`.
Формат метода:

```
public static void clearConstraints(Node child)
```

При использовании программы Scene Builder компонент `AnchorPane` можно найти на вкладке **Containers**. Щелкаем левой кнопкой мыши на компоненте и, не отпуская кнопку мыши, перетаскиваем его в центральную область окна. Аналогичным образом добавляем в контейнер кнопку (компонент `Button` из вкладки **Controls**). Чтобы задать привязку, выделяем кнопку и отображаем вкладку **Layout** (раздел **Anchor Pane Constraints**). Выделяем нужное поле и вводим значение отступа, например, от правой границы контейнера до правой границы узла. Граница стороны с установленной привязкой будет выделена на схеме красным цветом (рис. 3.4). Чтобы удалить привязку, достаточно щелкнуть левой кнопкой мыши на линии между узлом и границей контейнера на схеме.

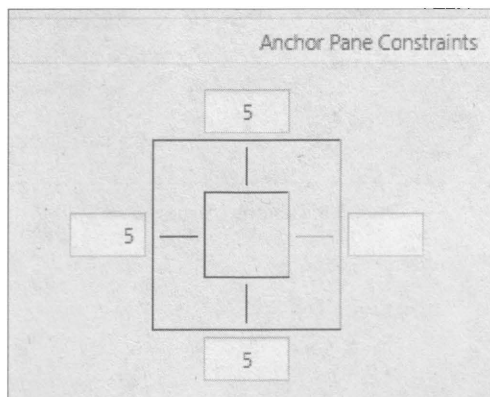


Рис. 3.4. Указание привязки к сторонам контейнера в программе Scene Builder

Пример описания свойств контейнера в FXML-файле:

```
<AnchorPane prefHeight="300.0" prefWidth="300.0">
  <children>
    <Button layoutX="146.0" layoutY="105.0"
      text="Кнопка"
      AnchorPane.bottomAnchor="5.0"
      AnchorPane.leftAnchor="5.0"
      AnchorPane.rightAnchor="5.0"
      AnchorPane.topAnchor="5.0" />
  </children>
</AnchorPane>
```

ЭЛЕКТРОННЫЙ АРХИВ

Пример использования класса `AnchorPane` содержится в проекте `FX_CH03_17_AnchorPane`, приведенном в каталоге `CH03_containers` сопровождающего книгу электронного архива (см. приложение).

3.18. Класс *TabPane*: панель с вкладками

Для создания панели с вкладками (рис. 3.5) предназначен класс `TabPane`. Панель состоит из области заголовка с ярлыками и набора вкладок с различными компонентами. В один момент времени показывается содержимое только одной вкладки. Щелчок мышью на ярлыке в области заголовка приводит к отображению содержимого соответствующей вкладки. Инструкция импорта класса `TabPane`:

```
// Модуль javafx.controls
import javafx.scene.control.TabPane;
```

Иерархия наследования:

Object — Node — Parent — Region — Control — TabPane

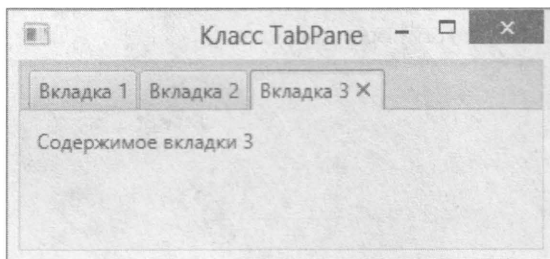


Рис. 3.5. Панель с вкладками

Конструкторы класса:

```
TabPane()
TabPane(Tab... tabs)
```

Первый конструктор создает пустой контейнер, а второй позволяет сразу добавить вкладки в контейнер, указав их через запятую. Для добавления вкладок после создания объекта используется метод `getTabs()`. Формат метода:

```
public final ObservableList<Tab> getTabs()
```

Пример создания панели и добавления одной вкладки:

```
TabPane pane = new TabPane();

Tab tab1 = new Tab();
tab1.setText("Вкладка 1");
tab1.setContent(new Label("Содержимое вкладки 1"));
pane.getTabs().add(tab1);
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH03_18_TabPane`, приведенном в каталоге `CH03_containers` сопровождающего книгу электронного архива (см. приложение).

3.18.1. Класс *Tab*: вкладка

Отдельная вкладка реализуется с помощью класса `Tab`. Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.Tab;
```

Конструкторы класса:

```
Tab()
Tab(String text)
Tab(String text, Node content)
```

В параметре `text` указывается текст надписи на ярлыке вкладки в области заголовка. Параметр `content` задает узел, который будет отображаться на вкладке, при условии, что вкладка активна. Добавим вторую вкладку в контейнер:

```
Tab tab2 = new Tab("Вкладка 2", new Label("Содержимое вкладки 2"));
pane.getTabs().add(tab2);
```

Рассмотрим основные свойства класса `Tab`:

- ❑ `text` — текст надписи на ярлыке вкладки в области заголовка. Методы доступа:

```
public final void setText(String value)
public final String getText()
public final StringProperty textProperty()
```
- ❑ `graphic` — узел, отображаемый слева от текста надписи на ярлыке вкладки. Методы доступа:

```
public final void setGraphic(Node value)
public final Node getGraphic()
public final ObjectProperty<Node> graphicProperty()
```
- ❑ `content` — узел, отображаемый на вкладке. Методы доступа:

```
public final void setContent(Node value)
public final Node getContent()
public final ObjectProperty<Node> contentProperty()
```
- ❑ `closable` — если свойство имеет значение `true`, то вкладку можно будет закрыть с помощью кнопки на ярлыке вкладки. Методы доступа:

```
public final void setClosable(boolean value)
public final boolean isClosable()
public final BooleanProperty closableProperty()
```
- ❑ `onClosed` — позволяет назначить обработчик события закрытия вкладки. Методы доступа:

```
public final void setOnClosed(EventHandler<Event> value)
public final EventHandler<Event> getOnClosed()
public final ObjectProperty<EventHandler<Event>>
    onClosedProperty()
```

- ❑ `onCloseRequest` — позволяет назначить обработчик события закрытия вкладки. Если внутри обработчика через объект события вызвать метод `consume()`, то вкладка закрыта не будет. Методы доступа:

```
public void setOnCloseRequest(EventHandler<Event> value)
public EventHandler<Event> getOnCloseRequest()
public final ObjectProperty<EventHandler<Event>>
    onCloseRequestProperty()
```

- ❑ `disable` — если указать значение `true`, то ярлык вкладки станет недоступным, и сделать активной вкладку пользователь не сможет. Методы доступа:

```
public final void setDisable(boolean value)
public final boolean isDisable()
public final BooleanProperty disableProperty()
```

- ❑ `disabled` — содержит значение `true`, если вкладка недоступна, и `false` — в противном случае. Методы доступа:

```
public final boolean isDisabled()
public final ReadOnlyBooleanProperty disabledProperty()
```

- ❑ `selected` — содержит значение `true`, если вкладка выбрана (активна), и `false` — в противном случае. Методы доступа:

```
public final boolean isSelected()
public final ReadOnlyBooleanProperty selectedProperty()
```

Чтобы сделать вкладку активной, можно воспользоваться следующим кодом:

```
pane.getSelectionModel().select(tab2);
```

- ❑ `onSelectionChanged` — позволяет назначить обработчик события изменения активности вкладки. Методы доступа:

```
public final void setOnSelectionChanged(
    EventHandler<Event> value)
public final EventHandler<Event> getOnSelectionChanged()
public final ObjectProperty<EventHandler<Event>>
    onSelectionChangedProperty()
```

- ❑ `tabPane` — содержит ссылку на панель или значение `null`. Методы доступа:

```
public final TabPane getTabPane()
public final ReadOnlyObjectProperty<TabPane> tabPaneProperty()
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH03_18_TabPane`, приведенном в каталоге `CH03_containers` сопровождающего книгу электронного архива (см. приложение).

3.18.2. Свойства класса *TabPane*

Класс `TabPane` содержит следующие основные свойства:

- ❑ `tabMinWidth` — минимальная ширина ярлыка вкладки. Методы доступа:

```
public final void setTabMinWidth(double value)
public final double getTabMinWidth()
public final DoubleProperty tabMinWidthProperty()
```

- ❑ `tabMinHeight` — минимальная высота ярлыка вкладки. Методы доступа:

```
public final void setTabMinHeight(double value)
public final double getTabMinHeight()
public final DoubleProperty tabMinHeightProperty()
```

- ❑ `tabMaxWidth` — максимальная ширина ярлыка вкладки. Методы доступа:

```
public final void setTabMaxWidth(double value)
public final double getTabMaxWidth()
public final DoubleProperty tabMaxWidthProperty()
```

- ❑ `tabMaxHeight` — максимальная высота ярлыка вкладки. Методы доступа:

```
public final void setTabMaxHeight(double value)
public final double getTabMaxHeight()
public final DoubleProperty tabMaxHeightProperty()
```

- ❑ `side` — задает сторону, на которой будут отображаться ярлыки вкладок. Можно указать константы: `LEFT`, `RIGHT`, `BOTTOM` и `TOP` (значение по умолчанию) из перечисления `Side`. Методы доступа:

```
import javafx.geometry.Side;
public final void setSide(Side value)
public final Side getSide()
public final ObjectProperty<Side> sideProperty()
```

- ❑ `rotateGraphic` — если свойство имеет значение `true`, то узел, отображаемый слева от надписи, будет поворачиваться вместе с текстом надписи. Методы доступа:

```
public final void setRotateGraphic(boolean value)
public final boolean isRotateGraphic()
public final BooleanProperty rotateGraphicProperty()
```

- ❑ `selectionModel` — содержит ссылку на модель выбора вкладки. Описание класса `SingleSelectionModel<T>` см. в *разд. 13.3*. Методы доступа:

```
import javafx.scene.control.SingleSelectionModel;
public final void setSelectionModel(
    SingleSelectionModel<Tab> value)
public final SingleSelectionModel<Tab> getSelectionModel()
public final ObjectProperty<SingleSelectionModel<Tab>>
    selectionModelProperty()
```

Чтобы сделать вкладку активной, можно воспользоваться следующим кодом:

```
pane.getSelectionModel().select(tab2);
```

- **tabClosingPolicy** — задает режим закрытия вкладок пользователем. Методы доступа:

```
public final void setTabClosingPolicy(
    TabPane.TabClosingPolicy value)
public final TabPane.TabClosingPolicy getTabClosingPolicy()
public final ObjectProperty<TabPane.TabClosingPolicy>
    tabClosingPolicyProperty()
```

Перечисление `TabPane.TabClosingPolicy` содержит следующие константы:

- `UNAVAILABLE` — пользователь не может закрыть вкладки;
- `SELECTED_TAB` — можно закрыть активную вкладку при условии, что свойство `closable` из класса `Tab` имеет значение `true` (значение по умолчанию);
- `ALL_TABS` — пользователь может закрыть любую вкладку, не делая ее активной, при условии, что свойство `closable` из класса `Tab` имеет значение `true`;

- **tabDragPolicy** — задает режим перетаскивания ярлыков вкладок. Можно указать константы: `FIXED` (ярлыки вкладок перетаскивать нельзя, значение по умолчанию) и `REORDER` (пользователь может изменить порядок следования ярлыков вкладок путем перетаскивания). Свойство доступно, начиная с JavaFX 10. Методы доступа:

```
public final void setTabDragPolicy(TabPane.TabDragPolicy value)
public final TabPane.TabDragPolicy getTabDragPolicy()
public final ObjectProperty<TabPane.TabDragPolicy>
    tabDragPolicyProperty()
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH03_18_TabPane`, приведенном в каталоге `CH03_containers` сопровождающего книгу электронного архива (см. приложение).

3.19. Класс *TitledPane*: панель с заголовком

Класс `TitledPane` создает панель с заголовком (рис. 3.6). При щелчке мышью на заголовке содержимое панели скрывается или отображается с анимацией. Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.TitledPane;
```

Иерархия наследования:

`Object` — `Node` — `Parent` — `Region` — `Control` — `Labeled` — `TitledPane`

Конструкторы класса:

```
TitledPane()
TitledPane(String title, Node content)
```

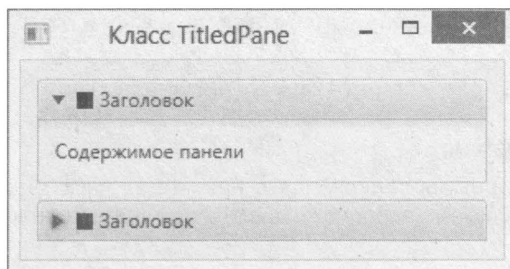


Рис. 3.6. Панель с заголовком

В параметре `title` указывается текст, который отобразится в заголовке, а в параметре `content` — узел, добавляемый на панель. Пример:

```
VBox vbox = new VBox();
vbox.setPadding(new Insets(10,0));
Label label = new Label("Содержимое панели");
vbox.getChildren().add(label);

TitledPane titledPane = new TitledPane();
titledPane.setText("Заголовок");
titledPane.setContent(vbox);
titledPane.setGraphic(new ImageView(createImage(10, 10, Color.BLUE)));
titledPane.setCollapsible(true);
titledPane.setAnimated(true);
```

Класс `TitledPane` наследует все свойства из класса `Labeled` (см. *разд. 11.3*) и содержит следующие свойства:

- ☐ `content` — задает узел, добавляемый на панель. Методы доступа:

```
public final void setContent(Node value)
public final Node getContent()
public final ObjectProperty<Node> contentProperty()
```
- ☐ `expanded` — если свойство имеет значение `true`, то содержимое панели будет отображено, а если `false` — то скрыто. Методы доступа:

```
public final void setExpanded(boolean value)
public final boolean isExpanded()
public final BooleanProperty expandedProperty()
```
- ☐ `animated` — если свойство имеет значение `true`, то содержимое панели будет отображаться и скрываться с анимацией, а если `false` — то сразу без анимации. Методы доступа:

```
public final void setAnimated(boolean value)
public final boolean isAnimated()
public final BooleanProperty animatedProperty()
```
- ☐ `collapsible` — если свойство имеет значение `true`, то содержимое панели можно будет скрывать и отображать с помощью щелчка мышью на заголовке панели, а если `false` — то нет. Методы доступа:

```
public final void setCollapsible(boolean value)
public final boolean isCollapsible()
public final BooleanProperty collapsibleProperty()
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH03_19_TitledPane`, приведенном в каталоге `CH03_containers` сопровождающего книгу электронного архива (см. приложение).

3.20. Класс *Accordion*: панель «аккордеон»

Класс `Accordion` позволяет создать контейнер с несколькими вкладками (рис. 3.7). Изначально отображается содержимое лишь одной вкладки, а у остальных доступны только заголовки. После щелчка мышью на заголовке вкладки она открывается, а остальные сворачиваются. Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.Accordion;
```

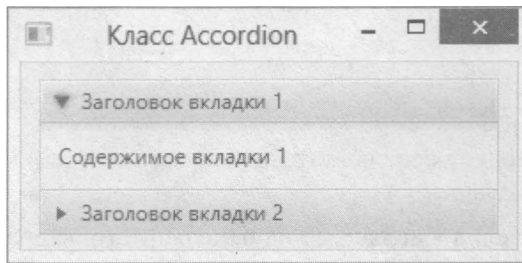


Рис. 3.7. Панель «аккордеон»

Иерархия наследования:

`Object` — `Node` — `Parent` — `Region` — `Control` — `Accordion`

Конструкторы класса:

```
Accordion()
Accordion(TitledPane... titledPanes)
```

Каждая вкладка реализуется с помощью класса `TitledPane` (см. разд. 3.19). Вкладки можно добавить с помощью конструктора или метода `getPanes()`. Формат метода:

```
public final ObservableList<TitledPane> getPanes()
```

Отобразить или свернуть содержимое вкладки позволяет свойство `expandedPane`.

Методы доступа:

```
public final void setExpandedPane(TitledPane value)
public final TitledPane getExpandedPane()
public final ObjectProperty<TitledPane> expandedPaneProperty()
```

Пример:

```
VBox vbox = new VBox();
vbox.setPadding(new Insets(10.0));
Label label = new Label("Содержимое вкладки 1");
vbox.getChildren().add(label);

TitledPane titledPane = new TitledPane("Заголовок вкладки 1", vbox);

TitledPane titledPane2 = new TitledPane("Заголовок вкладки 2",
    new Label("Содержимое вкладки 2"));

Accordion accordion = new Accordion();
accordion.getPanes().addAll(titledPane, titledPane2);
accordion.setExpandedPane(titledPane);
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH03_20_Accordion`, приведенном в каталоге `CH03_containers` сопровождающего книгу электронного архива (см. приложение).

3.21. Класс *SplitPane*: панель с изменяемыми размерами областей

Класс `SplitPane` реализует горизонтальную (по умолчанию) или вертикальную панель с разделительными линиями. Размеры областей на панели пользователь может изменять, перемещая разделительную линию с помощью мыши. Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.SplitPane;
```

Иерархия наследования:

Object — Node — Parent — Region — Control — SplitPane

Конструкторы класса:

```
SplitPane()
SplitPane(Node... items)
```

Первый конструктор создает горизонтальную панель с настройками по умолчанию, а второй конструктор позволяет сразу добавить контейнеры, указав их через запятую. Добавить контейнеры после создания объекта можно с помощью метода `getItems()`. Формат метода:

```
public ObservableList<Node> getItems()
```

Изменить ориентацию панели позволяет свойство `orientation`. В качестве значения указываются константы `HORIZONTAL` и `VERTICAL` из перечисления `Orientation`. Методы доступа:


```
public final void setOrientation(Orientation value)
public final Orientation getOrientation()
public final ObjectProperty<Orientation> orientationProperty()
```

Управлять размерами областей из программы позволяют следующие методы:

- ❑ `setDividerPositions()` — задает позиции разделительных линий через запятую. В качестве значений указываются вещественные числа от 0.0 до 1.0. Формат метода:

```
public void setDividerPositions(double... positions)
```

- ❑ `setDividerPosition()` — задает позицию разделительной линии с указанным индексом. Формат метода:

```
public void setDividerPosition(int dividerIndex, double position)
```

- ❑ `getDividerPositions()` — возвращает массив с позициями разделительных линий. Формат метода:

```
public double[] getDividerPositions()
```

- ❑ `getDividers()` — возвращает список с объектами разделительных линий. Формат метода:

```
public ObservableList<SplitPane.Divider> getDividers()
```

Класс `SplitPane.Divider` содержит свойство `position`, которое задает позицию разделительной линии. Методы доступа:

```
public final void setPosition(double value)
public final double getPosition()
public final DoubleProperty positionProperty()
```

- ❑ `setResizableWithParent()` — если во втором параметре указано значение `false`, то при изменении размеров панели размеры узла `node` изменяться не будут. Формат метода:

```
public static void setResizableWithParent(
    Node node, Boolean value)
```

- ❑ `isResizableWithParent()` — возвращает значение `false`, если при изменении размеров панели размеры узла `node` изменяться не будут. Формат метода:

```
public static Boolean isResizableWithParent(Node node)
```

Пример создания панели с тремя областями:

```
VBox vbox = new VBox();
vbox.setPadding(new Insets(10.0));
vbox.getChildren().add(new Label("Содержимое области 1"));

VBox vbox2 = new VBox();
vbox2.setPadding(new Insets(10.0));
vbox2.getChildren().add(new Label("Содержимое области 2"));
```

```
VBox vbox3 = new VBox();  
vbox3.setPadding(new Insets(10.0));  
vbox3.getChildren().add(new Label("Содержимое области 3"));  
SplitPane.setResizableWithParent(vbox3, false);
```

```
SplitPane splitPane = new SplitPane();  
splitPane.setOrientation(Orientation.HORIZONTAL);  
splitPane.getItems().addAll(vbox, vbox2, vbox3);  
splitPane.setDividerPositions(0.3, 0.7);
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH03_21_SplitPane`, приведенном в каталоге `CH03_containers` сопровождающего книгу электронного архива (см. приложение).

ГЛАВА 4



Обработка событий

При взаимодействии пользователя с окном происходят *события*. В ответ на события генерируются определенные сигналы. *Сигналы* — это своего рода извещения о том, что пользователь выполнил какое-либо действие, или в самой системе возникло некоторое условие. События являются важнейшей составляющей приложения с графическим интерфейсом, поэтому необходимо знать, как назначить обработчик события, как удалить обработчик, а также уметь правильно обработать событие. Какие сигналы генерирует тот или иной компонент, мы будем рассматривать при изучении конкретного компонента.

4.1. Назначение обработчиков событий

Назначить обработчик события можно несколькими способами:

- с помощью методов, названия которых начинаются с префикса `setOn`. Например, в предыдущих примерах мы много раз назначали обработчик нажатия кнопки с помощью метода `setOnAction()`:

```
Button button = new Button("Кнопка");
button.setOnAction(event -> {
    System.out.println("Нажата кнопка");
});
```

Такие методы изменяют значение соответствующего JavaFX-свойства. Например, метод `setOnAction()` задает значение свойства `onAction`. Получить текущее значение можно с помощью метода, название которого начинается с приставки `getOn`, например `getOnAction()`. Если обработчик не был назначен, то метод вернет значение `null`. Получить ссылку на свойство `onAction` позволяет метод `onActionProperty()`;

- с помощью методов `addEventFilter()` и `addEventHandler()`. Обработчик, назначенный методом `addEventFilter()`, вызывается перед обработчиком, назначенным методом `addEventHandler()`. Обратите внимание: эти методы позволяют зарегистрировать несколько обработчиков. Форматы методов:

```
public final <T extends Event> void addEventFilter(
    EventType<T> eventType, EventHandler<? super T> eventFilter)
public final <T extends Event> void addEventHandler(
    EventType<T> eventType, EventHandler<? super T> eventHandler)
```

Пример назначения обработчика нажатия кнопки с помощью метода addEventHandler():

```
button.addEventHandler(ActionEvent.ACTION, event -> {
    System.out.println("Нажата кнопка");
});
```

Если внутри обработчика, назначенного с помощью метода addEventFilter(), вызвать метод consume() через объект события, то обработчик, назначенный с помощью метода addEventHandler(), вызван не будет. Пример:

```
button.addEventFilter(ActionEvent.ACTION, event -> {
    System.out.println("Нажата кнопка (Filter)");
    event.consume(); // Прерываем всплытие события
});
button.addEventHandler(ActionEvent.ACTION, event -> {
    // Обработчик вызван не будет
    System.out.println("Нажата кнопка (Handler)");
});
```

- ❑ изменения различных значений можно отслеживать с помощью JavaFX-свойств. В этом случае для назначения обработчика используется метод addListener(). Формат метода:

```
public void addListener(ChangeListener<? super T> listener)
```

Интерфейс ChangeListener<T> содержит объявление метода changed():

```
public void changed(ObservableValue<? extends T> observable,
    T oldValue, T newValue)
```

Например, отследить изменение положения окна по оси x можно следующим образом:

```
stage.xProperty().addListener((obj, oldValue, newValue) -> {
    System.out.println("Изменение X " + newValue);
});
```

В метод регистрации обработчика мы можем передать:

- ❑ объект класса, реализующего функциональный интерфейс EventHandler<T extends Event>. Интерфейс содержит объявление метода handle():

```
public void handle(T event)
```

- ❑ анонимный вложенный класс:

```
button1.setOnAction(new EventHandler<ActionEvent>() {
    @Override
```

```
public void handle(ActionEvent event) {  
    System.out.println("Нажата кнопка 1");  
}  
});
```

□ **ссылку на метод:**

```
button2.setOnAction(this::onClick);
```

□ **лямбда-выражение:**

```
button3.setOnAction(event -> {  
    System.out.println("Нажата кнопка 3");  
});
```

При использовании программы Scene Builder названия методов, вызываемых при наступлении события, указываются на вкладке **Code** в соответствующих полях. Например, чтобы назначить обработчик нажатия кнопки, сначала выделяем кнопку на форме, переходим на вкладку **Code** и вводим значение (например, **onClick**) в поле **On Action**. Внутри FXML-файла название обработчика станет значением атрибута `onAction`, причем перед названием обработчика указывается символ `#`:

```
<Button onAction="#onClick" text="Кнопка" />
```

Внутри класса контроллера реализация обработчика будет выглядеть примерно следующим образом:

```
@FXML  
private void onClick(ActionEvent event) {  
    System.out.println("Нажата кнопка");  
}
```

Пример назначения обработчиков нажатия кнопки разными способами приведен в листинге 4.1.

Листинг 4.1. Назначение обработчиков событий

```
package application;  
  
import javafx.application.Application;  
import javafx.event.ActionEvent;  
import javafx.event.EventHandler;  
import javafx.geometry.Pos;  
import javafx.scene.Scene;  
import javafx.scene.control.Button;  
import javafx.scene.layout.VBox;  
import javafx.stage.Stage;  
  
public class Main extends Application {  
  
    public static void main(String[] args) {  
        Application.launch(args);  
    }  
}
```

```

@Override
public void start(Stage stage) throws Exception {
    VBox root = new VBox(15.0);
    root.setAlignment(Pos.CENTER);
    Button button1 = new Button("Кнопка 1");
    Button button2 = new Button("Кнопка 2");
    Button button3 = new Button("Кнопка 3");
    Button button4 = new Button("Кнопка 4");
    Button button5 = new Button("Кнопка 5");
    root.getChildren().addAll(button1, button2, button3,
                               button4, button5);
    button1.setOnAction(new EventHandler<ActionEvent>() {
        @Override
        public void handle(ActionEvent event) {
            System.out.println("Нажата кнопка 1");
        }
    });
    button2.setOnAction(this::onClick);
    button3.setOnAction(event -> {
        System.out.println("Нажата кнопка 3");
    });
    button4.addEventFilter(ActionEvent.ACTION, event -> {
        System.out.println("Нажата кнопка 4 (Filter)");
    });
    button4.addEventHandler(ActionEvent.ACTION, event -> {
        System.out.println("Нажата кнопка 4 (Handler)");
    });
    button5.addEventFilter(ActionEvent.ACTION, event -> {
        System.out.println("Нажата кнопка 5 (Filter)");
        event.consume(); // Прерываем всплывание события
    });
    button5.addEventHandler(ActionEvent.ACTION, event -> {
        // Обработчик вызван не будет
        System.out.println("Нажата кнопка 5 (Handler)");
    });
    Scene scene = new Scene(root, 400.0, 300.0);
    stage.setTitle("Назначение обработчиков");
    stage.setScene(scene);
    stage.xProperty().addListener((obj, oldValue, newValue) -> {
        System.out.println("Изменение X " + newValue);
    });
    stage.show();
}

private void onClick(ActionEvent event) {
    System.out.println("Нажата кнопка 2");
}
}

```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH04_01_setOn`, приведенном в каталоге `CH04_events` сопровождающего книгу электронного архива (см. приложение).

4.2. Блокировка и удаление обработчика

Если необходимо временно блокировать обработчики событий для узла, то достаточно сделать узел неактивным с помощью метода `setDisable()` из класса `Node`. Пример:

```
button.setDisable(true);
```

Если обработчик назначался с помощью метода, название которого начинается с приставки `setOn`, то для удаления обработчика можно вызвать тот же самый метод, но вместо ссылки передать ему значение `null`. Пример:

```
button.setOnAction(null);
```

Если обработчик назначался с помощью метода `addEventHandler()`, то для удаления обработчика нужно воспользоваться методом `removeEventHandler()`. Формат метода:

```
public final <T extends Event> void removeEventHandler(  
    EventType<T> eventType, EventHandler<? super T> eventHandler)
```

В первом параметре метод принимает тип события, а во втором — ссылку на обработчик. Чтобы правильно удалить обработчик, необходимо указать ссылку на тот же самый обработчик, который был и при назначении. Лучше сохранить ссылку на обработчик в какой-либо переменной, а затем использовать ее и при назначении обработчика, и при удалении его. Пример сохранения ссылок в полях класса:

```
private EventHandler<ActionEvent> handler1 = this::onClick;  
private EventHandler<ActionEvent> handler2 = event -> {  
    System.out.println("Нажата кнопка 3");  
};
```

Пример назначения и удаления обработчика:

```
button2.addEventHandler(ActionEvent.ACTION, handler1);  
// ...  
button2.removeEventHandler(ActionEvent.ACTION, handler1);
```

Если обработчик назначался с помощью метода `addEventFilter()`, то для удаления обработчика нужно воспользоваться методом `removeEventFilter()`. Формат метода:

```
public final <T extends Event> void removeEventFilter(  
    EventType<T> eventType, EventHandler<? super T> eventFilter)
```

Пример назначения и удаления обработчика:

```
button3.addEventFilter(ActionEvent.ACTION, handler2);  
// ...  
button3.removeEventFilter(ActionEvent.ACTION, handler2);
```


Если обработчик назначался с помощью метода `addListener()`, то для удаления обработчика нужно воспользоваться методом `removeListener()`. Формат метода:

```
public void removeListener(ChangeListener<? super T> listener)
```

Не забудьте предварительно сохранить ссылку на обработчик, например, так:

```
private ChangeListener<? super Number>
    listener = (obj, oldValue, newValue) -> {
        System.out.println("Изменение X " + newValue);
    };
```

Пример назначения и удаления обработчика:

```
stage.xProperty().addListener(listener);
// ...
stage.xProperty().removeListener(listener);
```

ЭЛЕКТРОННЫЙ АРХИВ

Пример удаления обработчиков событий содержится в проекте `FX_CH04_02_removeEventHandler`, приведенном в каталоге `CH04_events` сопровождающего книгу электронного архива (см. приложение).

4.3. Последовательность вызова обработчиков

Мы можем назначить обработчики несколькими способами, а в какой последовательности они будут выполняться, если использовать все способы сразу для одного узла? Последовательность вызова при нажатии кнопки будет выглядеть так:

```
button — Filter
button — Handler
button — setOnAction
```

Итак, вначале выполняется обработчик, назначенный с помощью метода `addEventFilter()`. Внутри этого обработчика мы можем фильтровать событие. Если необходимо прервать распространение события, то внутри обработчика через объект события нужно вызвать метод `consume()`. Пример:

```
button.addEventFilter(ActionEvent.ACTION, event -> {
    System.out.println("button — Filter");
    event.consume();
});
```

После вызова метода `consume()` последовательность будет такой:

```
button — Filter
```

Иными словами, другие обработчики вызваны не будут. Если внутри обработчика метод `consume()` не вызвать, то распространение события продолжится.

Методы `addEventFilter()` и `addEventHandler()` позволяют зарегистрировать несколько обработчиков. Последовательность вызова обработчиков в этом случае будет такой:

```
button - Filter
button - Filter2
button - Handler
button - Handler2
button - setOnAction
```

Методы `addEventFilter()` и `addEventHandler()` имеют все узлы, а также объекты сцены и окна. Если назначить обработчик нажатия кнопки для объекта контейнера, то можно перехватить нажатие любых кнопок внутри контейнера. Если назначить обработчик для объектов сцены или окна, то внутри обработчика будут доступны события нажатий вообще всех кнопок. Предположим, что мы назначили обработчики сразу для кнопки, контейнера, сцены и окна. Последовательность событий в этом случае будет такой:

```
stage - Filter
scene - Filter
root - Filter
button - Filter
button - Handler
button - setOnAction
root - Handler
scene - Handler
stage - Handler
```

Итак, событие вначале движется от объекта окна до источника, вызвавшего событие, а затем обратно к объекту окна. Вот таким образом событие проходит по всей древовидной иерархии от объекта окна до узла и обратно. Если на какой-либо стадии внутри обработчика вызвать метод `consume()` через объект события, то распространение события будет остановлено.

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH04_03_consume`, приведенном в каталоге `CH04_events` сопровождающего книгу электронного архива (см. *приложение*).

4.4. Генерация события из программы

В некоторых случаях необходимо вызвать событие из программы. Например, при заполнении последнего текстового поля и нажатии клавиши `<Enter>` можно имитировать нажатие кнопки и тем самым запустить обработчик этого события. Выполнить генерацию события из программы позволяет статический метод `fireEvent()` из класса `Event`. Формат метода:

```
import javafx.event.Event;
public static void fireEvent(EventTarget eventTarget, Event event)
```

В первом параметре указывается экземпляр класса, реализующего интерфейс `EventTarget`. Этот интерфейс реализуют все классы контейнеров и компонентов, а также классы сцены и окна, поэтому любой узел можно указать в качестве цели. Во втором параметре задается объект события. По нажатию одной кнопки сгенерируем событие для другой кнопки:

```
button2.setOnAction(event -> {
    System.out.println("button2");
    Event.fireEvent(button1, event);
});
```

Для генерации события можно также воспользоваться методом `fireEvent()` из классов `Node` и `Window`. Формат метода:

```
public final void fireEvent(Event event)
```

Пример:

```
button3.setOnAction(event -> {
    System.out.println("button3");
    button1.fireEvent(event);
});
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH04_04_fireEvent`, приведенном в каталоге `CH04_events` сопровождающего книгу электронного архива (см. приложение).

4.5. Класс *Event*

Класс `Event` является базовым классом для всех классов событий в JavaFX. Инструкция импорта класса:

```
// Модуль javafx.base
import javafx.event.Event;
```

Иерархия наследования:

```
Object — java.util.EventObject — Event
```

Конструкторы класса:

```
Event(EventType<? extends Event> eventType)
Event(Object source, EventTarget target,
    EventType<? extends Event> eventType)
```

Приведем основные методы (полный список смотрите в документации):

❑ `getEventType()` — возвращает тип события. Формат метода:

```
public EventType<? extends Event> getEventType()
```

❑ `getSource()` — возвращает ссылку на объект, который обрабатывает событие. Например, если обработчик нажатия кнопки зарегистрирован для объекта кон-

тейнера, то при нажатии кнопки внутри обработчика мы получим ссылку на контейнер, а не на кнопку. Метод наследуется из класса `EventObject`. Формат метода:

```
public Object getSource()
```

- ❑ `getTarget()` — возвращает ссылку на цель события. Формат метода:

```
public EventTarget getTarget()
```

- ❑ `fireEvent()` — позволяет сгенерировать событие (см. *разд. 4.4*). Формат метода:

```
public static void fireEvent(EventTarget eventTarget,  
                             Event event)
```

- ❑ `consume()` — устанавливает флаг прерывания распространения события (см. *разд. 4.3*). Формат метода:

```
public void consume()
```

- ❑ `isConsumed()` — возвращает статус флага прерывания распространения события. Формат метода:

```
public boolean isConsumed()
```

- ❑ `clone()` — создает копию объекта события. Формат метода:

```
public Object clone()
```

Класс `Event` содержит также следующие статические константы:

- ❑ `NULL_SOURCE_TARGET` — обозначает неизвестную цель события. Формат:

```
public static final EventTarget NULL_SOURCE_TARGET
```

- ❑ `ANY` — базовый тип для всех типов событий. Формат:

```
public static final EventType<Event> ANY
```

Иерархия основных классов событий выглядит следующим образом:

`Event`

<code>ActionEvent</code>	(например, нажатие кнопки)
<code>InputEvent</code>	(события ввода)
<code>KeyEvent</code>	(события клавиатуры, см. <i>разд. 4.8</i>)
<code>MouseEvent</code>	(события мыши, см. <i>разд. 4.9</i>)
<code>MouseDownEvent</code>	
<code>TouchEvent</code>	(события сенсорного экрана)
<code>GestureEvent</code>	(события жестов)
<code>ScrollEvent</code>	(прокрутка)
<code>RotateEvent</code>	(поворот)
<code>ZoomEvent</code>	(масштабирование)
<code>DragEvent</code>	(технология drag & drop, см. <i>разд. 4.11</i>)
<code>ContextMenuEvent</code>	(событие вызова контекстного меню, см. <i>разд. 15.10</i>)
<code>WindowEvent</code>	(события окна, см. <i>разд. 4.7</i>)

Типы событий также представлены в виде иерархии. Пример иерархии типов:

```
Event.ANY
  InputEvent.ANY
    KeyEvent.ANY
      KeyEvent.KEY_PRESSED
      KeyEvent.KEY_RELEASED
      KeyEvent.KEY_TYPED
    MouseEvent.ANY
      MouseEvent.MOUSE_PRESSED
      MouseEvent.MOUSE_RELEASED
      MouseEvent.MOUSE_CLICKED
      MouseEvent.MOUSE_ENTERED
      MouseEvent.MOUSE_EXITED
      MouseEvent.MOUSE_ENTERED_TARGET
      MouseEvent.MOUSE_EXITED_TARGET
      MouseEvent.MOUSE_MOVED
      MouseEvent.MOUSE_DRAGGED
      MouseDragEvent.ANY
        MouseDragEvent.MOUSE_DRAG_ENTERED
        MouseDragEvent.MOUSE_DRAG_EXITED
        MouseDragEvent.MOUSE_DRAG_ENTERED_TARGET
        MouseDragEvent.MOUSE_DRAG_EXITED_TARGET
        MouseDragEvent.MOUSE_DRAG_OVER
        MouseDragEvent.MOUSE_DRAG_RELEASED
```

Каждый класс события, помимо целевых типов, содержит тип `ANY`, который используется для перехвата любого целевого события. Если мы хотим, например, перехватывать все события ввода (события клавиатуры, мыши, сенсорного экрана и др.), то при назначении обработчика достаточно указать базовый класс `InputEvent` и его тип `ANY`. Пример обработки всех событий ввода:

```
scene.addEventFilter(InputEvent.ANY, event -> {
    System.out.println("Событие " + event.getEventType());
});
```

ЭЛЕКТРОННЫЙ АРХИВ

Пример использования класса `Event` содержится в проекте `FX_CH04_05_Event`, приведенном в каталоге `CH04_events` сопровождающего книгу электронного архива (см. приложение).

4.6. Использование таймеров

Таймеры позволяют через определенный интервал времени выполнять какую-либо задачу. Типичным примером необходимости использования таймера является отображение текущего времени с точностью до секунды.

ПРИМЕЧАНИЕ

Для создания таймера можно также воспользоваться классами `Transition` (см. *разд. 9.3*) и `AnimationTimer` (см. *разд. 9.15*).

4.6.1. Класс *Timer*

Создать таймер позволяет класс `Timer` из пакета `java.util`. Инструкция импорта:

```
// Модуль java.base
import java.util.Timer;
```

Конструкторы класса:

```
Timer()
Timer(boolean isDaemon)
Timer(String name)
Timer(String name, boolean isDaemon)
```

Первый конструктор создает таймер с настройками по умолчанию. Второй и четвертый конструкторы в параметре `isDaemon` позволяют указать, будет поток с таймером обычным (по умолчанию) или демоном (необходимо явно указать значение `true`). Поток-демон автоматически завершается (в любой стадии выполнения!) при отсутствии обычных потоков, тогда как обычный поток с таймером необходимо завершать явным образом (с помощью метода `cancel()`), иначе после закрытия всех окон приложения он так и продолжит выполняться. Параметр `name` задает название потока с таймером. Если название не задано явным образом, то поток будет называться, например, `Timer-0`. Пример создания таймера в потоке-демоде с названием `MyTimer`:

```
Timer timer = new Timer("MyTimer", true);
```

Класс `Timer` содержит следующие основные методы:

- ❑ `cancel()` — отменяет все задачи, но не прерывает выполнение текущей задачи. После завершения выполнения всех задач поток завершает работу. Формат метода:

```
public void cancel()
```

- ❑ `schedule()` — планирует выполнение задачи. Форматы метода:

```
import java.util.Date;
import java.util.TimerTask;
public void schedule(TimerTask task, Date time)
public void schedule(TimerTask task, Date firstTime,
                    long period)
public void schedule(TimerTask task, long delay)
public void schedule(TimerTask task, long delay, long period)
```

Первый формат планирует однократное выполнение задачи `task` в указанное время `time`. Если указанное время в прошлом, то задача выполняется немедленно.

но. Второй формат в параметре `firstTime` задает время первого выполнения задачи, а в параметре `period` — задержку (в миллисекундах) между последующими запусками. Третий формат планирует однократное выполнение задачи `task` после задержки `delay` (значение указывается в миллисекундах). Четвертый формат в параметре `delay` задает задержку (в миллисекундах) перед первым выполнением задачи, а в параметре `period` — задержку (в миллисекундах) между последующими запусками.

4.6.2. Класс *TimerTask*

Задача для таймера описывается с помощью абстрактного класса `TimerTask`. Инструкция импорта:

```
import java.util.TimerTask;
```

Мы должны наследовать класс `TimerTask` и переопределить абстрактный метод `run()`. Формат метода:

```
public void run()
```

Код, расположенный внутри метода `run()`, будет выполнен в потоке таймера один раз или многократно в зависимости от настроек таймера. Чтобы прервать выполнение задачи, следует вызвать метод `cancel()`. Формат метода:

```
public boolean cancel()
```

4.6.3. Изменение свойств узлов из разных потоков

Итак, задача выполняется в потоке таймера. Если внутри этой задачи необходимо изменять свойства узлов пользовательского интерфейса, то напрямую это сделать не получится. Свойства узлов можно менять только из потока диспетчера обработки событий (поток с названием `JavaFX Application Thread`). Любая попытка изменить свойства узлов из других потоков будет приводить к исключению `IllegalStateException`.

Чтобы изменить свойства узла из потока таймера, необходимо сформировать задачу (интерфейс `Runnable`) и передать ее статическому методу `runLater()` из класса `Platform`. В этом случае задача будет выполнена в потоке диспетчера обработки событий. Формат метода:

```
import javafx.application.Platform;  
public static void runLater(Runnable runnable)
```

Функциональный интерфейс `Runnable` содержит объявление метода `run()`:

```
public void run()
```

В качестве примера использования таймера создадим в окне часы, которые будут отображать текущее системное время с точностью до секунды (листинг 4.2).

Листинг 4.2. Вывод времени в окне с точностью до секунды

```
package application;

import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.util.Timer;
import java.util.TimerTask;
import javafx.application.Application;
import javafx.application.Platform;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class Main extends Application {
    private Timer timer;

    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) throws Exception {
        VBox root = new VBox();
        root.setAlignment(Pos.CENTER);
        Label label = new Label("");
        label.setStyle("-fx-font-size: 40pt");
        timer = new Timer();
        timer.schedule(new MyTimerTask(label), 0, 1000);
        root.getChildren().addAll(label);
        Scene scene = new Scene(root, 400, 150);
        stage.setTitle("Часы с точностью до секунды");
        stage.setScene(scene);
        stage.show();
    }

    @Override
    public void stop() {
        timer.cancel();
    }
}

class MyTimerTask extends TimerTask {
    private Label lbl;
```



```

public MyTimerTask(Label lbl) {
    this.lbl = lbl;
}

@Override
public void run() {
    Platform.runLater(() -> {
        lbl.setText(LocalTime.now().format(
            DateTimeFormatter.ofPattern("HH:mm:ss")));
    });
}
}

```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH04_06_Timer`, приведенном в каталоге `CH04_events` сопровождающего книгу электронного архива (см. приложение).

4.7. События окна

События отображения и сокрытия окна описываются с помощью класса `WindowEvent`. Инструкция импорта:

```

// Модуль javafx.graphics
import javafx.stage.WindowEvent;

```

Иерархия наследования:

`Object` — `EventObject` — `Event` — `WindowEvent`

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH04_07_WindowEvent`, приведенном в каталоге `CH04_events` сопровождающего книгу электронного архива (см. приложение).

4.7.1. Отображение, сокрытие и закрытие окна

В классе `WindowEvent` определены следующие типы событий:

- `ANY` — любое из приведенных далее событий. Формат:

```
public static final EventType<WindowEvent> ANY
```

Пример назначения обработчика:

```

stage.addEventHandler(WindowEvent.ANY, event -> {
    System.out.println("ANY - " + event.getEventType());
});

```

- `WINDOW_SHOWING` — событие непосредственно перед отображением окна. Формат:

```
public static final EventType<WindowEvent> WINDOW_SHOWING
```

Пример назначения обработчика:

```
stage.addEventHandler(WindowEvent.WINDOW_SHOWING, event -> {  
    System.out.println("WINDOW_SHOWING");  
});
```

Управлять обработчиком события позволяют также следующие методы из класса `Window`:

```
public final void setOnShowing(EventHandler<WindowEvent> value)  
public final EventHandler<WindowEvent> getOnShowing()  
public final ObjectProperty<EventHandler<WindowEvent>>  
    onShowingProperty()
```

❑ **WINDOW_SHOWN** — событие после отображения окна. Формат:

```
public static final EventType<WindowEvent> WINDOW_SHOWN
```

Управлять обработчиком события позволяют также следующие методы из класса `Window`:

```
public final void setOnShown(EventHandler<WindowEvent> value)  
public final EventHandler<WindowEvent> getOnShown()  
public final ObjectProperty<EventHandler<WindowEvent>>  
    onShownProperty()
```

❑ **WINDOW_CLOSE_REQUEST** — событие запроса на закрытие окна, например, при нажатии кнопки **Закрыть** в заголовке окна. Формат:

```
public static final EventType<WindowEvent> WINDOW_CLOSE_REQUEST
```

Управлять обработчиком события позволяют также следующие методы из класса `Window`:

```
public final void setOnCloseRequest(  
    EventHandler<WindowEvent> value)  
public final EventHandler<WindowEvent> getOnCloseRequest()  
public final ObjectProperty<EventHandler<WindowEvent>>  
    onCloseRequestProperty()
```

❑ **WINDOW_HIDING** — событие непосредственно перед сокрытием окна. Формат:

```
public static final EventType<WindowEvent> WINDOW_HIDING
```

Управлять обработчиком события позволяют также следующие методы из класса `Window`:

```
public final void setOnHiding(EventHandler<WindowEvent> value)  
public final EventHandler<WindowEvent> getOnHiding()  
public final ObjectProperty<EventHandler<WindowEvent>>  
    onHidingProperty()
```

❑ **WINDOW_HIDDEN** — событие после сокрытия окна. Формат:

```
public static final EventType<WindowEvent> WINDOW_HIDDEN
```

Управлять обработчиком события позволяют также следующие методы из класса `Window`:

```
public final void setOnHidden(EventHandler<WindowEvent> value)
public final EventHandler<WindowEvent> getOnHidden()
public final ObjectProperty<EventHandler<WindowEvent>>
    onHiddenProperty()
```

Отслеживать изменение состояния видимости окна позволяет свойство `showing`. Получить ссылку на свойство можно с помощью метода `showingProperty()`. Формат метода:

```
public final ReadOnlyBooleanProperty showingProperty()
```

Пример:

```
stage.showingProperty().addListener((obj, oldValue, newValue) -> {
    System.out.println("Состояние окна изменилось: " + newValue);
});
```

Отслеживать изменения фокуса ввода позволяет свойство `focused`. Получить ссылку на свойство можно с помощью метода `focusedProperty()`. Формат метода:

```
public final ReadOnlyBooleanProperty focusedProperty()
```

Пример назначения обработчика:

```
stage.focusedProperty().addListener((obj, oldValue, newValue) -> {
    System.out.println("focusedProperty: " + newValue);
});
```

Если необходимо предотвратить закрытие окна, то внутри обработчика события `WINDOW_CLOSE_REQUEST` необходимо вызвать метод `consume()` через объект события. Это может потребоваться, например, когда есть несохраненные данные, и пользователь пытается закрыть окно с помощью кнопки **Закрыть** в заголовке окна. Будет разумно перед закрытием окна спросить пользователя, хочет ли он сохранить данные, и если хочет, то запретить закрытие окна и вывести диалоговое окно сохранения данных. Пример запроса подтверждения закрытия окна приведен в листинге 4.3.

Листинг 4.3. Запрос подтверждения закрытия окна

```
stage.setOnCloseRequest(event -> {
    Alert dialog = new Alert(AlertType.CONFIRMATION,
        "Вы действительно хотите закрыть окно?");
    dialog.setTitle("Закрытие окна");
    dialog.setHeaderText(null);
    Optional<ButtonType> result = dialog.showAndWait();
    if (result.isPresent() && result.get() == ButtonType.OK) {
        System.out.println("Окно будет закрыто");
    }
})
```

```
else {  
    System.out.println("Окно не будет закрыто!");  
    event.consume();  
}  
});
```

В этом примере мы воспользовались классом `Alert` (см. *разд. 16.2*) для вывода модального диалогового окна с двумя кнопками: **OK** и **Cancel** (рис. 4.1). Класс `Alert` доступен, начиная с версии 8.40.

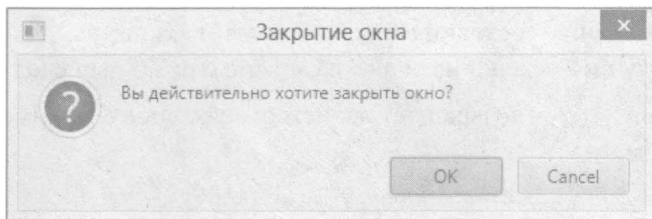


Рис. 4.1. Диалоговое окно с запросом подтверждения закрытия окна

4.7.2. Изменение положения окна и его размеров

Отслеживать изменения положения окна позволяют свойства `x` и `y`. Получить ссылки на свойства можно с помощью следующих методов:

❑ `xProperty()` — координата левого верхнего угла окна по оси `x`. Формат метода:

```
public final ReadOnlyDoubleProperty xProperty()
```

❑ `yProperty()` — координата левого верхнего угла окна по оси `y`. Формат метода:

```
public final ReadOnlyDoubleProperty yProperty()
```

Пример назначения обработчика события изменения положения окна:

```
stage.xProperty().addListener((obj, oldValue, newValue) -> {  
    System.out.println("Координата X изменилась: " + newValue);  
});  
stage.yProperty().addListener((obj, oldValue, newValue) -> {  
    System.out.println("Координата Y изменилась: " + newValue);  
});
```

Отслеживать изменения размеров окна позволяют свойства `width` и `height`. Получить ссылки на свойства можно с помощью следующих методов:

❑ `widthProperty()` — возвращает свойство, связанное с шириной. Формат метода:

```
public final ReadOnlyDoubleProperty widthProperty()
```

❑ `heightProperty()` — возвращает свойство, связанное с высотой. Формат метода:

```
public final ReadOnlyDoubleProperty heightProperty()
```

Пример отслеживания изменения размеров окна:

```
stage.widthProperty().addListener((obj, oldValue, newValue) -> {
    System.out.println("Ширина изменилась: " + newValue);
});
stage.heightProperty().addListener((obj, oldValue, newValue) -> {
    System.out.println("Высота изменилась: " + newValue);
});
```

4.7.3. Разворачивание и сворачивание окна

Отслеживать изменения состояния окна позволяют свойства `iconified`, `maximized` и `fullScreen`. Получить ссылки на свойства можно с помощью следующих методов:

❑ `iconifiedProperty()` — возвращает свойство, связанное с режимом сворачивания окна. Формат метода:

```
public final ReadOnlyBooleanProperty iconifiedProperty()
```

❑ `maximizedProperty()` — возвращает свойство, связанное с режимом максимального раскрытия окна. Формат метода:

```
public final ReadOnlyBooleanProperty maximizedProperty()
```

❑ `fullScreenProperty()` — возвращает свойство, связанное с полноэкранным режимом. Формат метода:

```
public final ReadOnlyBooleanProperty fullScreenProperty()
```

Пример назначения обработчика:

```
stage.iconifiedProperty().addListener((obj, oldValue, newValue) -> {
    System.out.println("iconifiedProperty: " + newValue);
});
```

4.8. События клавиатуры

События клавиатуры описываются с помощью класса `KeyEvent`. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.input.KeyEvent;
```

Иерархия наследования:

```
Object — EventObject — Event — InputEvent — KeyEvent
```

ЭЛЕКТРОННЫЙ АРХИВ

Пример назначения обработчиков содержится в проекте `FX_CH04_08_KeyEvent`, приведенном в каталоге `CH04_events` сопровождающего книгу электронного архива (см. *приложение*).

4.8.1. Типы событий

В классе `KeyEvent` определены следующие типы событий:

- **ANY** — любое из приведенных далее событий. Формат:

```
public static final EventType<KeyEvent> ANY
```

Пример назначения обработчика:

```
stage.addEventHandler(KeyEvent.ANY, event -> {  
    System.out.println("ANY - " + event.getEventType());  
});
```

- **KEY_PRESSED** — событие нажатия клавиши на клавиатуре. Если клавишу удерживать нажатой, то событие генерируется постоянно, пока клавиша не будет отпущена. Формат:

```
public static final EventType<KeyEvent> KEY_PRESSED
```

Управлять обработчиком события позволяют также следующие методы из классов `Node` и `Scene`:

```
public final void setOnKeyPressed(  
    EventHandler<? super KeyEvent> value)  
public final EventHandler<? super KeyEvent> getOnKeyPressed()  
public final ObjectProperty<EventHandler<? super KeyEvent>>  
    onKeyPressedProperty()
```

- **KEY_RELEASED** — событие при отпуске ранее нажатой клавиши. Формат:

```
public static final EventType<KeyEvent> KEY_RELEASED
```

Управлять обработчиком события позволяют также следующие методы из классов `Node` и `Scene`:

```
public final void setOnKeyReleased(  
    EventHandler<? super KeyEvent> value)  
public final EventHandler<? super KeyEvent> getOnKeyReleased()  
public final ObjectProperty<EventHandler<? super KeyEvent>>  
    onKeyReleasedProperty()
```

- **KEY_TYPED** — событие при вводе символа `Unicode`. При нажатии функциональных клавиш, например `<Shift>`, `<Ctrl>` и др., событие не генерируется. Формат:

```
public static final EventType<KeyEvent> KEY_TYPED
```

Управлять обработчиком события позволяют также следующие методы из классов `Node` и `Scene`:

```
public final void setOnKeyTyped(  
    EventHandler<? super KeyEvent> value)  
public final EventHandler<? super KeyEvent> getOnKeyTyped()  
public final ObjectProperty<EventHandler<? super KeyEvent>>  
    onKeyTypedProperty()
```

Пример получения символа:

```
root.setOnKeyTyped(event -> {
    System.out.println(event.getCharacter());
});
```

4.8.2. Получение информации о событии

Получить информацию о нажатых клавишах позволяют следующие методы из класса `KeyEvent`:

- ❑ `getCode()` — возвращает константу из перечисления `KeyCode`, которая соответствует нажатой клавише. Внутри обработчика события `KEY_TYPED` метод возвращает значение `KeyCode.UNDEFINED`. Формат метода:

```
import javafx.scene.input.KeyCode;
public final KeyCode getCode()
```

Пример обработки нажатия клавиши <F1>:

```
stage.addEventHandler(KeyEvent.KEY_PRESSED, event -> {
    if (event.getCode() == KeyCode.F1) {
        System.out.println("Нажата клавиша <F1>");
    }
});
```

- ❑ `getText()` — возвращает текстовое представление нажатой клавиши. Внутри обработчика события `KEY_TYPED` метод возвращает пустую строку. Формат метода:

```
public final String getText()
```

Пример определения нажатия клавиши <ы>:

```
stage.addEventHandler(KeyEvent.KEY_PRESSED, event -> {
    if (event.getText().equals("ы")) {
        System.out.println("Нажата клавиша <ы>");
    }
});
```

- ❑ `getCharacter()` — внутри обработчика события `KEY_TYPED` возвращает введенный символ. Внутри обработчиков событий `KEY_PRESSED` и `KEY_RELEASED` возвращает значение `KeyEvent.CHAR_UNDEFINED` (содержит нулевой символ). Формат метода:

```
public final String getCharacter()
```

Метод учитывает нажатие клавиш <Shift> и <Caps Lock>, а также язык раскладки клавиатуры, поэтому мы всегда получим введенный символ с учетом регистра и языка без необходимости выполнять дополнительные проверки. Пример обработки ввода буквы «я» в нижнем регистре:

```
stage.addEventHandler(KeyEvent.KEY_TYPED, event -> {
    if (event.getCharacter().equals("я")) {
        System.out.println("Введена буква 'я'");
    }
});
```

- ❑ `isShiftDown()` — возвращает значение `true`, если удерживалась нажатой клавиша `<Shift>`, и `false` — в противном случае. Формат метода:

```
public final boolean isShiftDown()
```

- ❑ `isControlDown()` — возвращает значение `true`, если удерживалась нажатой клавиша `<Ctrl>`, и `false` — в противном случае. Формат метода:

```
public final boolean isControlDown()
```

- ❑ `isAltDown()` — возвращает значение `true`, если удерживалась нажатой клавиша `<Alt>`, и `false` — в противном случае. Формат метода:

```
public final boolean isAltDown()
```

- ❑ `isMetaDown()` — возвращает значение `true`, если удерживалась нажатой клавиша `<Meta>` (в Mac), и `false` — в противном случае. Формат метода:

```
public final boolean isMetaDown()
```

- ❑ `isShortcutDown()` — возвращает значение `true`, если удерживалась нажатой клавиша `<Ctrl>` в Windows или `<Meta>` в Mac, и `false` — в противном случае. Формат метода:

```
public final boolean isShortcutDown()
```

4.8.3. Коды клавиш

Приведем наиболее часто используемые константы из перечисления `KeyCode`:

```
public static final KeyCode A           // A
public static final KeyCode B           // B
public static final KeyCode C           // C
public static final KeyCode D           // D
public static final KeyCode E           // E
public static final KeyCode F           // F
public static final KeyCode G           // G
public static final KeyCode H           // H
public static final KeyCode I           // I
public static final KeyCode J           // J
public static final KeyCode K           // K
public static final KeyCode L           // L
public static final KeyCode M           // M
public static final KeyCode N           // N
public static final KeyCode O           // O
public static final KeyCode P           // P
public static final KeyCode Q           // Q
public static final KeyCode R           // R
public static final KeyCode S           // S
public static final KeyCode T           // T
public static final KeyCode U           // U
public static final KeyCode V           // V
```



```

public static final KeyCode W           // W
public static final KeyCode X           // X
public static final KeyCode Y           // Y
public static final KeyCode Z           // Z
// Цифровые клавиши в основной части клавиатуры
public static final KeyCode DIGIT1      // 1
public static final KeyCode DIGIT2      // 2
public static final KeyCode DIGIT3      // 3
public static final KeyCode DIGIT4      // 4
public static final KeyCode DIGIT5      // 5
public static final KeyCode DIGIT6      // 6
public static final KeyCode DIGIT7      // 7
public static final KeyCode DIGIT8      // 8
public static final KeyCode DIGIT9      // 9
public static final KeyCode DIGIT0      // 0
// Цифровые клавиши в дополнительной части клавиатуры
public static final KeyCode NUMPAD1      // 1
public static final KeyCode NUMPAD2      // 2
public static final KeyCode NUMPAD3      // 3
public static final KeyCode NUMPAD4      // 4
public static final KeyCode NUMPAD5      // 5
public static final KeyCode NUMPAD6      // 6
public static final KeyCode NUMPAD7      // 7
public static final KeyCode NUMPAD8      // 8
public static final KeyCode NUMPAD9      // 9
public static final KeyCode NUMPAD0      // 0

public static final KeyCode SPACE        // <Пробел>
public static final KeyCode OPEN_BRACKET // [
public static final KeyCode CLOSE_BRACKET // ]
public static final KeyCode SEMICOLON    // ;
public static final KeyCode QUOTE         // одинарная кавычка
public static final KeyCode COMMA         // запятая
public static final KeyCode PERIOD        // точка
public static final KeyCode SLASH         // слеш /
public static final KeyCode BACK_SLASH    // обратный слеш \
public static final KeyCode BACK_QUOTE    // обратная кавычка `
public static final KeyCode MINUS         // минус
public static final KeyCode EQUALS         // =
public static final KeyCode MULTIPLY      // *
public static final KeyCode SUBTRACT      // -
public static final KeyCode ADD           // +

public static final KeyCode SHIFT         // <Shift>
public static final KeyCode CONTROL       // <Ctrl>
public static final KeyCode ALT           // <Alt>

```

```
public static final KeyCode CAPS           // <Caps Lock>
public static final KeyCode TAB            // <Tab>
public static final KeyCode ESCAPE        // <Esc>
public static final KeyCode ENTER         // <Enter>
public static final KeyCode BACK_SPACE    // <Backspace>
public static final KeyCode DELETE        // <Delete>
public static final KeyCode NUM_LOCK      // <NumLock>
public static final KeyCode HOME          // <Home>
public static final KeyCode END           // <End>
public static final KeyCode PAGE_UP       // <PgUp>
public static final KeyCode PAGE_DOWN     // <PgDn>
public static final KeyCode UP            // вверх
public static final KeyCode RIGHT         // вправо
public static final KeyCode DOWN          // вниз
public static final KeyCode LEFT          // влево
public static final KeyCode F1            // <F1>
// ...
public static final KeyCode F12           // <F12>
```

При проверке кода клавиши, возвращаемого методом `getCode()`, важно учитывать, что на клавиатуре одной клавише может быть сопоставлено несколько значений. Эти значения меняются в зависимости от раскладки клавиатуры, удержания клавиши `<Shift>`, нажатия кнопки `<Caps Lock>` и т. д. Например, метод `getCode()` возвращает значение `KeyCode.T`. Если выбрана русская раскладка клавиатуры, то на самом деле пользователь вводит русскую букву «Е», а если удерживается нажатой клавиша `<Shift>`, то важно еще учитывать и регистр буквы. При вводе букв и символов лучше воспользоваться методом `getCharacter()` внутри обработчика события `KEY_TYPED` или выполнять дополнительную проверку.

Пример обработки нажатия клавиши `<Tab>`:

```
stage.addEventFilter(KeyEvent.KEY_PRESSED, event -> {
    if (event.getCode() == KeyCode.TAB) {
        System.out.println("Нажата клавиша <Tab>");
    }
});
```

4.8.4. Управление фокусом ввода

При обработке нажатия клавиш следует учитывать, что компонент должен иметь возможность принимать фокус ввода и находиться в фокусе ввода. Некоторые компоненты не могут принимать фокус ввода по умолчанию, например надпись. Если невозможно установить фокус ввода на компонент, то обработчик события ввода нужно назначить через объект сцены или объект окна.

ПРИМЕЧАНИЕ

Управление фокусом ввода окна мы уже рассматривали в разд. 2.6.

В один момент времени только один компонент (или вообще ни одного) имеет фокус ввода. Пользователь может установить фокус ввода с помощью мыши, щелкнув левой кнопкой мыши на компоненте, или путем нажатия клавиши <Tab> (передача фокуса следующему компоненту) или комбинации <Shift>+<Tab> (передача фокуса предыдущему компоненту). Нажатие компонента, находящегося в фокусе ввода (например, кнопки), осуществляется с помощью клавиши <Пробел>.

Управлять фокусом ввода из программы позволяют следующие методы из класса `Node`:

- ❑ `requestFocus()` — устанавливает фокус ввода. Формат метода:

```
public void requestFocus()
```

При нажатии клавиши <Enter> внутри текстового поля установим фокус ввода на кнопку:

```
text.setOnKeyPressed(event -> {
    if (event.getCode() == KeyCode.ENTER) {
        button1.requestFocus();
    }
});
```

- ❑ `isFocused()` — возвращает `true`, если компонент имеет фокус ввода, и `false` — в противном случае. Формат метода:

```
public final boolean isFocused()
```

- ❑ `focusedProperty()` — возвращает ссылку на свойство, связанное с фокусом ввода. Формат метода:

```
public final ReadOnlyBooleanProperty focusedProperty()
```

- ❑ `isFocusTraversable()` — возвращает `true`, если компонент может принимать фокус ввода, и `false` — в противном случае. Формат метода:

```
public final boolean isFocusTraversable()
```

- ❑ `setFocusTraversable()` — разрешает (значение `true`) или запрещает (значение `false`) возможность принятия фокуса ввода. Формат метода:

```
public final void setFocusTraversable(boolean value)
```

Разрешим надписи получать фокус ввода:

```
Label label = new Label("Текстовая надпись");
label.setFocusTraversable(true);
```

- ❑ `focusTraversableProperty()` — возвращает ссылку на свойство `focusTraversable`. Формат метода:

```
public final BooleanProperty focusTraversableProperty()
```

Получить ссылку на узел, находящийся в фокусе ввода, позволяет метод `getFocusOwner()` из класса `Scene`. Формат метода:

```
public final Node getFocusOwner()
```

Отслеживать изменение фокуса можно с помощью свойства `focusOwner`. Получить ссылку на свойство позволяет метод `focusOwnerProperty()` из класса `Scene`. Формат метода:

```
public final ReadOnlyObjectProperty<Node> focusOwnerProperty()
```

Пример:

```
scene.focusOwnerProperty().addListener((obj, oldValue, newValue) -> {  
    System.out.println("Изменение focusOwner: " + newValue);  
    System.out.println(scene.getFocusOwner());  
});
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH04_08_focused`, приведенном в каталоге `CH04_events` сопровождающего книгу электронного архива (см. приложение).

4.8.5. Назначение клавиш быстрого доступа

Клавиши быстрого доступа (иногда их также называют «горячими» клавишами) позволяют установить фокус ввода с помощью нажатия специальной клавиши (например, `<Alt>` или `<Ctrl>`) и какой-либо дополнительной клавиши. Если после нажатия клавиш быстрого доступа в фокусе окажется кнопка (или пункт меню), то она будет нажата.

Чтобы задать клавишу быстрого доступа, следует в тексте надписи указать символ `_` (подчеркивание) перед буквой. В этом случае буква, перед которой указан символ `_`, будет подчеркнута, что станет подсказкой пользователю. При одновременном нажатии клавиши `<Alt>` и подчеркнутой буквы компонент окажется в фокусе ввода. Управлять разбором текста надписи позволяют следующие методы из класса `Labeled` (такие же методы есть и в классе `MenuItem`):

- ❑ `setMnemonicParsing()` — разрешает (значение `true`) или запрещает (значение `false`) разбор текста надписи. Формат метода:

```
public final void setMnemonicParsing(boolean value)
```

- ❑ `isMnemonicParsing()` — возвращает `true`, если разрешен разбор текста надписи, и `false` — в противном случае. Формат метода:

```
public final boolean isMnemonicParsing()
```

- ❑ `mnemonicParsingProperty()` — возвращает ссылку на свойство `mnemonicParsing`. Формат метода:

```
public final BooleanProperty mnemonicParsingProperty()
```

Пример назначения клавиши быстрого доступа для кнопки:

```
Button button = new Button("_Button");  
button.setMnemonicParsing(true);
```

При нажатии комбинации клавиш <Alt>+ кнопка окажется в фокусе ввода и будет нажата. В этом примере вторую инструкцию можно и не указывать явным образом, т. к. разбор надписи для кнопок разрешен по умолчанию.

Настройка разбора текста надписи в программе Scene Builder осуществляется на вкладке **Properties** в разделе **Extras** (флажок **Mnemonic Parsing**). Пример указания атрибута `mnemonicParsing` в FXML-файле:

```
<Label mnemonicParsing="true" text="Label" textFill="#c62d2d" />
```

Некоторые компоненты (например, текстовое поле) не имеют надписи. Чтобы задать клавиши быстрого доступа для таких компонентов, необходимо отдельно создать надпись и связать ее с компонентом с помощью метода `setLabelFor()` из класса `Label`. Управлять связыванием позволяют следующие методы:

```
public final void setLabelFor(Node value)
public final Node getLabelFor()
public ObjectProperty<Node> labelForProperty()
```

Пример связывания надписи с однострочным текстовым полем:

```
Label labelForText = new Label("Tex_t:");
TextField text = new TextField();
labelForText.setMnemonicParsing(true);
labelForText.setLabelFor(text);
```

Назначить комбинацию клавиш быстрого доступа можно с помощью метода `addMnemonic()` из класса `Scene`. Формат метода:

```
public void addMnemonic(Mnemonic m)
```

В качестве параметра метод принимает экземпляр класса `Mnemonic`. Формат конструктора класса:

```
// Модуль javafx.graphics
import javafx.scene.input.Mnemonic;
Mnemonic(Node node, KeyCombination keyCombination)
```

Пример назначения клавиш быстрого доступа для кнопки:

```
// <Alt>+<P>
scene.addMnemonic(new Mnemonic(button3,
    KeyCombination.keyCombination("Alt+'P'")));
// <Shift>+<Alt>+<W>
scene.addMnemonic(new Mnemonic(button3,
    new KeyCodeCombination(KeyCode.W,
    KeyCombination.ALT_DOWN, KeyCombination.SHIFT_DOWN)));
```

Отменить регистрацию позволяет метод `removeMnemonic()` из класса `Scene`. Формат метода:

```
public void removeMnemonic(Mnemonic m)
```

Получить перечень всех комбинаций клавиш быстрого доступа позволяет метод `getMnemonics()` из класса `Scene`. Формат метода:

```
import javafx.collections.ObservableList;
import javafx.collections.ObservableMap;
public ObservableMap<KeyCombination, ObservableList<Mnemonic>>
    getMnemonics()
```

Пример:

```
ObservableMap<KeyCombination, ObservableList<Mnemonic>> map =
    scene.getMnemonics();
for (KeyCombination key: map.keySet()) {
    System.out.println(key.toString() + ":");
    for (Mnemonic m: map.get(key)) {
        System.out.println("---" + m.getNode().getClass());
    }
}
```

В классе `MenuItem` дополнительно существуют следующие методы для управления клавишами-ускорителями:

```
public final void setAccelerator(KeyCombination value)
public final KeyCombination getAccelerator()
public final ObjectProperty<KeyCombination> acceleratorProperty()
```

Метод `setAccelerator()` позволяет назначить комбинацию клавиш для пункта меню. Более подробно работу с меню мы рассмотрим в *разд. 15.3.3*, а сейчас просто приведем пример назначения комбинации клавиш-ускорителей:

```
MenuItem openMenuItem = new MenuItem("Open...");
openMenuItem.setAccelerator(new KeyCodeCombination(KeyCode.O,
    KeyCombination.CONTROL_DOWN));
```

Получить перечень всех комбинаций клавиш-ускорителей позволяет метод `getAccelerators()` из класса `Scene`. Формат метода:

```
public ObservableMap<KeyCombination, Runnable> getAccelerators()
```

Пример:

```
ObservableMap<KeyCombination, Runnable> map2 = scene.getAccelerators();
for (KeyCombination key: map2.keySet()) {
    System.out.println(key.toString());
}
```

ЭЛЕКТРОННЫЙ АРХИВ

Пример назначения клавиш быстрого доступа содержится в проекте `FX_CH04_08_KeyCombination`, приведенном в каталоге `CH04_events` сопровождающего книгу электронного архива (см. *приложение*).

4.9. События мыши

События мыши обрабатываются не реже, чем события клавиатуры. Мы можем обработать нажатие и отпускание кнопки мыши, перемещение указателя, а также вхождение указателя в область компонента и выхода из этой области. В зависимо-

сти от ситуации можно изменить вид указателя — например, при выполнении длительной операции отобразить указатель ожидания.

События мыши описываются с помощью класса `MouseEvent`. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.input.MouseEvent;
```

Иерархия наследования:

`Object` — `EventObject` — `Event` — `InputEvent` — `MouseEvent`

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH04_09_MouseEvent`, приведенном в каталоге `CH04_events` сопровождающего книгу электронного архива (см. приложение).

4.9.1. Типы событий

В классе `MouseEvent` определены следующие типы событий:

- `ANY` — любое из приведенных далее событий, а также события класса `MouseEvent`. Формат:

```
public static final EventType<MouseEvent> ANY
```

Пример обработки всех событий (кроме `MOUSE_ENTERED` и `MOUSE_EXITED`) мыши в окне:

```
stage.addEventHandler(MouseEvent.ANY, event -> {
    System.out.println("ANY - " + event.getEventType());
});
```

- `MOUSE_PRESSED` — нажатие кнопки мыши. Формат:

```
public static final EventType<MouseEvent> MOUSE_PRESSED
```

Управлять обработчиком события позволяют также следующие методы из классов `Node` и `Scene`:

```
public final void setOnMousePressed(
    EventHandler<? super MouseEvent> value)
public final EventHandler<? super MouseEvent>
    getOnMousePressed()
public final ObjectProperty<EventHandler<? super MouseEvent>>
    onMousePressedProperty()
```

- `MOUSE_RELEASED` — отпускание ранее нажатой кнопки мыши. Формат:

```
public static final EventType<MouseEvent> MOUSE_RELEASED
```

Управлять обработчиком события позволяют также следующие методы из классов `Node` и `Scene`:

```
public final void setOnMouseReleased(
    EventHandler<? super MouseEvent> value)
```

```
public final EventHandler<? super MouseEvent>
    getOnMouseReleased()
public final ObjectProperty<EventHandler<? super MouseEvent>>
    onMouseReleasedProperty()
```

- ❑ **MOUSE_CLICKED** — щелчок мышью (нажатие и отпускание кнопки мыши). Формат:

```
public static final EventType<MouseEvent> MOUSE_CLICKED
```

Последовательность событий при щелчке:

```
MOUSE_PRESSED
MOUSE_RELEASED
MOUSE_CLICKED
```

Управлять обработчиком события позволяют также следующие методы из классов **Node** и **Scene**:

```
public final void setOnMouseClicked(
    EventHandler<? super MouseEvent> value)
public final EventHandler<? super MouseEvent>
    getOnMouseClicked()
public final ObjectProperty<EventHandler<? super MouseEvent>>
    onMouseClickedProperty()
```

- ❑ **MOUSE_MOVED** — перемещение указателя без нажатых кнопок. Формат:

```
public static final EventType<MouseEvent> MOUSE_MOVED
```

Управлять обработчиком события позволяют также следующие методы из классов **Node** и **Scene**:

```
public final void setOnMouseMoved(
    EventHandler<? super MouseEvent> value)
public final EventHandler<? super MouseEvent>
    getOnMouseMoved()
public final ObjectProperty<EventHandler<? super MouseEvent>>
    onMouseMovedProperty()
```

- ❑ **MOUSE_DRAGGED** — перемещение указателя с нажатой кнопкой. Формат:

```
public static final EventType<MouseEvent> MOUSE_DRAGGED
```

Управлять обработчиком события позволяют также следующие методы из классов **Node** и **Scene**:

```
public final void setOnMouseDragged(
    EventHandler<? super MouseEvent> value)
public final EventHandler<? super MouseEvent>
    getOnMouseDragged()
public final ObjectProperty<EventHandler<? super MouseEvent>>
    onMouseDraggedProperty()
```

- ❑ **DRAG_DETECTED** — начало процесса перетаскивания. Формат:

```
public static final EventType<MouseEvent> DRAG_DETECTED
```


Управлять обработчиком события позволяют также следующие методы из классов Node и Scene:

```
public final void setOnDragDetected(
    EventHandler<? super MouseEvent> value)
public final EventHandler<? super MouseEvent>
    getOnDragDetected()
public final ObjectProperty<EventHandler<? super MouseEvent>>
    onDragDetectedProperty()
```

- ❑ **MOUSE_ENTERED** — наведение указателя мыши на узел (событие доставляется только узлу). Формат:

```
public static final EventType<MouseEvent> MOUSE_ENTERED
```

Управлять обработчиком события позволяют также следующие методы из классов Node и Scene:

```
public final void setOnMouseEntered(
    EventHandler<? super MouseEvent> value)
public final EventHandler<? super MouseEvent>
    getOnMouseEntered()
public final ObjectProperty<EventHandler<? super MouseEvent>>
    onMouseEnteredProperty()
```

- ❑ **MOUSE_EXITED** — выведение указателя мыши (событие доставляется только узлу). Формат:

```
public static final EventType<MouseEvent> MOUSE_EXITED
```

Пример изменения цвета прямоугольника при наведении и выведении указателя мыши:

```
rect.addEventHandler(MouseEvent.MOUSE_ENTERED, event -> {
    rect.setFill(Color.RED);
});
rect.addEventHandler(MouseEvent.MOUSE_EXITED, event -> {
    rect.setFill(Color.CHOCOLATE);
});
```

Управлять обработчиком события позволяют также следующие методы из классов Node и Scene:

```
public final void setOnMouseExited(
    EventHandler<? super MouseEvent> value)
public final EventHandler<? super MouseEvent>
    getOnMouseExited()
public final ObjectProperty<EventHandler<? super MouseEvent>>
    onMouseExitedProperty()
```

- ❑ **MOUSE_ENTERED_TARGET** — наведение указателя мыши. Чтобы определить какой узел является целью события, следует вызвать метод `getTarget()` через объект события. Формат:

```
public static final EventType<MouseEvent> MOUSE_ENTERED_TARGET
```

❑ `MOUSE_EXITED_TARGET` — выводение указателя мыши. Чтобы определить какой узел является целью события, следует вызвать метод `getTarget()` через объект события. Формат:

```
public static final EventType<MouseEvent> MOUSE_EXITED_TARGET
```

Чем же отличаются события `MOUSE_ENTERED` и `MOUSE_EXITED` от `MOUSE_ENTERED_TARGET` и `MOUSE_EXITED_TARGET`? Предположим, на сцену мы добавили контейнер, а в него прямоугольник. Если мы назначим обработчик события `MOUSE_ENTERED` для объекта сцены и наведем указатель мыши на прямоугольник, то получим событие вхождения указателя мыши только внутрь сцены. Если же назначим обработчик события `MOUSE_ENTERED_TARGET` для объекта сцены, то последовательно получим три события вхождения: внутрь сцены, внутрь контейнера и внутрь прямоугольника. Куда именно произведено вхождение, можно определить с помощью метода `getTarget()` объекта события:

```
scene.addEventHandler(MouseEvent.MOUSE_ENTERED_TARGET, event -> {
    System.out.println("getTarget(): " + event.getTarget());
});
```

Назначить обработчик для события наведения указателя мыши на узел и выводения указателя мыши позволяет свойство `hover` из класса `Node`. Методы доступа:

```
public final boolean isHover()
public final ReadOnlyBooleanProperty hoverProperty()
```

Пример назначения обработчика:

```
Rectangle rect = new Rectangle(100.0, 100.0);
rect.hoverProperty().addListener((obj, oldValue, newValue) -> {
    System.out.println("Изменение hover " + newValue);
});
```

Обработать нажатие и отпускание кнопки мыши над узлом можно также с помощью свойства `pressed` из класса `Node`. Методы доступа:

```
public final boolean isPressed()
public final ReadOnlyBooleanProperty pressedProperty()
```

Пример назначения обработчика:

```
Rectangle rect = new Rectangle(100.0, 100.0);
rect.pressedProperty().addListener((obj, oldValue, newValue) -> {
    System.out.println("Изменение pressed " + newValue);
});
```

Если мы хотим, чтобы узел не принимал события мыши, то можно вызвать метод `setMouseTransparent()` из класса `Node` и передать ему значение `true`. Значение `false` отменяет запрет. Формат метода:

```
public final void setMouseTransparent(boolean value)
```

Пример:

```
Rectangle rect = new Rectangle(50.0, 50.0);
rect.setMouseTransparent(true);
```

Проверить, принимает ли узел события мыши, позволяют методы `isMouseTransparent()` и `mouseTransparentProperty()` из класса `Node`. Форматы методов:

```
public final boolean isMouseTransparent()
public final BooleanProperty mouseTransparentProperty()
```

4.9.2. Получение информации о событии

Получить информацию о событии позволяют следующие основные методы из класса `MouseEvent` (полный список методов смотрите в документации):

- ❑ `getButton()` — возвращает информацию о нажатой кнопке в виде перечисления `MouseButton`, содержащего константы: `NONE`, `PRIMARY` (левая), `MIDDLE` (средняя) и `SECONDARY` (правая). Формат метода:

```
public final MouseButton getButton()
```

- ❑ `isPrimaryButtonDown()` — возвращает значение `true`, если была нажата левая кнопка мыши, и `false` — в противном случае. Формат метода:

```
public final boolean isPrimaryButtonDown()
```

- ❑ `isMiddleButtonDown()` — возвращает значение `true`, если была нажата средняя кнопка мыши, и `false` — в противном случае. Формат метода:

```
public final boolean isMiddleButtonDown()
```

- ❑ `isSecondaryButtonDown()` — возвращает значение `true`, если была нажата правая кнопка мыши, и `false` — в противном случае. Формат метода:

```
public final boolean isSecondaryButtonDown()
```

- ❑ `getClickCount()` — возвращает количество щелчков мышью. Формат метода:

```
public final int getClickCount()
```

Пример обработки двойного щелчка левой кнопкой мыши:

```
scene.addEventHandler(MouseEvent.MOUSE_PRESSED, event -> {
    if (event.isPrimaryButtonDown() &&
        event.getClickCount() == 2) {
        System.out.println("Двойной щелчок левой кнопкой");
    }
});
```

- ❑ `getX()`, `getY()` и `getZ()` — возвращают координаты указателя относительно узла. Например, если обрабатываем перемещение указателя внутри прямоугольника, то получим координаты относительно левого верхнего угла прямоугольника. Форматы методов:

```
public final double getX()
public final double getY()
public final double getZ()
```

- ❑ `getSceneX()` и `getSceneY()` — возвращают координаты указателя относительно сцены. Форматы методов:

```
public final double getSceneX()
public final double getSceneY()
```

- ❑ `getScreenX()` и `getScreenY()` — возвращают координаты указателя относительно левого верхнего угла экрана. Форматы методов:

```
public final double getScreenX()
public final double getScreenY()
```

- ❑ `isShiftDown()` — возвращает значение `true`, если удерживалась нажатой клавиша `<Shift>`, и `false` — в противном случае. Формат метода:

```
public final boolean isShiftDown()
```

- ❑ `isControlDown()` — возвращает значение `true`, если удерживалась нажатой клавиша `<Ctrl>`, и `false` — в противном случае. Формат метода:

```
public final boolean isControlDown()
```

- ❑ `isAltDown()` — возвращает значение `true`, если удерживалась нажатой клавиша `<Alt>`, и `false` — в противном случае. Формат метода:

```
public final boolean isAltDown()
```

- ❑ `isMetaDown()` — возвращает значение `true`, если удерживалась нажатой клавиша `<Meta>` (в Mac), и `false` — в противном случае. Формат метода:

```
public final boolean isMetaDown()
```

- ❑ `isShortcutDown()` — возвращает значение `true`, если удерживалась нажатой клавиша `<Ctrl>` в Windows или `<Meta>` в Mac, и `false` — в противном случае. Формат метода:

```
public final boolean isShortcutDown()
```

- ❑ `isDragDetect()` — возвращает значение `true` в начале процесса перетаскивания (возникло событие `DRAG_DETECTED`), и `false` — в противном случае. Формат метода:

```
public boolean isDragDetect()
```

- ❑ `isStillSincePress()` — возвращает значение `true`, если указатель мыши не двигался между событиями, и `false` — в противном случае. Формат метода:

```
public final boolean isStillSincePress()
```

- ❑ `isPopupTrigger()` — возвращает значение `true`, если событие привело к открытию всплывающего меню, и `false` — в противном случае. Формат метода:

```
public final boolean isPopupTrigger()
```

- ❑ `isSynthesized()` — возвращает значение `true`, если событие сгенерировано сенсорным экраном, и `false` — в противном случае. Формат метода:

```
public boolean isSynthesized()
```

4.9.3. Прокрутка колесика мыши

Многие мыши комплектуются колесиком, которое обычно используется для управления прокруткой некоторой области. Обработать поворот этого колесика, а также жесты прокрутки сенсорного экрана позволяет класс `ScrollEvent`. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.input.ScrollEvent;
```

Иерархия наследования:

`Object` — `EventObject` — `Event` — `InputEvent` — `GestureEvent` — `ScrollEvent`

В классе `ScrollEvent` определены следующие типы событий:

□ **ANY** — любое из приведенных далее значений. Формат:

```
public static final EventType<ScrollEvent> ANY
```

□ **SCROLL** — событие прокрутки. Формат:

```
public static final EventType<ScrollEvent> SCROLL
```

Пример назначения обработчика события:

```
scene.addEventHandler(ScrollEvent.SCROLL, event -> {
    System.out.println("SCROLL");
});
```

Управлять обработчиком события позволяют также следующие методы из классов `Node` и `Scene`:

```
public final void setOnScroll(
    EventHandler<? super ScrollEvent> value)
public final EventHandler<? super ScrollEvent> getOnScroll()
public final ObjectProperty<EventHandler<? super ScrollEvent>>
    onScrollProperty()
```

□ **SCROLL_STARTED** — начало жеста прокрутки (при использовании колесика мыши событие не генерируется). Формат:

```
public static final EventType<ScrollEvent> SCROLL_STARTED
```

Управлять обработчиком события позволяют также следующие методы из классов `Node` и `Scene`:

```
public final void setOnScrollStarted(
    EventHandler<? super ScrollEvent> value)
public final EventHandler<? super ScrollEvent>
    getOnScrollStarted()
public final ObjectProperty<EventHandler<? super ScrollEvent>>
    onScrollStartedProperty()
```

□ **SCROLL_FINISHED** — конец жеста прокрутки (при использовании колесика мыши событие не генерируется). Формат:

```
public static final EventType<ScrollEvent> SCROLL_FINISHED
```

Управлять обработчиком события позволяют также следующие методы из классов Node и Scene:

```
public final void setOnScrollFinished(
    EventHandler<? super ScrollEvent> value)
public final EventHandler<? super ScrollEvent>
    getOnScrollFinished()
public final ObjectProperty<EventHandler<? super ScrollEvent>>
    onScrollFinishedProperty()
```

Класс `ScrollEvent` наследует методы: `getX()`, `getY()`, `getZ()`, `getSceneX()`, `getSceneY()`, `getScreenX()`, `getScreenY()`, `isShiftDown()`, `isControlDown()`, `isAltDown()`, `isMetaDown()`, `isShortcutDown()` из класса `GestureEvent`. Предназначение этих методов точно такое же, что и у одноименных методов из класса `MouseEvent` (см. разд. 4.9.2).

Получить информацию о прокрутке колесика мыши позволяют следующие методы из класса `ScrollEvent` (полный список методов смотрите в документации):

- ❑ `getDeltaX()` — возвращает количество пикселей для прокрутки по горизонтали. Чтобы получить горизонтальное направление, нужно при повороте колесика удерживать нажатой клавишу `<Shift>`. Значение может быть положительным или отрицательным в зависимости от направления поворота. Формат метода:

```
public double getDeltaX()
```

- ❑ `getDeltaY()` — возвращает количество пикселей для прокрутки по вертикали. Значение может быть положительным или отрицательным в зависимости от направления поворота. Формат метода:

```
public double getDeltaY()
```

Пример изменения местоположения прямоугольника с помощью поворота колесика мыши:

```
scene.addEventHandler(ScrollEvent.SCROLL, event -> {
    rect.setTranslateX(rect.getTranslateX() + event.getDeltaX());
    rect.setTranslateY(rect.getTranslateY() + event.getDeltaY());
});
```

- ❑ `getMultiplierX()` и `getMultiplierY()` — возвращают множитель, использованный для преобразования единиц вращения колесика мыши в пиксели. Форматы методов:

```
public double getMultiplierX()
public double getMultiplierY()
```

Пример вывода текущего значения:

```
System.out.println(event.getMultiplierY()); // 40.0
```

ЭЛЕКТРОННЫЙ АРХИВ

Пример обработки прокрутки колесика мыши содержится в проекте `FX_CH04_09_ScrollEvent`, приведенном в каталоге `CH04_events` сопровождающего книгу электронного архива (см. приложение).

4.9.4. Изменение внешнего вида указателя мыши

Для изменения внешнего вида указателя мыши при вхождении его в область узла предназначены следующие методы из класса `Node`:

- `setCursor()` — задает внешний вид указателя мыши для узла. Формат метода:

```
import javafx.scene.Cursor;
public final void setCursor(Cursor value)
```

В качестве параметра указывается экземпляр классов `Cursor` или `ImageCursor` (пользовательский курсор произвольной формы) или следующие статические константы из класса `Cursor`: `NONE` (указатель не отображается), `DEFAULT` (указатель по умолчанию), `HAND` (указатель в виде руки с указывающим пальцем), `CLOSED_HAND` (сжатая рука), `OPEN_HAND` (разжатая рука), `WAIT` (курсор ожидания), `TEXT` (текстовый I-образный указатель), `CROSSHAIR` (крестообразный указатель), `DISAPPEAR`, `MOVE` (стрелки, направленные вверх, вниз, влево и вправо), `E_RESIZE`, `H_RESIZE` (стрелки, направленные влево и вправо), `N_RESIZE`, `NE_RESIZE`, `NW_RESIZE`, `S_RESIZE`, `SE_RESIZE`, `SW_RESIZE`, `V_RESIZE` (стрелки, направленные вверх и вниз), `W_RESIZE`. Пример назначения курсора для прямоугольника:

```
Rectangle rect = new Rectangle(100.0, 100.0);
rect.setFill(Color.CHOCOLATE);
rect.setCursor(Cursor.OPEN_HAND);
```

Чтобы вернуть вид указателя по умолчанию, нужно вместо объекта передать в метод `setCursor()` значение `null`:

```
rect.setCursor(null);
```

- `getCursor()` — возвращает текущий курсор или значение `null`. Формат метода:

```
public final Cursor getCursor()
```

- `cursorProperty()` — возвращает ссылку на свойство. Формат метода:

```
public final ObjectProperty<Cursor> cursorProperty()
```

При использовании программы `Scene Builder` внешний вид указателя мыши задается на вкладке **Properties** в разделе **Node** (список **Cursor**). Пример описания внешнего вида указателя мыши в `FXML`-файле:

```
<Label text="Label">
  <cursor>
    <Cursor fx:constant="CROSSHAIR" />
  </cursor>
</Label>
```

Управлять видом указателя мыши для сцены позволяют методы `setCursor()`, `getCursor()` и `cursorProperty()` из класса `Scene`. Форматы методов:

```
public final void setCursor(Cursor value)
public final Cursor getCursor()
public final ObjectProperty<Cursor> cursorProperty()
```

ЭЛЕКТРОННЫЙ АРХИВ

Пример изменения указателя мыши содержится в проекте `FX_CH04_09_Cursor`, приведенном в каталоге `CH04_events` сопровождающего книгу электронного архива (см. *приложение*).

4.10. Работа с буфером обмена

Для обмена данными между приложениями используется *буфер обмена*. Одно приложение помещает данные в буфер обмена, а второе приложение (или то же самое) может извлечь их оттуда. Получить ссылку на глобальный объект буфера обмена позволяет статический метод `getSystemClipboard()` из класса `Clipboard`. Формат метода:

```
// Модуль javafx.graphics
import javafx.scene.input.Clipboard;
public static Clipboard getSystemClipboard()
```

ЭЛЕКТРОННЫЙ АРХИВ

Пример работы с буфером обмена содержится в проекте `FX_CH04_10_Clipboard`, приведенном в каталоге `CH04_events` сопровождающего книгу электронного архива (см. *приложение*).

4.10.1. Класс *Clipboard*

Управлять содержимым буфера обмена позволяют следующие методы из класса `Clipboard`:

- `setContent()` — добавляет данные в буфер обмена. Формат метода:

```
public final boolean setContent(
    Map<DataFormat, Object> content)
```

Пример добавления простого текста в буфер обмена:

```
Clipboard cb = Clipboard.getSystemClipboard();
ClipboardContent content = new ClipboardContent();
content.putString("Текст, добавляемый в буфер обмена");
cb.setContent(content);
```

- `clear()` — очищает буфер обмена. Формат метода:

```
public final void clear()
```

- `hasString()` — возвращает значение `true`, если в буфере содержатся данные, имеющие тип `DataFormat.PLAIN_TEXT` (простой текст), и `false` — в противном случае. Формат метода:

```
public final boolean hasString()
```

- `getString()` — позволяет получить данные, ассоциированные с типом `DataFormat.PLAIN_TEXT`. Формат метода:

```
public final String getString()
```


Пример получения простого текста:

```
Clipboard cb = Clipboard.getSystemClipboard();
if (cb.hasString()) {
    System.out.println(cb.getString());
}
```

- ❑ **hasHtml()** — возвращает значение `true`, если в буфере содержатся данные, имеющие тип `DataFormat.HTML` (HTML-текст), и `false` — в противном случае. **Формат метода:**

```
public final boolean hasHtml()
```

- ❑ **getHtml()** — позволяет получить данные, ассоциированные с типом `DataFormat.HTML`. **Формат метода:**

```
public final String getHtml()
```

- ❑ **hasRtf()** — возвращает значение `true`, если в буфере содержатся данные, имеющие тип `DataFormat.RTF` (текст в формате RTF), и `false` — в противном случае. **Формат метода:**

```
public final boolean hasRtf()
```

- ❑ **getRtf()** — позволяет получить данные, ассоциированные с типом `DataFormat.RTF`. **Формат метода:**

```
public final String getRtf()
```

- ❑ **hasUrl()** — возвращает значение `true`, если в буфере содержатся данные, имеющие тип `DataFormat.URL` (URL-адрес), и `false` — в противном случае. **Формат метода:**

```
public final boolean hasUrl()
```

- ❑ **getUrl()** — позволяет получить данные, ассоциированные с типом `DataFormat.URL`. **Обратите внимание:** метод может возвращать сразу несколько URL-адресов через символы-разделители строк. **Формат метода:**

```
public final String getUrl()
```

- ❑ **hasFiles()** — возвращает значение `true`, если в буфере содержатся данные, имеющие тип `DataFormat.FILES` (список файлов), и `false` — в противном случае. **Формат метода:**

```
public final boolean hasFiles()
```

- ❑ **getFiles()** — позволяет получить данные, ассоциированные с типом `DataFormat.FILES`. **Метод возвращает список объектов класса `File`. Формат метода:**

```
public final List<File> getFiles()
```

Пример:

```
Clipboard cb = Clipboard.getSystemClipboard();
if (cb.hasFiles()) {
```

```
System.out.println(cb.GetFiles().toString());  
} // [C:\book\cp1251.txt, C:\book\cp866.txt]
```

- ❑ `hasImage()` — возвращает значение `true`, если в буфере содержатся данные, имеющие тип `DataFormat.IMAGE` (изображение), и `false` — в противном случае. Формат метода:

```
public final boolean hasImage()
```

- ❑ `getImage()` — позволяет получить данные, ассоциированные с типом `DataFormat.IMAGE`. Метод возвращает объект класса `Image` или значение `null`. У меня на компьютере с ОС Windows метод позволяет нормально получить скриншот, сделанный с помощью клавиши <Print Screen>, и изображение, добавленное в буфер обмена из моей программы, но если скопировать изображение из какой-либо другой программы (например, Paint или Photoshop), то изображение из буфера обмена получается испорченным. Формат метода:

```
public final Image getImage()
```

- ❑ `hasContent()` — возвращает значение `true`, если в буфере содержатся данные, имеющие тип `dataFormat`, и `false` — в противном случае. Формат метода:

```
public final boolean hasContent(DataFormat dataFormat)
```

- ❑ `getContent()` — позволяет получить данные, ассоциированные с типом `dataFormat`. Метод возвращает объект класса `Object` или значение `null`. Формат метода:

```
public final Object getContent(DataFormat dataFormat)
```

- ❑ `getContentTypes()` — возвращает множество со всеми доступными типами. Формат метода:

```
public final Set<DataFormat> getContentTypes()
```

4.10.2. Класс *ClipboardContent*

Данные, добавляемые в буфер обмена, описываются с помощью класса `ClipboardContent`. Инструкция импорта:

```
// Модуль javafx.graphics  
import javafx.scene.input.ClipboardContent;
```

Иерархия наследования:

```
Object — AbstractMap<K, V> — HashMap<DataFormat, Object> —  
    ClipboardContent
```

Как видно из иерархии, класс `ClipboardContent` наследует класс `HashMap<K, V>`, а следовательно, реализует интерфейс `Map<K, V>`. Поэтому мы можем воспользоваться методами из этого интерфейса.

Формат конструктора класса `ClipboardContent`:

```
ClipboardContent()
```

Класс `ClipboardContent` содержит следующие методы, позволяющие добавить данные:

- ❑ `putString()` — добавляет данные, ассоциированные с типом `DataFormat.PLAIN_TEXT`. **Формат метода:**

```
public final boolean putString(String s)
```

- ❑ `putHtml()` — добавляет данные, ассоциированные с типом `DataFormat.HTML`. **Формат метода:**

```
public final boolean putHtml(String html)
```

- ❑ `putRtf()` — добавляет данные, ассоциированные с типом `DataFormat.RTF`. **Формат метода:**

```
public final boolean putRtf(String rtf)
```

- ❑ `putUrl()` — добавляет данные, ассоциированные с типом `DataFormat.URL`. **Формат метода:**

```
public final boolean putUrl(String url)
```

- ❑ `putFiles()` и `putFilesByPath()` — добавляют данные, ассоциированные с типом `DataFormat.FILES`. **Форматы методов:**

```
public final boolean putFiles(List<File> files)
public final boolean putFilesByPath(List<String> filePaths)
```

- ❑ `putImage()` — добавляет данные, ассоциированные с типом `DataFormat.IMAGE`. **Формат метода:**

```
public final boolean putImage(Image img)
```

- ❑ `put()` — добавляет данные, ассоциированные с типом `dataFormat`. **Формат метода:**

```
public Object put(DataFormat dataFormat, Object value)
```

Пример добавления простого текста в буфер обмена:

```
Clipboard cb = Clipboard.getSystemClipboard();
ClipboardContent content = new ClipboardContent();
content.putString("Текст, добавляемый в буфер обмена");
cb.setContent(content);
```

Проверить наличие данных определенного типа позволяют методы: `hasString()`, `hasHtml()`, `hasRtf()`, `hasUrl()`, `hasFiles()` и `hasImage()`. Форматы методов:

```
public final boolean hasString()
public final boolean hasHtml()
public final boolean hasRtf()
public final boolean hasUrl()
public final boolean hasFiles()
public final boolean hasImage()
```

Получить данные позволяют методы: `getString()`, `getHtml()`, `getRtf()`, `getUrl()`, `getFiles()`, `getImage()` и `get()`. Форматы методов:

```
public final String getString()
public final String getHtml()
public final String getRtf()
public final String getUrl()
public final List<File> getFiles()
public final Image getImage()
public Object get(Object dataFormat)
```

Если мы хотим добавить в буфер обмена данные пользовательского типа, то вначале нужно зарегистрировать этот тип, создав экземпляр класса `DataFormat`. Формат конструктора класса:

```
// Модуль javafx.graphics
import javafx.scene.input.DataFormat;
DataFormat(String... ids)
```

Пример регистрации пользовательского типа `type/my_type`:

```
DataFormat df = new DataFormat("type/my_type");
```

Дважды зарегистрировать один и тот же тип нельзя. Получить ссылку на тип после регистрации позволяет статический метод `lookupMimeType()` из класса `DataFormat`. Формат метода:

```
public static DataFormat lookupMimeType(String mimeType)
```

Объекты, которые планируется передавать через буфер обмена, должны поддерживать сериализацию. Иными словами, классы должны реализовывать интерфейс `Serializable`. Например, создадим класс `Point`, поддерживающий сериализацию (для уменьшения количества кода поля `x` и `y` объявлены публичными):

```
class Point implements Serializable {
    private static final long serialVersionUID = -730791726267447983L;
    public double x, y;

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
    @Override
    public String toString() {
        return "Point [x=" + x + ", y=" + y + "]";
    }
}
```

Создадим экземпляр этого класса и добавим его в буфер обмена, создав ассоциацию с типом `type/my_type`:

```
DataFormat df = DataFormat.lookupMimeType("type/my_type");
Clipboard cb = Clipboard.getSystemClipboard();
```

```
ClipboardContent content = new ClipboardContent();
content.put(df, new Point(10, 20));
cb.setContent(content);
```

Теперь получим объект из буфера обмена:

```
DataFormat df = DataFormat.lookupMimeType("type/my_type");
Clipboard cb = Clipboard.getSystemClipboard();
if (cb.hasContent(df)) {
    Point p = (Point) cb.getContent(df);
    System.out.println(p); // Point [x=10.0, y=20.0]
}
```

4.11. Технология drag & drop

Технология drag & drop позволяет обмениваться данными различных типов между компонентами как одного приложения, так и разных приложений, путем перетаскивания и сбрасывания объектов с помощью мыши. Типичным примером использования технологии служит перемещение файлов в программе Проводник из ОС Windows. Чтобы переместить файл в другой каталог, достаточно нажать левую кнопку мыши над значком файла и, не отпуская кнопку, перетащить файл на значок каталога, а затем отпустить кнопку мыши. Если необходимо скопировать файл, а не переместить, то следует дополнительно удерживать нажатой клавишу <Ctrl>.

События перетаскивания и сброса описываются с помощью класса `DragEvent`. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.input.DragEvent;
```

Иерархия наследования:

Object — EventObject — Event — InputEvent — DragEvent

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH04_11_drag_and_drop`, приведенном в каталоге `CH04_events` сопровождающего книгу электронного архива (см. приложение).

4.11.1. Типы событий

В классе `DragEvent` определены следующие типы событий:

□ ANY — любое из приведенных далее событий. Формат:

```
public static final EventType<DragEvent> ANY
```

Пример обработки всех событий перетаскивания:

```
scene.addEventFilter(DragEvent.ANY, event -> {
    System.out.println(event.getEventType());
});
```

- ❑ **DRAG_OVER** — перетаскивание внутри области узла. Формат:

```
public static final EventType<DragEvent> DRAG_OVER
```

Внутри обработчика этого события нужно проверить тип перетаскиваемых данных и, если тип устраивает, вызвать метод `acceptTransferModes()` через объект события и указать в качестве параметра поддерживаемые режимы обработки сброса. Пример подтверждения возможности копирования или перемещения текста при сбросе:

```
label.addEventHandler(DragEvent.DRAG_OVER, event -> {
    Dragboard db = event.getDragboard();
    if (db.hasString()) {
        event.acceptTransferModes(TransferMode.COPY_OR_MOVE);
    }
    event.consume();
});
```

Управлять обработчиком события позволяют также следующие методы из классов **Node** и **Scene**:

```
public final void setOnDragOver(
    EventHandler<? super DragEvent> value)
public final EventHandler<? super DragEvent> getOnDragOver()
public final ObjectProperty<EventHandler<? super DragEvent>>
    onDragOverProperty()
```

- ❑ **DRAG_ENTERED** — наведение указателя мыши на узел при перетаскивании (событие доставляется только узлу). Формат:

```
public static final EventType<DragEvent> DRAG_ENTERED
```

Управлять обработчиком события позволяют также следующие методы из классов **Node** и **Scene**:

```
public final void setOnDragEntered(
    EventHandler<? super DragEvent> value)
public final EventHandler<? super DragEvent>
    getOnDragEntered()
public final ObjectProperty<EventHandler<? super DragEvent>>
    onDragEnteredProperty()
```

- ❑ **DRAG_EXITED** — выведение указателя мыши из области узла при перетаскивании (событие доставляется только узлу). Формат:

```
public static final EventType<DragEvent> DRAG_EXITED
```

Управлять обработчиком события позволяют также следующие методы из классов **Node** и **Scene**:

```
public final void setOnDragExited(
    EventHandler<? super DragEvent> value)
public final EventHandler<? super DragEvent> getOnDragExited()
```

```
public final ObjectProperty<EventHandler<? super DragEvent>>
    onDragExitedProperty()
```

- ❑ **DRAG_ENTERED_TARGET** — наведение указателя мыши на узел при перетаскивании. Чтобы определить, какой узел является целью события, следует вызвать метод `getTarget()` через объект события. Формат:

```
public static final EventType<DragEvent> DRAG_ENTERED_TARGET
```

- ❑ **DRAG_EXITED_TARGET** — выведение указателя мыши из области узла при перетаскивании. Чтобы определить, какой узел является целью события, следует вызвать метод `getTarget()` через объект события. Формат:

```
public static final EventType<DragEvent> DRAG_EXITED_TARGET
```

- ❑ **DRAG_DROPPED** — отпускание кнопки мыши при перетаскивании (событие сброса). Формат:

```
public static final EventType<DragEvent> DRAG_DROPPED
```

Внутри обработчика этого события нужно выполнить обработку сброса данных: проверить тип данных и получить эти данные. После получения данных необходимо сообщить результат операции сброса, вызвав метод `setDropCompleted()` через объект события. Пример обработки сброса строки:

```
label.addEventHandler(DragEvent.DRAG_DROPPED, event -> {
    Dragboard db = event.getDragboard();
    boolean success = false;
    if (db.hasString()) {
        label.setText(db.getString());
        success = true;
    }
    event.setDropCompleted(success);
    event.consume();
});
```

Управлять обработчиком события позволяют также следующие методы из классов `Node` и `Scene`:

```
public final void setOnDragDropped(
    EventHandler<? super DragEvent> value)
public final EventHandler<? super DragEvent>
    getOnDragDropped()
public final ObjectProperty<EventHandler<? super DragEvent>>
    onDragDroppedProperty()
```

- ❑ **DRAG_DONE** — событие окончания операции перетаскивания. Формат:

```
public static final EventType<DragEvent> DRAG_DONE
```

Внутри обработчика этого события можно узнать результат сброса и какой режим использовался при удачном сбросе. Пример проверки статуса сброса:

```
label.addEventHandler(DragEvent.DRAG_DONE, event -> {
    if (event.isAccepted()) {
```

```

        System.out.println("Сброс выполнен успешно. Режим: " +
            event.getTransferMode());
    }
    else {
        System.out.println("Сброс не выполнен");
    }
});

```

Управлять обработчиком события позволяют также следующие методы из классов `Node` и `Scene`:

```

public final void setOnDragDone(
    EventHandler<? super DragEvent> value)
public final EventHandler<? super DragEvent> getOnDragDone()
public final ObjectProperty<EventHandler<? super DragEvent>>
    onDragDoneProperty()

```

4.11.2. Запуск перетаскивания

Операция перетаскивания состоит из трех этапов. Первый этап запускает процесс, второй этап обрабатывает момент перемещения и сброса объекта, а третий этап позволяет проверить результат операции. События перетаскивания могут обрабатываться как одним приложением, так и двумя разными приложениями. Запуск перетаскивания осуществляется следующим образом:

- внутри обработчика события `MouseEvent.DRAG_DETECTED` вызываем метод `startDragAndDrop()` и в качестве параметра передаем возможные режимы сброса (например, копирование или перемещение). Этот метод есть как в классе `Node`, так и в классе `Scene`. Формат метода:

```
public Dragboard startDragAndDrop(TransferMode... transferModes)
```

Метод возвращает объект класса `Dragboard`. Инструкция импорта класса:

```
import javafx.scene.input.Dragboard;
```

Иерархия наследования:

```
Object — Clipboard — Dragboard
```

Класс `Dragboard` наследует класс `Clipboard` (см. *разд. 4.10.1*), а следовательно, и все методы из этого класса. Иными словами, работать с объектом `Dragboard` можно точно так же, как и с буфером обмена;

- после получения ссылки на объект `Dragboard` добавляем перетаскиваемые данные в буфер. Данные, добавляемые в буфер, описываются с помощью класса `ClipboardContent` (см. *разд. 4.10.2*);
- вызываем метод `consume()` через объект события.

Пример запуска перетаскивания текста из надписи:

```

label.addEventHandler(MouseEvent.DRAG_DETECTED, event -> {
    Dragboard db = label.startDragAndDrop(TransferMode.COPY_OR_MOVE);

```



```
ClipboardContent content = new ClipboardContent();
content.putString(label.getText());
db.setContent(content);
event.consume();
});
```

Проверка результата операции перетаскивания осуществляется узлом, который запустил процесс перетаскивания. Для этого нужно назначить обработчик для события `DragEvent.DRAG_DONE`. Если операция перетаскивания завершена успешно, то метод `isAccepted()` объекта события вернет значение `true`. С помощью метода `getTransferMode()` можно определить, какой режим использовался при сбросе, например, копирование или перемещение. Если использовался режим перемещения, то можно, например, удалить исходные данные. Пример обработки результата сброса:

```
label.addEventHandler(DragEvent.DRAG_DONE, event -> {
    if (event.isAccepted()) {
        System.out.println("Сброс выполнен успешно. Режим: " +
            event.getTransferMode());
    }
    else {
        System.out.println("Сброс не выполнен");
    }
});
```

4.11.3. Обработка сброса

Обработка перетаскивания и сброса выполняется следующим образом:

- ❑ внутри обработчика события `DragEvent.DRAG_OVER` нужно проверить тип перетаскиваемых данных и, если тип устраивает, вызвать метод `acceptTransferModes()` через объект события и указать в качестве параметра поддерживаемые режимы обработки сброса. Если мы указали несколько режимов, то пользователь может выбрать нужный ему режим (переключение между режимами выполняется путем нажатия клавиши `<Ctrl>`). Обработчик вызывается постоянно при перемещении указателя мыши с нажатой кнопкой. Пример подтверждения возможности копирования или перемещения текста при сбросе:

```
label.addEventHandler(DragEvent.DRAG_OVER, event -> {
    Dragboard db = event.getDragboard();
    if (db.hasString()) {
        event.acceptTransferModes(TransferMode.COPY_OR_MOVE);
    }
    event.consume();
});
```

- ❑ внутри обработчика события `DragEvent.DRAG_DROPPED` можно выполнить обработку сброса данных: проверить тип данных и получить эти данные. Событие возникает, когда пользователь отпускает кнопку мыши, завершая тем самым

процесс перетаскивания. После получения данных необходимо сообщить результат операции сброса, вызвав метод `setDropCompleted()` через объект события. Результат сброса будет доступен внутри обработчика события `DragEvent.DRAG_DONE`. Пример обработки сброса строки:

```
label.addEventHandler(DragEvent.DRAG_DROPPED, event -> {
    Dragboard db = event.getDragboard();
    boolean success = false;
    if (db.hasString()) {
        label.setText(db.getString());
        success = true;
    }
    event.setDropCompleted(success);
    event.consume();
});
```

4.11.4. Получение информации о событии

Получить информацию о событии позволяют следующие основные методы из класса `DragEvent` (полный список методов смотрите в документации):

- ❑ `acceptTransferModes()` — внутри обработчика события `DragEvent.DRAG_OVER` позволяет указать режимы, поддерживаемые при сбросе. Формат метода:

```
public void acceptTransferModes(TransferMode... transferModes)
```

Перечисление `TransferMode` содержит следующие константы:

```
public static final TransferMode[] ANY
public static final TransferMode[] COPY_OR_MOVE
public static final TransferMode COPY
public static final TransferMode MOVE
public static final TransferMode LINK
public static final TransferMode[] NONE
```

- ❑ `getTransferMode()` — возвращает режим сброса, например `TransferMode.COPY` или `TransferMode.MOVE`. Внутри обработчика события `DragEvent.DRAG_DONE` метод вернет фактический режим, который был использован при удачном сбросе, или значение `null` при неудачном. Формат метода:

```
public final TransferMode getTransferMode()
```

- ❑ `setDropCompleted()` — внутри обработчика события `DragEvent.DRAG_DROPPED` позволяет задать статус обработки сброса. В случае успешной операции нужно указать значение `true`. После вызова этого метода работа с буфером становится невозможной. Формат метода:

```
public void setDropCompleted(boolean isTransferDone)
```

- ❑ `isAccepted()` — возвращает значение `true`, если сброс выполнен успешно, и `false` — в противном случае. Формат метода:

```
public final boolean isAccepted()
```

- `getDragboard()` — возвращает ссылку на буфер, используемый при перетаскивании. Работать с этим буфером можно точно так же, как и с буфером обмена (см. *разд. 4.10*), поскольку класс `Dragboard` наследует класс `Clipboard`. Формат метода:

```
public final Dragboard getDragboard()
```

- `getX()`, `getY()` и `getZ()` — возвращают координаты указателя относительно узла. Например, если обрабатываем перемещение указателя внутри прямоугольника, то получим координаты относительно левого верхнего угла прямоугольника. Форматы методов:

```
public final double getX()
public final double getY()
public final double getZ()
```

- `getSceneX()` и `getSceneY()` — возвращают координаты указателя относительно сцены. Форматы методов:

```
public final double getSceneX()
public final double getSceneY()
```

- `getScreenX()` и `getScreenY()` — возвращают координаты указателя относительно левого верхнего угла экрана. Форматы методов:

```
public final double getScreenX()
public final double getScreenY()
```

Не следует также забывать, что класс `DragEvent` наследует класс `Event`, а следовательно, наследует и все его методы (см. *разд. 4.5*). Особенно не нужно забывать вызывать метод `consume()` внутри обработчиков событий перетаскивания после обработки событий.

4.11.5. Класс *Dragboard*

Класс `Dragboard` описывает буфер, используемый для хранения данных при перетаскивании. Инструкция импорта класса:

```
// Модуль javafx.graphics
import javafx.scene.input.Dragboard;
```

Иерархия наследования:

```
Object — Clipboard — Dragboard
```

Класс `Dragboard` наследует класс `Clipboard` (см. *разд. 4.10.1*), а следовательно, и все методы из этого класса. Иными словами, работать с объектом `Dragboard` можно точно так же, как и с буфером обмена. Данные, добавляемые в буфер, описываются с помощью класса `ClipboardContent` (см. *разд. 4.10.2*). Классы `Clipboard` и `ClipboardContent` мы уже рассматривали, поэтому не будем повторяться.

Помимо добавления данных в буфер и получения данных, класс `Dragboard` позволяет задать изображение, которое будет отображаться под указателем мыши в про-

цессе перетаскивания. Для этого предназначены следующие методы из класса `Dragboard`:

❑ `setDragView()` — добавляет изображение. Форматы метода:

```
public void setDragView(Image image)
public void setDragView(Image image, double offsetX,
                        double offsetY)
```

❑ `setDragViewOffsetX()` и `setDragViewOffsetY()` — задают смещение по осям X и Y соответственно. Форматы методов:

```
public void setDragViewOffsetX(double offsetX)
public void setDragViewOffsetY(double offsetY)
```

Добавление изображения и настройку смещения необходимо выполнять внутри обработчика события `MouseEvent.DRAG_DETECTED`. Пример:

```
label.addEventHandler(MouseEvent.DRAG_DETECTED, event -> {
    Dragboard db = label.startDragAndDrop(TransferMode.COPY_OR_MOVE);
    try {
        db.setDragView(new Image(
            getClass().getResourceAsStream("/img/blue32x32.png")));
        db.setDragViewOffsetX(32.0);
        db.setDragViewOffsetY(32.0);
    } catch (Exception e) {
        System.out.println("Не удалось загрузить картинку");
    }
    ClipboardContent content = new ClipboardContent();
    content.putString(label.getText());
    db.setContent(content);
    event.consume();
});
```

4.11.6. Класс *MouseDragEvent*

Если внутри обработчика события `MouseEvent.DRAG_DETECTED` вместо метода `startDragAndDrop()` вызвать метод `startFullDrag()`, то станут доступными для обработки события перетаскивания узла, описываемые классом `MouseDragEvent`. Формат метода `startFullDrag()`:

```
public void startFullDrag()
```

Инструкция импорта класса `MouseDragEvent`:

```
// Модуль javafx.graphics
import javafx.scene.input.MouseDragEvent;
```

Иерархия наследования:

```
Object — EventObject — Event — InputEvent —
      MouseEvent — MouseDragEvent
```

Класс `MouseEvent` наследует все методы из класса `MouseEvent` и добавляет метод `getGestureSource()`, который возвращает ссылку на перетаскиваемый узел. Формат метода:

```
public Object getGestureSource()
```

В классе `MouseEvent` определены следующие типы событий:

❑ ANY — любое из приведенных далее событий. Формат:

```
public static final EventType<MouseEvent> ANY
```

❑ `MOUSE_DRAG_ENTERED` — наведение указателя мыши на узел при перетаскивании (событие доставляется только узлу). Формат:

```
public static final EventType<MouseEvent> MOUSE_DRAG_ENTERED
```

Управлять обработчиком события позволяют также следующие методы из классов `Node` и `Scene`:

```
public final void setOnMouseDragEntered(
    EventHandler<? super MouseEvent> value)
public final EventHandler<? super MouseEvent>
    getOnMouseDragEntered()
public final ObjectProperty<EventHandler<
    ? super MouseEvent>> onMouseDragEnteredProperty()
```

❑ `MOUSE_DRAG_EXITED` — выведение указателя мыши из области узла при перетаскивании (событие доставляется только узлу). Формат:

```
public static final EventType<MouseEvent> MOUSE_DRAG_EXITED
```

Управлять обработчиком события позволяют также следующие методы из классов `Node` и `Scene`:

```
public final void setOnMouseDragExited(
    EventHandler<? super MouseEvent> value)
public final EventHandler<? super MouseEvent>
    getOnMouseDragExited()
public final ObjectProperty<EventHandler<
    ? super MouseEvent>> onMouseDragExitedProperty()
```

❑ `MOUSE_DRAG_ENTERED_TARGET` — наведение указателя мыши на узел при перетаскивании. Формат:

```
public static final EventType<MouseEvent>
    MOUSE_DRAG_ENTERED_TARGET
```

❑ `MOUSE_DRAG_EXITED_TARGET` — выведение указателя мыши из области узла при перетаскивании. Формат:

```
public static final EventType<MouseEvent>
    MOUSE_DRAG_EXITED_TARGET
```

- ❑ **MOUSE_DRAG_OVER** — перетаскивание внутри области узла. Формат:

```
public static final EventType<MouseEvent> MOUSE_DRAG_OVER
```

Управлять обработчиком события позволяют также следующие методы из классов **Node** и **Scene**:

```
public final void setOnMouseDragOver(
    EventHandler<? super MouseEvent> value)
public final EventHandler<? super MouseEvent>
    getOnMouseDragOver()
public final ObjectProperty<EventHandler<
    ? super MouseEvent>> onMouseDragOverProperty()
```

- ❑ **MOUSE_DRAG_RELEASED** — отпускание кнопки мыши при перетаскивании (событие сброса). Формат:

```
public static final EventType<MouseEvent>
    MOUSE_DRAG_RELEASED
```

Управлять обработчиком события позволяют также следующие методы из классов **Node** и **Scene**:

```
public final void setOnMouseDragReleased(
    EventHandler<? super MouseEvent> value)
public final EventHandler<? super MouseEvent>
    getOnMouseDragReleased()
public final ObjectProperty<EventHandler<
    ? super MouseEvent>> onMouseDragReleasedProperty()
```

ЭЛЕКТРОННЫЙ АРХИВ

Пример обработки событий содержится в проекте `FX_CH04_11_MouseDragEvent`, приведенном в каталоге `CH04_events` сопровождающего книгу электронного архива (см. *приложение*).

4.12. Класс *Robot*: генерация событий мыши и клавиатуры

Начиная с JavaFX 11, в состав библиотеки входит класс `Robot`, с помощью которого можно имитировать различные события мыши и клавиатуры, как будто источником события является пользователь. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.robot.Robot;
```

Иерархия наследования:

```
Object — Robot
```

Конструктор класса:

```
Robot()
```

Пример создания объекта:

```
Robot robot = new Robot();
```

Методы класса Robot:

- ❑ `getMouseX()`, `getMouseY()` и `getMousePosition()` — возвращают позицию указателя мыши. **Форматы методов:**

```
public double getMouseX()  
public double getMouseY()  
public Point2D getMousePosition()
```

- ❑ `mouseMove()` — перемещает указатель мыши в указанную позицию. **Форматы метода:**

```
public void mouseMove(double x, double y)  
public final void mouseMove(Point2D location)
```

- ❑ `mouseClick()` — выполняет щелчок мышью. Нажатые кнопки мыши указываются в качестве параметра. **Формат метода:**

```
public void mouseClick(MouseButton... buttons)
```

Пример:

```
robot.mouseClick(MouseButton.PRIMARY);
```

- ❑ `mousePress()` — нажатие кнопки мыши. **Формат метода:**

```
public void mousePress(MouseButton... buttons)
```

- ❑ `mouseRelease()` — отпускание ранее нажатой кнопки мыши. **Формат метода:**

```
public void mouseRelease(MouseButton... buttons)
```

Пример щелчка левой кнопкой мыши:

```
robot.mousePress(MouseButton.PRIMARY);  
robot.mouseRelease(MouseButton.PRIMARY);
```

- ❑ `mouseWheel()` — прокрутка колесика мыши. **Формат метода:**

```
public void mouseWheel(int wheelAmt)
```

- ❑ `keyPress()` — нажатие клавиши на клавиатуре. **Формат метода:**

```
public void keyPress(KeyCode keyCode)
```

- ❑ `keyRelease()` — отпускание ранее нажатой клавиши. **Формат метода:**

```
public void keyRelease(KeyCode keyCode)
```

Пример нажатия и отпускания клавиши <Q>:

```
robot.keyPress(KeyCode.Q);  
robot.keyRelease(KeyCode.Q);
```

□ `keyType()` — комбинация `keyPress()` и `keyRelease()`. Формат метода:

```
public void keyType(KeyCode keyCode)
```

Пример нажатия клавиши <W>:

```
robot.keyType(KeyCode.W);
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH04_12_Robot`, приведенном в каталоге `CH04_events` сопровождающего книгу электронного архива (см. приложение).

ГЛАВА 5



2D-графика

Начиная с этой главы, мы приступаем к изучению работы с графикой, изображениями, эффектами и анимацией. JavaFX поддерживает как двухмерную (2D) графику, так и трехмерную (3D). Итак, за работу!

5.1. Указание координат и размеров

Прежде чем мы начнем изучать работу с графикой и фигурами, необходимо рассмотреть несколько вспомогательных классов, с помощью которых задаются координаты точек и размеры прямоугольных областей. Обратите внимание: все методы рассматриваемых классов возвращают или какое-либо значение, или новый объект. Ни один метод не изменяет свойства существующего объекта.

5.1.1. Класс *Point2D*: координаты точки

Класс `Point2D` описывает координаты точки в двумерном пространстве и позволяет выполнять различные действия с векторами. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.geometry.Point2D;
```

Формат конструктора класса:

```
Point2D(double x, double y)
```

Пример создания объекта:

```
Point2D p = new Point2D(10.0, 20.0);
System.out.println(p); // Point2D [x = 10.0, y = 20.0]
```

В классе `Point2D` определена статическая константа `ZERO`, которая возвращает объект с нулевыми координатами. Формат:

```
public static final Point2D ZERO
```

Пример вывода значения:

```
System.out.println(Point2D.ZERO); // Point2D [x = 0.0, y = 0.0]
```

Приведем методы класса `Point2D`:

- ❑ `getX()` и `getY()` — возвращают координаты по осям X и Y соответственно. Форматы методов:

```
public final double getX()
public final double getY()
```

Пример:

```
Point2D p = new Point2D(10.0, 20.0);
System.out.println(p.getX() + " " + p.getY()); // 10.0 20.0
```

- ❑ `equals()` — сравнивает две точки и возвращает значение `true`, если координаты равны. Формат метода:

```
public boolean equals(Object obj)
```

Пример:

```
Point2D p = new Point2D(10.0, 20.0);
Point2D p2 = new Point2D(40.0, 80.0);
Point2D p3 = new Point2D(10.0, 20.0);
System.out.println(p.equals(p2)); // false
System.out.println(p.equals(p3)); // true
```

- ❑ `add()` — сложение векторов. Форматы метода:

```
public Point2D add(double x, double y)
public Point2D add(Point2D point)
```

Пример:

```
Point2D p = new Point2D(10.0, 20.0);
Point2D p2 = new Point2D(40.0, 80.0);
System.out.println(p.add(5.0, 3.0));
// Point2D [x = 15.0, y = 23.0]
System.out.println(p.add(p2));
// Point2D [x = 50.0, y = 100.0]
```

- ❑ `subtract()` — вычитание векторов. Форматы метода:

```
public Point2D subtract(double x, double y)
public Point2D subtract(Point2D point)
```

Пример:

```
Point2D p = new Point2D(10.0, 20.0);
System.out.println(p.subtract(5.0, 3.0));
// Point2D [x = 5.0, y = 17.0]
System.out.println(p.subtract(new Point2D(10.0, 20.0)));
// Point2D [x = 0.0, y = 0.0]
```

- ❑ `multiply()` — умножение вектора на скаляр. Формат метода:

```
public Point2D multiply(double factor)
```

Пример:

```
Point2D p = new Point2D(10.0, 20.0);
System.out.println(p.multiply(5.0));
// Point2D [x = 50.0, y = 100.0]
```

- **distance()** — возвращает расстояние между двумя точками. Форматы метода:

```
public double distance(double x1, double y1)
public double distance(Point2D point)
```

Пример:

```
Point2D p = new Point2D(10.0, 20.0);
System.out.println(p.distance(50.0, 20.0)); // 40.0
System.out.println(p.distance(new Point2D(10.0, 20.0)));
// 0.0
```

- **normalize()** — производит нормализацию вектора. *Нормализованным* называется вектор с длиной, равной единице. Формат метода:

```
public Point2D normalize()
```

Пример:

```
Point2D p = new Point2D(3.0, 4.0);
System.out.println(p.normalize());
// Point2D [x = 0.6, y = 0.8]
```

Процесс вычисления значений выглядит следующим образом:

```
double len = Math.sqrt(3.0 * 3.0 + 4.0 * 4.0);
System.out.println(len); // 5.0
System.out.println(3.0 / len); // 0.6
System.out.println(4.0 / len); // 0.8
```

Действительно ли длина вектора равна единице? Давайте проверим:

```
System.out.println(Math.sqrt(0.6 * 0.6 + 0.8 * 0.8)); // 1.0
```

- **magnitude()** — возвращает длину вектора. Формат метода:

```
public double magnitude()
```

Пример:

```
Point2D p = new Point2D(0.6, 0.8);
System.out.println(p.magnitude()); // 1.0
System.out.println(Math.sqrt(0.6 * 0.6 + 0.8 * 0.8)); // 1.0
```

- **dotProduct()** — скалярное произведение векторов. Форматы метода:

```
public double dotProduct(double x, double y)
public double dotProduct(Point2D vector)
```

Пример:

```
Point2D p = new Point2D(3.0, 2.0);
System.out.println(p.dotProduct(new Point2D(1.0, 4.0)));
```

```
// 11.0
System.out.println(p.dotProduct(1.0, 4.0)); // 11.0
System.out.println(3.0 * 1.0 + 2.0 * 4.0); // 11.0
```

- ❑ **crossProduct()** — **векторное произведение векторов. Форматы метода:**

```
public Point3D crossProduct(double x, double y)
public Point3D crossProduct(Point2D vector)
```

Пример:

```
Point2D p = new Point2D(1.0, 2.0);
Point2D p2 = new Point2D(3.0, 4.0);
System.out.println(p.crossProduct(p2));
// Point3D [x = 0.0, y = 0.0, z = -2.0]
```

Процесс вычисления выглядит следующим образом:

```
double[] a = {1.0, 2.0, 0.0};
double[] b = {3.0, 4.0, 0.0};
System.out.println((a[1] * b[2] - a[2] * b[1]) + ", " +
                   (a[2] * b[0] - a[0] * b[2]) + ", " +
                   (a[0] * b[1] - a[1] * b[0]));
// 0.0, 0.0, -2.0
```

- ❑ **midpoint()** — **возвращает координаты точки, расположенной посередине между двумя указанными точками. Форматы метода:**

```
public Point2D midpoint(double x, double y)
public Point2D midpoint(Point2D point)
```

Пример:

```
Point2D p = new Point2D(10.0, 20.0);
System.out.println(p.midpoint(new Point2D(10.0, 80.0)));
// Point2D [x = 10.0, y = 50.0]
```

- ❑ **angle()** — **возвращает угол (в градусах) между векторами или тремя точками. Если указан нулевой вектор или если три точки не отличаются друг от друга, метод вернет значение NaN. Форматы метода:**

```
public double angle(double x, double y)
public double angle(Point2D point)
public double angle(Point2D p1, Point2D p2)
```

Пример:

```
Point2D p = new Point2D(1.0, 0.0);
Point2D p2 = new Point2D(0.0, 1.0);
Point2D p3 = new Point2D(0.0, 0.0);
System.out.println(p.angle(p2)); // 90.0
System.out.println(p.angle(-1.0, 0.0)); // 180.0
System.out.println(p3.angle(p, p2)); // 90.0
```

5.1.2. Класс *Dimension2D*: размеры прямоугольной области

Класс `Dimension2D` описывает размеры прямоугольной области. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.geometry.Dimension2D;
```

Формат конструктора класса:

```
Dimension2D(double width, double height)
```

Получить значения ширины и высоты позволяют методы `getWidth()` и `getHeight()`.

Форматы методов:

```
public final double getWidth()
public final double getHeight()
```

Пример:

```
Dimension2D rect = new Dimension2D(100.0, 200.0);
System.out.println(rect);
// Dimension2D [width = 100.0, height = 200.0]
System.out.println(rect.getWidth()); // 100.0
System.out.println(rect.getHeight()); // 200.0
```

5.1.3. Класс *Rectangle2D*: координаты и размеры прямоугольной области

Класс `Rectangle2D` описывает координаты и размеры прямоугольной области. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.geometry.Rectangle2D;
```

Формат конструктора класса:

```
Rectangle2D(double minX, double minY, double width, double height)
```

Пример создания объекта:

```
Rectangle2D rect = new Rectangle2D(1.0, 2.0, 100.0, 200.0);
System.out.println(rect);
// Rectangle2D [minX = 1.0, minY=2.0, maxX=101.0,
// maxY=202.0, width=100.0, height=200.0]
```

В классе `Rectangle2D` определена статическая константа `EMPTY`, которая возвращает объект с нулевыми координатами и размерами. Формат:

```
public static final Rectangle2D EMPTY
```

Пример вывода значения:

```
System.out.println(Rectangle2D.EMPTY);
// Rectangle2D [minX = 0.0, minY=0.0, maxX=0.0,
// maxY=0.0, width=0.0, height=0.0]
```

Приведем методы класса `Rectangle2D`:

- ❑ `getMinX()` и `getMinY()` — возвращают координаты левого верхнего угла. Форматы методов:

```
public double getMinX()
public double getMinY()
```

- ❑ `getMaxX()` и `getMaxY()` — возвращают координаты правого нижнего угла. Форматы методов:

```
public double getMaxX()
public double getMaxY()
```

- ❑ `getWidth()` и `getHeight()` — возвращают ширину и высоту соответственно. Форматы методов:

```
public double getWidth()
public double getHeight()
```

Пример:

```
Rectangle2D rect = new Rectangle2D(1.0, 2.0, 100.0, 200.0);
System.out.println(rect.getMinX() + " " + rect.getMinY());
// 1.0 2.0
System.out.println(rect.getMaxX() + " " + rect.getMaxY());
// 101.0 202.0
System.out.println(rect.getWidth() + " " + rect.getHeight());
// 100.0 200.0
```

- ❑ `equals()` — сравнивает два объекта и возвращает значение `true`, если координаты и размеры объектов равны. Формат метода:

```
public boolean equals(Object obj)
```

Пример:

```
Rectangle2D rect = new Rectangle2D(1.0, 2.0, 100.0, 200.0);
Rectangle2D rect2 = new Rectangle2D(1.0, 1.0, 100.0, 200.0);
Rectangle2D rect3 = new Rectangle2D(1.0, 2.0, 100.0, 200.0);
System.out.println(rect.equals(rect2)); // false
System.out.println(rect.equals(rect3)); // true
```

- ❑ `intersects()` — возвращает значение `true`, если прямоугольные области пересекаются, и `false` — в противном случае. Форматы метода:

```
public boolean intersects(double x, double y,
                        double w, double h)
public boolean intersects(Rectangle2D r)
```

Пример:

```
Rectangle2D rect = new Rectangle2D(0.0, 0.0, 100.0, 200.0);
Rectangle2D rect2 = new Rectangle2D(100.0, 0.0, 10.0, 20.0);
System.out.println(rect.intersects(rect2)); // false
System.out.println(
    rect.intersects(99.0, 0.0, 10.0, 20.0)); // true
```

- `contains()` — возвращает значение `true`, если указанная точка находится внутри прямоугольника или указанный прямоугольник находится внутри границ текущего прямоугольника, и `false` — в противном случае. Форматы метода:

```
public boolean contains(double x, double y)
public boolean contains(Point2D p)
public boolean contains(double x, double y,
                        double w, double h)
public boolean contains(Rectangle2D r)
```

Пример:

```
Rectangle2D rect = new Rectangle2D(0.0, 0.0, 100.0, 200.0);
Rectangle2D rect2 = new Rectangle2D(95.0, 0.0, 10.0, 20.0);
System.out.println(rect.contains(200.0, 0.0)); // false
System.out.println(
    rect.contains(new Point2D(30.0, 20.0))); // true
System.out.println(rect.contains(rect2)); // false
System.out.println(
    rect.contains(90.0, 180.0, 10.0, 20.0)); // true
```

5.1.4. Классы *Bounds* и *BoundingBox*

Абстрактный класс `Bounds` описывает координаты и размеры области, внутри которой расположен узел. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.geometry.Bounds;
```

Реализацией абстрактного класса `Bounds` является класс `BoundingBox`. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.geometry.BoundingBox;
```

Форматы конструктора класса `BoundingBox`:

```
BoundingBox(double minX, double minY,
            double width, double height)
BoundingBox(double minX, double minY, double minZ,
            double width, double height, double depth)
```

Первый конструктор описывает область в двумерном пространстве, а второй — в трехмерном. Во втором конструкторе дополнительно указывается координата по оси *z* (параметр `minZ`) и глубина (параметр `depth`). Пример создания объекта:

```
Bounds b = new BoundingBox(0.0, 0.0, 100.0, 200.0);
System.out.println(b);
// BoundingBox [minX:0.0, minY:0.0, minZ:0.0,
// width:100.0, height:200.0, depth:0.0, maxX:100.0,
// maxY:200.0, maxZ:0.0]
```


Приведем основные методы:

- ❑ `getMinX()`, `getMinY()` и `getMinZ()` — возвращают координаты левого верхнего угла. **Форматы методов:**

```
public final double getMinX()
public final double getMinY()
public final double getMinZ()
```

- ❑ `getMaxX()`, `getMaxY()` и `getMaxZ()` — возвращают координаты правого нижнего угла. **Форматы методов:**

```
public final double getMaxX()
public final double getMaxY()
public final double getMaxZ()
```

- ❑ `getWidth()`, `getHeight()` и `getDepth()` — возвращают ширину, высоту и глубину соответственно. **Форматы методов:**

```
public final double getWidth()
public final double getHeight()
public final double getDepth()
```

Пример:

```
Bounds b = new BoundingBox(5.0, 7.0, 100.0, 200.0);
System.out.println(b.getMinX() + " " + b.getMinY() +
    " " + b.getMinZ());
// 5.0 7.0 0.0
System.out.println(b.getMaxX() + " " + b.getMaxY() +
    " " + b.getMaxZ());
// 105.0 207.0 0.0
System.out.println(b.getWidth() + " " + b.getHeight());
// 100.0 200.0
System.out.println(b.getDepth());
// 0.0
```

- ❑ `getCenterX()`, `getCenterY()` и `getCenterZ()` — возвращают координаты центра области. Методы доступны, начиная с JavaFX 11. **Форматы методов:**

```
public final double getCenterX()
public final double getCenterY()
public final double getCenterZ()
```

Пример:

```
Bounds b = new BoundingBox(5.0, 7.0, 100.0, 200.0);
System.out.println(b.getCenterX() + " " + b.getCenterY() +
    " " + b.getCenterZ());
// 55.0 107.0 0.0
```

- ❑ `intersects()` — возвращает значение `true`, если области пересекаются, и `false` — в противном случае. **Форматы метода:**

```
public boolean intersects(Bounds b)
public boolean intersects(double x, double y,
                        double w, double h)
public boolean intersects(double x, double y, double z,
                        double w, double h, double d)
```

Пример:

```
Bounds b = new BoundingBox(0.0, 0.0, 100.0, 200.0);
Bounds b2 = new BoundingBox(101.0, 0.0, 10.0, 20.0);
System.out.println(b.intersects(b2));    // false
System.out.println(
    b.intersects(99.0, 0.0, 10.0, 20.0)); // true
```

- **contains()** — возвращает значение true, если указанная точка находится внутри области или указанная область находится внутри границ текущей области, и false — в противном случае. Форматы метода:

```
public boolean contains(double x, double y)
public boolean contains(Point2D p)
public boolean contains(double x, double y, double z)
public boolean contains(Point3D p)
public boolean contains(Bounds b)
public boolean contains(double x, double y,
                        double w, double h)
public boolean contains(double x, double y, double z,
                        double w, double h, double d)
```

Пример:

```
Bounds b = new BoundingBox(0.0, 0.0, 100.0, 200.0);
Bounds b2 = new BoundingBox(95.0, 0.0, 10.0, 20.0);
System.out.println(b.contains(200.0, 0.0)); // false
System.out.println(
    b.contains(new Point2D(30.0, 20.0)));    // true
System.out.println(b.contains(b2));         // false
System.out.println(
    b.contains(90.0, 180.0, 10.0, 20.0));    // true
```

- **isEmpty()** — возвращает значение true, если какой-либо из размеров (ширина, высота или глубина) меньше нуля, и false — в противном случае. Формат метода:

```
public boolean isEmpty()
```

Пример:

```
Bounds b = new BoundingBox(0.0, 0.0, 100.0, 200.0);
Bounds b2 = new BoundingBox(95.0, 0.0, -10.0, 20.0);
System.out.println(b.isEmpty());           // false
System.out.println(b2.isEmpty());          // true
```

Получить координаты и размеры области, внутри которой расположен узел, позволяют следующие свойства из класса `Node`:

- `layoutBounds` — описывает область, используемую для расчета макета узла. Не включает данные о положении и смещении, а также не учитывает эффекты и преобразования. Методы доступа:

```
public final Bounds getLayoutBounds()
public final ReadOnlyObjectProperty<Bounds>
    layoutBoundsProperty()
```

- `boundsInLocal` — описывает область в не трансформированном локальном пространстве координат узла. Помимо размеров узла, используемых для расчета макета, учитывает и вспомогательные элементы, например границы узла и результат применения эффектов. Методы доступа:

```
public final Bounds getBoundsInLocal()
public final ReadOnlyObjectProperty<Bounds>
    boundsInLocalProperty()
```

- `boundsInParent` — описывает область в пространстве координат родительского узла. Учитывает положение, смещение, различные трансформации и др. Методы доступа:

```
public final Bounds getBoundsInParent()
public final ReadOnlyObjectProperty<Bounds>
    boundsInParentProperty()
```

Предположим, у нас есть кнопка внутри корневого контейнера, для которой заданы координаты и предпочтительные размеры, причем кнопка повернута на 45 градусов:

```
Button btn = new Button("Вывести координаты");
btn.setPrefSize(140.0, 25.0);
btn.relocate(100.0, 100.0);
btn.setRotate(45.0);
```

Результат вывода координат и размеров областей будет выглядеть следующим образом:

```
getLayoutBounds(): BoundingBox [minX:0.0, minY:0.0, minZ:0.0,
    width:140.0, height:25.0, depth:0.0, maxX:140.0, maxY:25.0, maxZ:0.0]
getBoundsInLocal(): BoundingBox [minX:-1.399999976158142,
    minY:-1.399999976158142, minZ:0.0, width:142.79998779296875,
    height:27.799999237060547, depth:0.0, maxX:141.3999878168106,
    maxY:26.399999260902405, maxZ:0.0]
getBoundsInParent(): BoundingBox [minX:109.68379211425781,
    minY:52.18379211425781, minZ:0.0, width:120.63241577148438,
    height:120.63241577148438, depth:0.0, maxX:230.3162078857422,
    maxY:172.8162078857422, maxZ:0.0]
```

ЭЛЕКТРОННЫЙ АРХИВ

Пример использования свойств `layoutBounds`, `boundsInLocal` и `boundsInParent` содержится в проектах `FX_CH03_02_size` и `FX_CH03_04_relocate`, приведенных в каталоге `CH03_containers` сопровождающего книгу электронного архива (см. приложение).

5.2. Класс *Paint*

Абстрактный класс `Paint` является базовым для классов `Color` (цвет, см. разд. 5.3), `LinearGradient` (линейный градиент, см. разд. 5.4), `RadialGradient` (радиальный градиент, см. разд. 5.5) и `ImagePattern` (изображение в качестве фона, см. разд. 6.7). Именно этот класс указывается, когда нужно задать свойства фона. Например, формат метода `setFill()` из класса `Scene` выглядит следующим образом:

```
public final void setFill(Paint value)
```

Благодаря указанию класса `Paint`, мы можем задать как сплошной цвет, так и выполнить заливку (градиентную или на основе текстуры).

Инструкция импорта класса `Paint`:

```
// Модуль javafx.graphics
import javafx.scene.paint.Paint;
```

Класс `Paint` содержит следующие методы:

- ❑ `valueOf()` — позволяет создать экземпляры классов `Color`, `LinearGradient` или `RadialGradient` на основе строки специального формата. Метод является статическим. Формат метода:

```
public static Paint valueOf(String value)
```

Пример создания объекта класса `Color`:

```
Paint p = Paint.valueOf("#ff0000");
System.out.println(p); // 0xff0000ff
System.out.println(p.getClass());
// class javafx.scene.paint.Color
```

- ❑ `isOpaque()` — возвращает значение `true`, если заливка полностью непрозрачна, и `false` — в противном случае. Метод является абстрактным и реализуется в производных классах. Формат метода:

```
public abstract boolean isOpaque()
```

Пример:

```
Paint p = Paint.valueOf("#ff0000");
System.out.println(p.isOpaque()); // true
Paint p2 = Paint.valueOf("rgba(255,0,0,0.5)");
System.out.println(p2.isOpaque()); // false
```

5.3. Класс *Color*: цвет

Класс *Color* описывает цвет. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.paint.Color;
```

Иерархия наследования:

Object — Paint — Color

Формат конструктора класса *Color*:

```
Color(double red, double green, double blue, double opacity)
```

В качестве значений указывается вещественное число от 0.0 до 1.0. В параметрах *red*, *green* и *blue* значение задает долю красной, зеленой и синей составляющих цвета соответственно. Параметр *opacity* задает степень непрозрачности (значение 0.0 — полностью прозрачный цвет, а значение 1.0 — полностью непрозрачный). Пример указания синего цвета:

```
Rectangle rect = new Rectangle(400.0, 50.0);
rect.setFill(new Color(0.0, 0.0, 1.0, 1.0));
```

Вместо указания точных значений можно воспользоваться многочисленными статическими константами, например: *BLACK*, *WHITE*, *RED*, *GREEN*, *BLUE*, *YELLOW* и др. Если необходимо задать прозрачный цвет, то следует воспользоваться константой *TRANSPARENT*. Полный список констант смотрите в документации. Пример указания красного цвета:

```
Rectangle rect2 = new Rectangle(400.0, 50.0);
rect2.setFill(Color.RED);
```

Создать объект цвета позволяют также следующие статические методы из класса *Color*:

- *color()* — значения указываются в виде вещественных чисел от 0.0 до 1.0, как и в конструкторе класса *Color*. Форматы метода:

```
public static Color color(double red, double green,
                           double blue)
public static Color color(double red, double green,
                           double blue, double opacity)
```

Пример:

```
Rectangle rect3 = new Rectangle(400.0, 50.0);
rect3.setFill(Color.color(0.0, 1.0, 0.0));
Rectangle rect4 = new Rectangle(400.0, 50.0);
rect4.setFill(Color.color(0.0, 1.0, 0.0, 0.5));
```

- *rgb()* — значения составляющих цвета указываются в виде целых чисел от 0 до 255, а значение непрозрачности — в виде вещественного числа от 0.0 до 1.0. Форматы метода:

```
public static Color rgb(int red, int green, int blue)
public static Color rgb(int red, int green, int blue,
                        double opacity)
```

Пример:

```
Rectangle rect5 = new Rectangle(400.0, 50.0);
rect5.setFill(Color.rgb(255, 255, 0));
Rectangle rect6 = new Rectangle(400.0, 50.0);
rect6.setFill(Color.rgb(255, 255, 0, 0.5));
```

- ❑ **hsb()** — цвет на основе составляющих цветовой модели HSB. Параметр **hue** (значение от 0 до 360) задает цветовой тон, параметр **saturation** (значение от 0.0 до 1.0) — насыщенность, параметр **brightness** (значение от 0.0 до 1.0) — яркость, а параметр **opacity** (значение от 0.0 до 1.0) — степень непрозрачности. **Форматы метода:**

```
public static Color hsb(double hue, double saturation,
                        double brightness)
public static Color hsb(double hue, double saturation,
                        double brightness, double opacity)
```

Пример:

```
Rectangle rect7 = new Rectangle(400.0, 50.0);
rect7.setFill(Color.hsb(0.0, 1.0, 1.0));
Rectangle rect8 = new Rectangle(400.0, 50.0);
rect8.setFill(Color.hsb(0.0, 1.0, 1.0, 0.5));
```

- ❑ **web()** и **valueOf()** — позволяют указать цвет в виде строки. Например: "white", "0xffffffff", "#ffffff", "#fff", "rgb(255, 255, 255)", "rgb(100%, 100%, 100%)", "rgba(255, 255, 255, 0.5)", "hsl(0, 0%, 100%)", "hsla(0, 0%, 100%, 0.5)". **Форматы методов:**

```
public static Color web(String colorString)
public static Color web(String colorString, double opacity)
public static Color valueOf(String value)
```

Пример:

```
Rectangle rect11 = new Rectangle(400.0, 50.0);
rect11.setFill(Color.web("#ffffff"));
Rectangle rect18 = new Rectangle(400.0, 50.0);
rect18.setFill(Color.web("hsl(0, 0%, 100%)", 0.5));
Rectangle rect19 = new Rectangle(400.0, 50.0);
rect19.setFill(Color.valueOf("silver"));
```

- ❑ **gray()** — создает серый цвет. Значение параметра **gray** указывается в виде вещественного числа от 0.0 (черный цвет) до 1.0 (белый цвет). **Форматы метода:**

```
public static Color gray(double gray)
public static Color gray(double gray, double opacity)
```

Пример:

```
Rectangle rect21 = new Rectangle(400.0, 50.0);
rect21.setFill(Color.gray(0.5));
Rectangle rect23 = new Rectangle(400.0, 50.0);
rect23.setFill(Color.gray(1.0, 0.5));
```

- **grayRgb()** — создает серый цвет. Значение параметра `gray` указывается в виде целого числа от 0 (черный цвет) до 255 (белый цвет). Форматы метода:

```
public static Color grayRgb(int gray)
public static Color grayRgb(int gray, double opacity)
```

Пример:

```
Rectangle rect25 = new Rectangle(400.0, 50.0);
rect25.setFill(Color.grayRgb(128));
Rectangle rect27 = new Rectangle(400.0, 50.0);
rect27.setFill(Color.grayRgb(255, 0.5));
```

Класс Color содержит следующие методы:

- **getRed(), getGreen() и getBlue()** — возвращают значения (от 0.0 до 1.0) компонентов цвета в цветовой модели RGB. Форматы методов:

```
public final double getRed()
public final double getGreen()
public final double getBlue()
```

Пример:

```
Color c = Color.rgb(255, 128, 0);
System.out.println(c.getRed()); // 1.0
System.out.println(c.getGreen()); // 0.501960813999176
System.out.println(c.getBlue()); // 0.0
System.out.println((int) (c.getRed() * 255)); // 255
System.out.println((int) (c.getGreen() * 255)); // 128
System.out.println((int) (c.getBlue() * 255)); // 0
```

- **getHue(), getSaturation() и getBrightness()** — возвращают значения (цветовой тон — от 0.0 до 360.0, насыщенность и яркость — от 0.0 до 1.0) компонентов цвета в цветовой модели HSB. Форматы методов:

```
public double getHue()
public double getSaturation()
public double getBrightness()
```

Пример:

```
Color c2 = Color.hsb(30.0, 1.0, 1.0);
System.out.println(c2.getHue()); // 30.0
System.out.println(c2.getSaturation()); // 1.0
System.out.println(c2.getBrightness()); // 1.0
```

- ❑ `getOpacity()` — возвращает степень непрозрачности (вещественное число от 0.0 до 1.0). Формат метода:

```
public final double getOpacity()
```

- ❑ `toString()` — возвращает текстовое представление цвета в цветовой модели RGB с альфа-каналом. Формат метода:

```
public String toString()
```

Пример:

```
Color c3 = Color.rgb(255, 128, 0);  
System.out.println(c3.toString());    // 0xff8000ff
```

- ❑ `equals()` — сравнивает два объекта и возвращает значение `true`, если цвет совпадает. Формат метода:

```
public boolean equals(Object obj)
```

- ❑ `brighter()` — создает более яркий цвет. Формат метода:

```
public Color brighter()
```

- ❑ `darker()` — создает более темный цвет. Формат метода:

```
public Color darker()
```

Пример:

```
System.out.println(Color.rgb(255, 128, 0).darker());  
// 0xb25a00ff
```

- ❑ `saturate()` — создает более насыщенный цвет. Формат метода:

```
public Color saturate()
```

- ❑ `desaturate()` — создает менее насыщенный цвет. Формат метода:

```
public Color desaturate()
```

Пример:

```
System.out.println(Color.rgb(255, 128, 0).desaturate());  
// 0xffa64dff
```

- ❑ `invert()` — инвертирует цвет. Формат метода:

```
public Color invert()
```

Пример:

```
System.out.println(Color.rgb(255, 128, 0).invert());  
// 0x007fffff
```

- ❑ `grayscale()` — преобразует цвет в серые тона. Формат метода:

```
public Color grayscale()
```

Пример:

```
System.out.println(Color.rgb(255, 128, 0).grayscale());  
// 0x909090ff
```


- `interpolate()` — создает промежуточные цвета между текущим цветом и цветом `endValue` в зависимости от значения параметра `t` (вещественное число от 0.0 до 1.0). Метод объявлен в интерфейсе `Interpolatable<Color>`. Формат метода:

```
public Color interpolate(Color endValue, double t)
```

Пример создания промежуточных цветов между черным и белым цветом:

```
Color b = Color.rgb(0, 0, 0);
Color w = Color.rgb(255, 255, 255);
System.out.println(b.interpolate(w, 0.2)); // 0x333333ff
System.out.println(b.interpolate(w, 0.5)); // 0x808080ff
System.out.println(b.interpolate(w, 0.8)); // 0xccccccff
System.out.println(b.interpolate(w, 1.0)); // 0xffffffff
```

- `deriveColor()` — создает новый цвет, изменяя параметры текущего цвета. Формат метода:

```
public Color deriveColor(
    double hueShift, double saturationFactor,
    double brightnessFactor, double opacityFactor)
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH05_03_Color`, приведенном в каталоге `CH05_graphics` сопровождающего книгу электронного архива (см. приложение).

5.4. Класс *LinearGradient*: линейный градиент

Класс `LinearGradient` описывает линейный градиент. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.paint.LinearGradient;
```

Иерархия наследования:

`Object` — `Paint` — `LinearGradient`

Форматы конструктора класса `LinearGradient`:

```
LinearGradient(double startX, double startY,
               double endX, double endY, boolean proportional,
               CycleMethod cycleMethod, List<Stop> stops)
LinearGradient(double startX, double startY,
               double endX, double endY, boolean proportional,
               CycleMethod cycleMethod, Stop... stops)
```

Параметры `startX` и `startY` задают начальные координаты, а параметры `endX` и `endY` — конечные координаты. Если в параметре `proportional` указано значение `true`, то координаты задаются в относительных единицах от 0.0 до 1.0, где координата 0.0 совпадает с началом области, а координата 1.0 — с концом. Пример заливки всего прямоугольника линейным градиентом от черного до белого цвета по горизонтали:

```
Stop[] stops = new Stop[] {  
    new Stop(0.0, Color.BLACK), new Stop(1.0, Color.WHITE)  
};  
Rectangle rect = new Rectangle(400.0, 50.0);  
rect.setStroke(Color.BLACK);  
rect.setFill(new LinearGradient(0.0, 0.0, 1.0, 0.0, true,  
    CycleMethod.NO_CYCLE, stops));
```

Если в параметре `proportional` указано значение `false`, то координаты задаются в системе координат области. Пример заливки прямоугольника линейным градиентом от черного до белого цвета по горизонтали от начала области до позиции 200.0 px по оси X:

```
Rectangle rect7 = new Rectangle(400.0, 50.0);  
rect7.setStroke(Color.BLACK);  
rect7.setFill(new LinearGradient(0.0, 0.0, 200.0, 0.0, false,  
    CycleMethod.NO_CYCLE, stops));
```

Параметр `cycleMethod` задает режим повтора заливки (рис. 5.1). В качестве значения можно указать следующие константы из перечисления `CycleMethod`:

```
import javafx.scene.paint.CycleMethod;
```

☐ `NO_CYCLE` — без повтора, оставшаяся область заливается конечным цветом. Формат:

```
public static final CycleMethod NO_CYCLE
```

☐ `REPEAT` — повтор градиента. Формат:

```
public static final CycleMethod REPEAT
```

☐ `REFLECT` — повтор градиента в обратном порядке. Формат:

```
public static final CycleMethod REFLECT
```

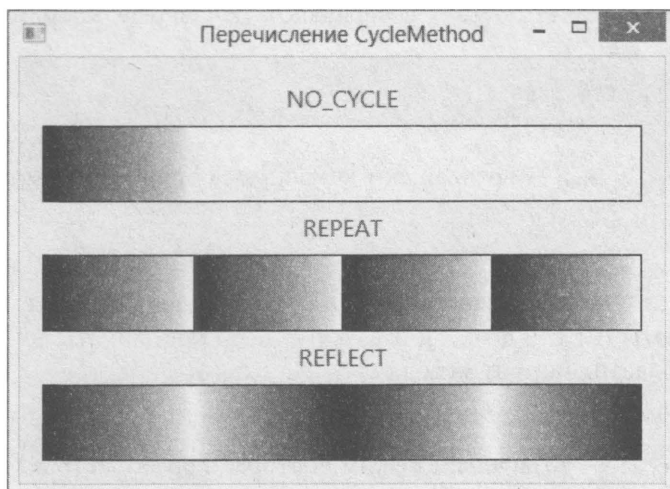


Рис. 5.1. Режимы повтора заливки линейным градиентом

Цвет и положение точек на шкале градиента задаются с помощью параметра `stops`. В качестве значения можно указать список с экземплярами класса `Stop`, массив или просто привести объекты через запятую. Формат конструктора класса `Stop`:

```
import javafx.scene.paint.Stop;
Stop(double offset, Color color)
```

В первом параметре указывается относительное положение точки (от 0.0 до 1.0) на шкале градиента, а во втором параметре — цвет. Пример указания списка из трех точек:

```
List<Stop> stops2 = new ArrayList<Stop>();
stops2.add(new Stop(0.0, Color.RED));
stops2.add(new Stop(0.5, Color.GREEN));
stops2.add(new Stop(1.0, Color.BLUE));
Rectangle rect10 = new Rectangle(400.0, 50.0);
rect10.setStroke(Color.BLACK);
rect10.setFill(new LinearGradient(0.0, 0.0, 1.0, 0.0, true,
                                CycleMethod.NO_CYCLE, stops2));
```

Создать объект линейного градиента на основе строки специального формата позволяет статический метод `valueOf()`. Формат метода:

```
public static LinearGradient valueOf(String value)
```

Пример:

```
Rectangle rect11 = new Rectangle(400.0, 50.0);
rect11.setStroke(Color.BLACK);
rect11.setFill(LinearGradient.valueOf(
    "linear-gradient(from 0% 0% to 50% 0%, reflect, " +
    "#000000 0%, #ffffff 100%)"));
```

Приведем основные методы класса `LinearGradient`:

❑ `getStartX()` и `getStartY()` — возвращают начальные координаты. Форматы методов:

```
public final double getStartX()
public final double getStartY()
```

❑ `getEndX()` и `getEndY()` — возвращают конечные координаты. Форматы методов:

```
public final double getEndX()
public final double getEndY()
```

❑ `isProportional()` — возвращает значение `true`, если используются относительные координаты (от 0.0 до 1.0), и `false` — если координаты указаны в системе координат области. Формат метода:

```
public final boolean isProportional()
```

❑ `getCycleMethod()` — возвращает режим повтора. Формат метода:

```
public final CycleMethod getCycleMethod()
```

❑ `getStops()` — возвращает список с точками. Формат метода:

```
public final List<Stop> getStops()
```

Пример перебора всех точек:

```
Stop[] stops = new Stop[] {  
    new Stop(0.0, Color.BLACK), new Stop(1.0, Color.WHITE)  
};  
LinearGradient lg = new LinearGradient(0.0, 0.0, 1.0, 0.0,  
    true, CycleMethod.NO_CYCLE, stops);  
lg.getStops().stream().forEachOrdered(s -> {  
    System.out.println(s.getOffset() + " " + s.getColor());  
});
```

❑ `toString()` — возвращает строку специального формата. Формат метода:

```
public String toString()
```

Пример такой строки:

```
System.out.println(lg.toString());  
// linear-gradient(from 0.0% 0.0% to 100.0% 0.0%,  
// 0x000000ff 0.0%, 0xffffffff 100.0%)
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH05_04_LinearGradient`, приведенном в каталоге `CH05_graphics` сопровождающего книгу электронного архива (см. приложение).

5.5. Класс *RadialGradient*: радиальный градиент

Класс `RadialGradient` описывает радиальный градиент. Инструкция импорта:

```
// Модуль javafx.graphics  
import javafx.scene.paint.RadialGradient;
```

Иерархия наследования:

```
Object — Paint — RadialGradient
```

Форматы конструктора класса `RadialGradient`:

```
RadialGradient(double focusAngle, double focusDistance,  
    double centerX, double centerY, double radius,  
    boolean proportional, CycleMethod cycleMethod,  
    Stop... stops)
```

```
RadialGradient(double focusAngle, double focusDistance,  
    double centerX, double centerY, double radius,  
    boolean proportional, CycleMethod cycleMethod,  
    List<Stop> stops)
```

Радиальный градиент представляет собой круг радиусом `radius` с координатами центральной точки: `centerX` и `centerY`. Если в параметре `proportional` указано значение `false`, то координаты и радиус задаются в системе координат области (в пикселах). Пример:

```
Stop[] stops = new Stop[] {
    new Stop(0.0, Color.BLACK), new Stop(1.0, Color.WHITE)
};
Rectangle rect = new Rectangle(200.0, 200.0);
rect.setStroke(Color.BLACK);
rect.setFill(new RadialGradient(0.0, 0.0, 100.0, 100.0, 100.0, false,
    CycleMethod.NO_CYCLE, stops));
```

Если в параметре `proportional` указано значение `true`, то координаты и радиус задаются в относительных единицах от 0.0 до 1.0, где координата 0.5 совпадает с центром области. Пример вывода круга с градиентом по центру квадрата:

```
Rectangle rect2 = new Rectangle(200.0, 200.0);
rect2.setStroke(Color.BLACK);
rect2.setFill(new RadialGradient(0.0, 0.0, 0.5, 0.5, 0.5, true,
    CycleMethod.NO_CYCLE, stops));
```

Внутри круга располагается точка фокусировки. Цвет этой точки будет совпадать с первым цветом. Смещение точки фокусировки относительно центра круга задается параметром `focusDistance`. Значение 0.0 совпадает с центром круга, а значение 1.0 смещает точку на границу круга. Угол, на который смещается точка фокусировки, задается параметром `focusAngle`. Значение указывается в градусах от 0.0 до 360.0. Пример:

```
Rectangle rect7 = new Rectangle(200.0, 200.0);
rect7.setStroke(Color.BLACK);
rect7.setFill(new RadialGradient(240.0, 0.8, 0.3, 0.3, 0.1, true,
    CycleMethod.NO_CYCLE, stops));
```

Параметр `cycleMethod` задает режим повтора заливки. В качестве значения можно указать следующие константы из перечисления `CycleMethod`:

```
import javafx.scene.paint.CycleMethod;
```

☐ `NO_CYCLE` — без повтора, оставшаяся область заливается конечным цветом. Формат:

```
public static final CycleMethod NO_CYCLE
```

☐ `REPEAT` — повтор градиента. Формат:

```
public static final CycleMethod REPEAT
```

☐ `REFLECT` — повтор градиента в обратном порядке. Формат:

```
public static final CycleMethod REFLECT
```

Цвет и положение точек на шкале градиента задаются с помощью параметра `stops`. В качестве значения можно указать список с экземплярами класса `Stop`, массив или просто привести объекты через запятую. Формат конструктора класса `Stop`:

```
import javafx.scene.paint.Stop;
Stop(double offset, Color color)
```

В первом параметре указывается относительное положение точки (от 0.0 до 1.0) на шкале градиента, а во втором параметре — цвет. Пример указания списка из трех точек:

```
List<Stop> stops2 = new ArrayList<Stop>();
stops2.add(new Stop(0.0, Color.RED));
stops2.add(new Stop(0.5, Color.GREEN));
stops2.add(new Stop(1.0, Color.BLUE));
Rectangle rect10 = new Rectangle(200.0, 200.0);
rect10.setStroke(Color.BLACK);
rect10.setFill(new RadialGradient(0.0, 0.0, 0.5, 0.5, 0.5, true,
                                CycleMethod.NO_CYCLE, stops2));
```

Создать объект радиального градиента на основе строки специального формата позволяет статический метод `valueOf()`. Формат метода:

```
public static RadialGradient valueOf(String value)
```

Пример:

```
Rectangle rect11 = new Rectangle(200.0, 200.0);
rect11.setStroke(Color.BLACK);
rect11.setFill(RadialGradient.valueOf(
    "radial-gradient(center 100px 100px, radius 100px, red 0%, " +
    "green 50%, blue 100%)");
```

Приведем основные методы класса `RadialGradient`:

❑ `getCenterX()` и `getCenterY()` — возвращают координаты центра круга с градиентом в относительных или абсолютных единицах. Форматы методов:

```
public final double getCenterX()
public final double getCenterY()
```

❑ `getRadius()` — возвращает радиус круга с градиентом в относительных или абсолютных единицах. Формат метода:

```
public final double getRadius()
```

❑ `isProportional()` — возвращает значение `true`, если используются относительные единицы (от 0.0 до 1.0), и `false` — если абсолютные (в пикселях). Формат метода:

```
public final boolean isProportional()
```

❑ `getFocusDistance()` — возвращает смещение точки фокусировки. Формат метода:

```
public final double getFocusDistance()
```

❑ `getFocusAngle()` — возвращает угол, на который смещается точка фокусировки. Формат метода:

```
public final double getFocusAngle()
```

- ❑ `getCycleMethod()` — возвращает режим повтора. Формат метода:

```
public final CycleMethod getCycleMethod()
```

- ❑ `getStops()` — возвращает список с точками. Формат метода:

```
public final List<Stop> getStops()
```

Пример перебора всех точек:

```
Stop[] stops = new Stop[] {
    new Stop(0.0, Color.BLACK), new Stop(1.0, Color.WHITE)
};
RadialGradient rg = new RadialGradient(
    240.0, 1.0, 100.0, 100.0, 100.0, false,
    CycleMethod.NO_CYCLE, stops);
rg.getStops().stream().forEachOrdered(s -> {
    System.out.println(s.getOffset() + " " + s.getColor());
});
```

- ❑ `toString()` — возвращает строку специального формата. Формат метода:

```
public String toString()
```

Пример такой строки:

```
System.out.println(rg.toString());
// radial-gradient(focus-angle 240.0deg, focus-distance 100.0%,
// center 100.0px 100.0px, radius 100.0px,
// 0x000000ff 0.0%, 0xffffffff 100.0%)
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH05_05_RadialGradient`, приведенном в каталоге `CH05_graphics` сопровождающего книгу электронного архива (см. приложение).

5.6. Режимы наложения

Когда мы добавляем узлы в контейнер, они распределяются по слоям. Порядок добавления узлов в контейнер определяет порядок отображения узлов при перекрытии. Первый добавленный узел размещается в самом низу стопки слоев, а последний — на самом ее верху. Изменить порядок следования слоев позволяют следующие методы из класса `Node`:

- ❑ `toBack()` — перемещает слой в самый низ стопки (ссылка на узел будет перемещена в самое начало списка). Формат метода:

```
public void toBack()
```

- ❑ `toFront()` — перемещает слой на самый верх стопки (ссылка на узел будет перемещена в самый конец списка). Формат метода:

```
public void toFront()
```

Если вы когда-нибудь пользовались программой Photoshop, то вспомните, что для каждого слоя с изображениями программа позволяла указать режим наложения. От выбранного режима зависит, как будут взаимодействовать пиксели на разных слоях при наложении друг на друга. JavaFX также позволяет указать режим наложения для каждого узла. Для управления режимами наложения для узла предназначены следующие методы из класса `Node`:

```
public final void setBlendMode(BlendMode value)
public final BlendMode getBlendMode()
public final ObjectProperty<BlendMode> blendModelProperty()
```

В качестве значения параметра `value` метод `setBlendMode()` принимает константы из перечисления `BlendMode`. Инструкция импорта:

```
import javafx.scene.effect.BlendMode;
```

Пример указания режима для прямоугольника:

```
Rectangle rect = new Rectangle(100.0, 100.0);
rect.setBlendMode(BlendMode.SRC_OVER);
```

Основные режимы: `SRC_OVER` и `SRC_ATOP`. Режим по умолчанию — `SRC_OVER` — определяет, что пиксели нижнего слоя полностью перекрываются пикселями верхнего слоя. Как только мы меняем значение непрозрачности или режим наложения, пиксели на разных слоях начинают взаимодействовать друг с другом, а не просто перекрывать друг друга. Это дает возможность создавать весьма интересные эффекты. Остальные режимы можно разделить на несколько групп:

- ☐ группа компонентов цвета — включает режимы `RED`, `GREEN` и `BLUE`;
- ☐ группа затемнения — включает режимы `DARKEN` (результат противоположен режиму `LIGHTEN`), `MULTIPLY` (результат противоположен режиму `SCREEN`) и `COLOR_BURN`. Если верхний слой содержит только белые пиксели, то содержимое слоя станет прозрачным;
- ☐ группа осветления — включает режимы `LIGHTEN` (результат противоположен режиму `DARKEN`), `SCREEN` (результат противоположен режиму `MULTIPLY`), `COLOR_DODGE` и `ADD`. Если верхний слой содержит только черные пиксели, то содержимое слоя станет прозрачным;
- ☐ группа увеличения контрастности — включает режимы `OVERLAY`, `SOFT_LIGHT` и `HARD_LIGHT`. Если верхний слой залит 50-процентным серым цветом, то содержимое слоя станет прозрачным;
- ☐ группа сравнения — включает режимы `DIFFERENCE` (разница) и `EXCLUSION` (исключение).

ЭЛЕКТРОННЫЙ АРХИВ

Увидеть результат применения этих режимов можно в проекте `FX_CH05_06_BlendMode`, приведенном в каталоге `CH05_graphics` сопровождающего книгу электронного архива (см. приложение).

5.7. Маски слоя

Помимо использования режима наложения, можно ограничить видимость слоя с помощью *маски*. Часть слоя, расположенная внутри области маски, будет видна, а оставшаяся часть — скрыта. В качестве маски можно использовать узел, например прямоугольник или компонент `ImageView`. Управлять масками позволяют следующие методы из класса `Node`:

```
public final void setClip(Node value)
public final Node getClip()
public final ObjectProperty<Node> clipProperty()
```

Пример установки маски в виде прямоугольника со скругленными углами для панели:

```
Pane pane = new Pane();
Rectangle mask = new Rectangle(540.0, 340.0);
mask.relocate(30.0, 30.0);
mask.setArcWidth(20);
mask.setArcHeight(20);
pane.setClip(mask);
```

Пример использования изображения в формате PNG в качестве маски для панели:

```
Pane pane = new Pane();
ImageView iv = new ImageView();
try {
    Image img = new Image(
        getClass().getResourceAsStream("/img/mask.png"));
    iv.setImage(img);
    pane.setClip(iv);
} catch (Exception e) {
    System.out.println("Не удалось загрузить изображение");
}
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH05_07_clip`, приведенном в каталоге `CH05_graphics` сопровождающего книгу электронного архива (см. *приложение*).

5.8. Класс *Shape*

Абстрактный класс `Shape` является базовым для большинства классов фигур — таких как `Line` (прямая линия), `Rectangle` (прямоугольник), `Circle` (круг), `Text` (текст) и др. Все эти классы мы рассмотрим далее в этой главе. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.shape.Shape;
```

Иерархия наследования:

`Object` — `Node` — `Shape`

Как видно из иерархии, класс `Shape` является наследником класса `Node`, а следовательно, наследует и все его методы. С помощью этих методов можно назначить обработчики событий, указать местоположение фигуры, задать степень непрозрачности и др. Фигура является узлом, поэтому можно использовать объекты фигур везде, где ожидается класс `Node`.

Так как класс `Shape` является базовым для большинства классов фигур, мы можем объявить переменную с классом `Shape` и хранить в ней ссылки на любые производные классы фигур:

```
Shape rect = new Rectangle(100.0, 100.0);  
Shape line = new Line(0.0, 0.0, 100.0, 0.0);
```

В классе `Shape` определены следующие статические методы:

- ❑ `union()` — возвращает фигуру, которая является результатом объединения фигур `shape1` и `shape2`. Формат метода:

```
public static Shape union(Shape shape1, Shape shape2)
```
- ❑ `subtract()` — возвращает фигуру, которая является результатом вычитания фигуры `shape2` из фигуры `shape1`. Формат метода:

```
public static Shape subtract(Shape shape1, Shape shape2)
```
- ❑ `intersect()` — возвращает фигуру, которая является результатом пересечения фигур `shape1` и `shape2`. Формат метода:

```
public static Shape intersect(Shape shape1, Shape shape2)
```

ПРИМЕЧАНИЕ

При использовании программы `Scene Builder` все фигуры можно найти на вкладке `Shapes`.

5.8.1. Изменение характеристик фона фигуры

Указать характеристики фона фигуры позволяют следующие методы из класса `Shape`:

- ❑ `setFill()` — задает характеристики фона. Формат метода:

```
public final void setFill(Paint value)
```
- ❑ `getFill()` — возвращает характеристики фона. Формат метода:

```
public final Paint getFill()
```
- ❑ `fillProperty()` — возвращает ссылку на свойство, которое связано с характеристиками фона. Формат метода:

```
public final ObjectProperty<Paint> fillProperty()
```

Если характеристики фона не заданы, то используется черный цвет. В качестве значения параметра `value` метода `setFill()` можно указать:

- ❑ **объект класса Color** — задает сплошной цвет фона. Пример:

```
Shape rect = new Rectangle(200.0, 200.0);
System.out.println(rect.getFill()); // 0x000000ff
rect.setFill(Color.RED);
System.out.println(rect.getFill()); // 0xff0000ff
```

- ❑ **объект класса LinearGradient** — заливка линейным градиентом. Пример:

```
Shape rect2 = new Rectangle(200.0, 200.0);
Stop[] stops = new Stop[] {
    new Stop(0, Color.BLACK), new Stop(1, Color.WHITE)
};
rect2.setFill(new LinearGradient(0, 0, 1, 0, true,
                                CycleMethod.NO_CYCLE, stops));
```

- ❑ **объект класса RadialGradient** — радиальный градиент. Пример:

```
Shape rect3 = new Rectangle(200.0, 200.0);
rect3.setFill(new RadialGradient(240.0, 0.7, 0.5, 0.5, 0.5,
                                true, CycleMethod.REFLECT, stops));
```

- ❑ **объект класса ImagePattern** — устанавливает фоновое изображение. Пример:

```
Shape rect4 = new Rectangle(200.0, 200.0);
try {
    ImagePattern im = new ImagePattern(new Image(
        getClass().getResourceAsStream("/img/bg_image.jpg")));
    rect4.setFill(im);
} catch (Exception e) {
    System.out.println("Не удалось загрузить изображение");
}
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH05_08_Shape`, приведенном в каталоге `CH05_graphics` сопровождающего книгу электронного архива (см. приложение).

5.8.2. Изменение характеристик обводки

Управлять характеристиками обводки фигуры позволяют следующие свойства из класса `Shape`:

- ❑ **stroke** — цвет обводки. По умолчанию обводка не отображается, и свойство имеет значение `null`. Методы доступа:

```
public final void setStroke(Paint value)
public final Paint getStroke()
public final ObjectProperty<Paint> strokeProperty()
```

В качестве значения параметра `value` можно указать сплошной цвет (класс `Color`), линейный градиент (класс `LinearGradient`), радиальный градиент (класс `RadialGradient`) или изображение (класс `ImagePattern`). Пример:

```
Shape rect = new Rectangle(200.0, 200.0);
rect.setFill(Color.WHITE);
rect.setStroke(Color.web("#0000ff"));
```

- **strokeWidth** — ширина (толщина) обводки. Значение по умолчанию: 1.0. Методы доступа:

```
public final void setStrokeWidth(double value)
public final double getStrokeWidth()
public final DoubleProperty strokeWidthProperty()
```

Пример:

```
rect.setStrokeWidth(5.0);
```

- **strokeType** — положение обводки относительно границы фигуры. Значение по умолчанию: `StrokeType.CENTERED`. Методы доступа:

```
public final void setStrokeType(StrokeType value)
public final StrokeType getStrokeType()
public final ObjectProperty<StrokeType> strokeTypeProperty()
```

В качестве значения параметра `value` можно указать следующие константы из перечисления `StrokeType`:

```
import javafx.scene.shape.StrokeType;
```

- `CENTERED` — по центру границы (значение по умолчанию);
- `INSIDE` — внутри фигуры;
- `OUTSIDE` — снаружи фигуры.

Пример:

```
rect7.setStrokeType(StrokeType.OUTSIDE);
```

- **strokeDashOffset** — смещение начала пунктирной обводки. Методы доступа:

```
public final void setStrokeDashOffset(double value)
public final double getStrokeDashOffset()
public final DoubleProperty strokeDashOffsetProperty()
```

По умолчанию обводка выполняется сплошной линией. Чтобы сделать линию пунктирной, нужно вызвать метод `getStrokeDashArray()` и добавить в список значения. Четные индексы списка задают длину штриха, а нечетные — длину пропуска. Формат метода `getStrokeDashArray()`:

```
public final ObservableList<Double> getStrokeDashArray()
```

Пример:

```
Shape rect8 = new Rectangle(200.0, 200.0);
rect8.setFill(Color.WHITE);
rect8.setStroke(Color.web("#000000"));
rect8.setStrokeWidth(5.0);
rect8.getStrokeDashArray().addAll(25.0, 15.0);
```

Следует учитывать, что длина штриха зависит от значения свойства `strokeLineCap`. По умолчанию свойство имеет значение `SQUARE`, а это означает, что к длине штриха будет добавлена толщина линии обводки. Если необходимо, чтобы толщина не добавлялась, то следует использовать значение `BUTT`. Пример:

```
Shape rect10 = new Rectangle(200.0, 200.0);
rect10.setFill(Color.WHITE);
rect10.setStroke(Color.web("#000000"));
rect10.setStrokeWidth(5.0);
rect10.getStrokeDashArray().addAll(5.0, 5.0);
rect10.setStrokeLineCap(StrokeLineCap.BUTT);
rect10.setStrokeDashOffset(5.0);
```

- **strokeLineCap** — форма окончания линии или штриха. Значение по умолчанию: `StrokeLineCap.SQUARE`. Методы доступа:

```
public final void setStrokeLineCap(StrokeLineCap value)
public final StrokeLineCap getStrokeLineCap()
public final ObjectProperty<StrokeLineCap>
    strokeLineCapProperty()
```

В качестве значения параметра `value` можно указать следующие константы из перечисления `StrokeLineCap` (рис. 5.2):

```
import javafx.scene.shape.StrokeLineCap;
```

- `SQUARE` — квадратные концы (прибавляются к длине линии или штриха); значение по умолчанию;
- `BUTT` — концы никак не оформляются;
- `ROUND` — закругленные концы (прибавляются к длине линии или штриха);

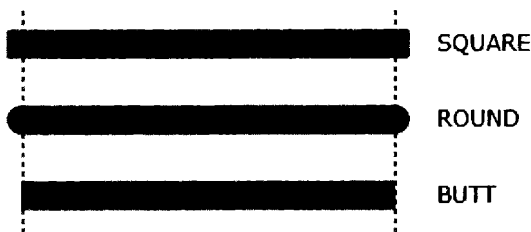


Рис. 5.2. Форма окончания линии или штриха (свойство `strokeLineCap`)

- **strokeLineJoin** — форма окончания в месте соединения двух линий обводки. Значение по умолчанию: `StrokeLineJoin.MITER`. Методы доступа:

```
public final void setStrokeLineJoin(StrokeLineJoin value)
public final StrokeLineJoin getStrokeLineJoin()
public final ObjectProperty<StrokeLineJoin>
    strokeLineJoinProperty()
```

В качестве значения параметра `value` можно указать следующие константы из перечисления `StrokeLineJoin` (рис. 5.3):

```
import javafx.scene.shape.StrokeLineJoin;
```

- `MITER` — обычные углы (значение по умолчанию);
- `BEVEL` — скошенные углы;
- `ROUND` — закругленные углы;



Рис. 5.3. Форма окончания в месте соединения двух линий (свойство `strokeLineJoin`)

- `strokeMiterLimit` — ограничение длины угла при использовании стиля `StrokeLineJoin.MITER`. Значение по умолчанию: 10.0. Если значение превышено, то угол будет скошенным. Методы доступа:

```
public final void setStrokeMiterLimit(double value)
public final double getStrokeMiterLimit()
public final DoubleProperty strokeMiterLimitProperty()
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH05_08_Shape`, приведенном в каталоге `CH05_graphics` сопровождающего книгу электронного архива (см. приложение).

5.8.3. Управление режимом сглаживания

При выводе некоторых фигур (например, круга, треугольника или текста) контур может отображаться в виде «лесенки». Управлять сглаживанием контуров позволяет свойство `smooth` из класса `Shape`. По умолчанию режим сглаживания включен. Методы доступа:

```
public final void setSmooth(boolean value)
public final boolean isSmooth()
public final BooleanProperty smoothProperty()
```

Если в параметре `value` метода `setSmooth()` указать значение `true`, то режим сглаживания будет включен, а если `false` — то выключен. Пример включения режима сглаживания явным образом для треугольника:

```
Polygon shape1 = new Polygon();
shape1.getPoints().addAll(new Double[] {
    0.0, 0.0,
    150.0, 50.0,
    0.0, 100.0
});
```

```
shape1.setFill(Color.WHITE);  
shape1.setStroke(Color.web("#000000"));  
shape1.setStrokeWidth(15.0);  
shape1.setSmooth(true);
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH05_08_Shape`, приведенном в каталоге `CH05_graphics` сопровождающего книгу электронного архива (см. приложение).

5.9. Класс *Line*: прямая линия

Класс `Line` описывает прямую линию, соединяющую две точки. Инструкция импорта:

```
// Модуль javafx.graphics  
import javafx.scene.shape.Line;
```

Иерархия наследования:

`Object` — `Node` — `Shape` — `Line`

Конструкторы класса:

```
Line()  
Line(double startX, double startY, double endX, double endY)
```

Первый конструктор создает линию с нулевыми координатами начала и конца. Пример:

```
Line line = new Line();  
line.setStartX(20.0);  
line.setStartY(20.0);  
line.setEndX(100.0);  
line.setEndY(20.0);  
line.setStroke(Color.RED);
```

Второй конструктор позволяет сразу указать координаты начала (параметры `startX` и `startY`) и конца линии (параметры `endX` и `endY`). Пример:

```
Line line2 = new Line(100.0, 100.0, 200.0, 200.0);  
line2.setStroke(Color.BLUE);  
line2.setStrokeWidth(10.0);  
line2.setStrokeLineCap(StrokeLineCap.ROUND);
```

Класс `Line` наследует все методы из классов `Shape` и `Node`. Об этом не следует забывать. Например, чтобы изменить цвет линии, нужно воспользоваться методом `setStroke()` из класса `Shape`, указать толщину линии позволяет метод `setStrokeWidth()`, а форма концов линии задается с помощью метода `setStrokeLineCap()`. Для создания пунктирной линии используется метод `getStrokeDashArray()` из класса `Shape`. Пример:

```
Line line4 = new Line(20.0, 50.0, 200.0, 50.0);
line4.setStroke(Color.BLACK);
line4.setStrokeWidth(3.0);
line4.getStrokeDashArray().addAll(25.0, 10.0);
line4.setStrokeLineCap(StrokeLineCap.BUTT);
```

Управлять координатами начала и конца линии позволяют следующие свойства:

❑ **startX** — координата начала линии по оси *x*. Методы доступа:

```
public final void setStartX(double value)
public final double getStartX()
public final DoubleProperty startXProperty()
```

❑ **startY** — координата начала линии по оси *y*. Методы доступа:

```
public final void setStartY(double value)
public final double getStartY()
public final DoubleProperty startYProperty()
```

❑ **endX** — координата конца линии по оси *x*. Методы доступа:

```
public final void setEndX(double value)
public final double getEndX()
public final DoubleProperty endXProperty()
```

❑ **endY** — координата конца линии по оси *y*. Методы доступа:

```
public final void setEndY(double value)
public final double getEndY()
public final DoubleProperty endYProperty()
```

ЭЛЕКТРОННЫЙ АРХИВ

Пример использования класса `Line` содержится в проекте `FX_CH05_09_Line`, приведенном в каталоге `CH05_graphics` сопровождающего книгу электронного архива (см. *приложение*).

5.10. Класс *Arc*: дуга или сектор

Класс `Arc` описывает дугу или сектор. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.shape.Arc;
```

Иерархия наследования:

Object — Node — Shape — Arc

Конструкторы класса:

```
Arc()
Arc(double centerX, double centerY, double radiusX, double radiusY,
    double startAngle, double length)
```

Первый конструктор создает объект со значениями параметров по умолчанию.

Пример:


```
Arc arc = new Arc();
arc.setCenterX(110.0);
arc.setCenterY(110.0);
arc.setRadiusX(100.0);
arc.setRadiusY(100.0);
arc.setStartAngle(45.0);
arc.setLength(270.0);
arc.setType(ArcType.OPEN);
```

Второй конструктор в параметрах `centerX` и `centerY` позволяет указать координаты центра окружности, в параметре `radiusX` — радиус окружности по горизонтали, в параметре `radiusY` — радиус окружности по вертикали, в параметре `startAngle` — начальный угол в градусах, а в параметре `length` — длину в градусах, начиная с угла `startAngle`.

Обратите внимание: по умолчанию внутреннее пространство окружности под дугой заливается черным цветом, а обводка не отображается. Чтобы отключить внутреннюю заливку и оставить только дугу, нужно указать значение `Color.TRANSPARENT` с помощью метода `setFill()` и задать цвет обводки с помощью метода `setStroke()`. Пример:

```
Arc arc2 = new Arc(300.0, 110.0, 100.0, 100.0, 45.0, 270.0);
arc2.setFill(Color.TRANSPARENT);
arc2.setStroke(Color.BLACK);
arc2.setStrokeWidth(3.0);
arc2.setType(ArcType.OPEN);
```

Управлять характеристиками фигуры позволяют следующие свойства:

❑ `centerX` — координата центра окружности по оси `x`. Методы доступа:

```
public final void setCenterX(double value)
public final double getCenterX()
public final DoubleProperty centerXProperty()
```

❑ `centerY` — координата центра окружности по оси `y`. Методы доступа:

```
public final void setCenterY(double value)
public final double getCenterY()
public final DoubleProperty centerYProperty()
```

❑ `radiusX` — радиус окружности по горизонтали. Методы доступа:

```
public final void setRadiusX(double value)
public final double getRadiusX()
public final DoubleProperty radiusXProperty()
```

❑ `radiusY` — радиус окружности по вертикали. Методы доступа:

```
public final void setRadiusY(double value)
public final double getRadiusY()
public final DoubleProperty radiusYProperty()
```

- ❑ **startAngle** — начальный угол в градусах. Нулевой угол находится в позиции трех часов. Значения угла отсчитываются против часовой стрелки. Методы доступа:

```
public final void setStartAngle(double value)
public final double getStartAngle()
public final DoubleProperty startAngleProperty()
```

- ❑ **length** — длина дуги в градусах, начиная с угла **startAngle**. Методы доступа:

```
public final void setLength(double value)
public final double getLength()
public final DoubleProperty lengthProperty()
```

- ❑ **type** — тип дуги. Значение по умолчанию: **ArcType.OPEN**. Методы доступа:

```
public final void setType(ArcType value)
public final ArcType getType()
public final ObjectProperty<ArcType> typeProperty()
```

В параметре **value** можно указать следующие константы из перечисления **ArcType** (рис. 5.4):

```
import javafx.scene.shape.ArcType;
```

- **OPEN** — концы дуги не соединяются (значение по умолчанию);
- **CHORD** — концы дуги соединяются прямой линией;
- **ROUND** — концы дуги соединяются через центр окружности.

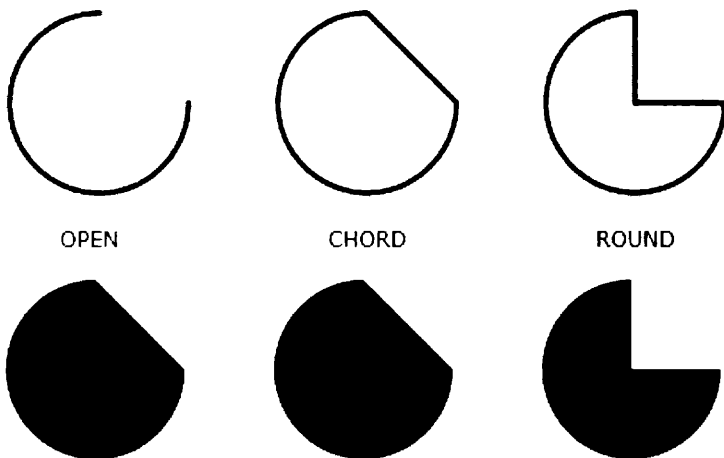


Рис. 5.4. Константы из перечисления **ArcType**

Пример:

```
Arc arc4 = new Arc(110.0, 320.0, 100.0, 100.0, 45.0, 90.0);
arc4.setFill(Color.ANTIQUEWHITE);
arc4.setStroke(Color.BLACK);
```

```
arc4.setStrokeWidth(3.0);
arc4.setType(ArcType.ROUND);
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH05_10_Arc`, приведенном в каталоге `CH05_graphics` сопровождающего книгу электронного архива (см. *приложение*).

5.11. Класс *Polyline*: ломаная линия

Класс `Polyline` описывает ломаную линию. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.shape.Polyline;
```

Иерархия наследования:

Object — Node — Shape — Polyline

Конструкторы класса:

```
Polyline()
Polyline(double... points)
```

Первый конструктор создает пустой объект. Пример создания объекта и добавления координат углов линии:

```
Polyline polyline = new Polyline();
polyline.getPoints().addAll(
    5.0, 5.0,
    100.0, 100.0,
    150.0, 50.0,
    200.0, 100.0
);
```

Второй конструктор позволяет сразу задать координаты углов линии. Значения указываются через запятую или в виде массива. Четные индексы задают координаты по оси *x*, а нечетные — по оси *y*. Пример:

```
Polyline polyline2 = new Polyline(
    5.0, 200.0,
    100.0, 300.0,
    150.0, 220.0,
    200.0, 300.0
);
polyline2.setStroke(Color.RED);
polyline2.setStrokeWidth(5.0);
```

Получить или добавить координаты углов линии позволяет метод `getPoints()`.

Формат метода:

```
public final ObservableList<Double> getPoints()
```

Пример добавления значений с помощью метода `addAll()` из класса `ObservableList<E>`:

```
polyline2.getPoints().addAll(new Double[] {400.0, 100.0});  
polyline2.getPoints().addAll(400.0, 300.0);
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH05_11_Polyline`, приведенном в каталоге `CH05_graphics` сопровождающего книгу электронного архива (см. приложение).

5.12. Класс *CubicCurve*: кубическая кривая Безье

Класс `CubicCurve` описывает кубическую кривую Безье. Инструкция импорта:

```
// Модуль javafx.graphics  
import javafx.scene.shape.CubicCurve;
```

Иерархия наследования:

Object – Node – Shape – CubicCurve

Конструкторы класса:

```
CubicCurve()  
CubicCurve(double startX, double startY,  
            double controlX1, double controlY1,  
            double controlX2, double controlY2,  
            double endX, double endY)
```

Первый конструктор создает объект со значениями по умолчанию. Пример:

```
CubicCurve cubic = new CubicCurve();  
cubic.setStartX(50.0);  
cubic.setStartY(300.0);  
cubic.setEndX(300.0);  
cubic.setEndY(300.0);  
cubic.setControlX1(100.0);  
cubic.setControlY1(100.0);  
cubic.setControlX2(250.0);  
cubic.setControlY2(100.0);  
cubic.setFill(Color.TRANSPARENT);  
cubic.setStroke(Color.DARKGREEN);  
cubic.setStrokeWidth(3.0);
```

Второй конструктор позволяет сразу указать координаты четырех точек: начальной точки (параметры `startX` и `startY`), конечной точки (параметры `endX` и `endY`), первой опорной точки (параметры `controlX1` и `controlY1`) и второй опорной точки (параметры `controlX2` и `controlY2`). Пример:

```
CubicCurve cubic2 = new CubicCurve(10.0, 100.0, 60.0, 20.0,  
                                   110.0, 180.0, 160.0, 100.0);  
cubic2.setFill(Color.TRANSPARENT);
```

```
cubic2.setStroke(Color.DARKGREEN);  
cubic2.setStrokeLineCap(StrokeLineCap.ROUND);  
cubic2.setStrokeWidth(3.0);
```

Управлять характеристиками фигуры позволяют следующие свойства:

❑ **startX** — координата начальной точки по оси *x*. Методы доступа:

```
public final void setStartX(double value)  
public final double getStartX()  
public final DoubleProperty startXProperty()
```

❑ **startY** — координата начальной точки по оси *y*. Методы доступа:

```
public final void setStartY(double value)  
public final double getStartY()  
public final DoubleProperty startYProperty()
```

❑ **endX** — координата конечной точки по оси *x*. Методы доступа:

```
public final void setEndX(double value)  
public final double getEndX()  
public final DoubleProperty endXProperty()
```

❑ **endY** — координата конечной точки по оси *y*. Методы доступа:

```
public final void setEndY(double value)  
public final double getEndY()  
public final DoubleProperty endYProperty()
```

❑ **controlX1** — координата первой опорной точки по оси *x*. Методы доступа:

```
public final void setControlX1(double value)  
public final double getControlX1()  
public final DoubleProperty controlX1Property()
```

❑ **controlY1** — координата первой опорной точки по оси *y*. Методы доступа:

```
public final void setControlY1(double value)  
public final double getControlY1()  
public final DoubleProperty controlY1Property()
```

❑ **controlX2** — координата второй опорной точки по оси *x*. Методы доступа:

```
public final void setControlX2(double value)  
public final double getControlX2()  
public final DoubleProperty controlX2Property()
```

❑ **controlY2** — координата второй опорной точки по оси *y*. Методы доступа:

```
public final void setControlY2(double value)  
public final double getControlY2()  
public final DoubleProperty controlY2Property()
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH05_12_CubicCurve`, приведенном в каталоге `CH05_graphics` сопровождающего книгу электронного архива (см. приложение).

5.13. Класс *QuadCurve*: квадратичная кривая

Класс *QuadCurve* описывает квадратичную кривую. В отличие от класса *CubicCurve*, используется только одна опорная точка вместо двух. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.shape.QuadCurve;
```

Иерархия наследования:

Object — Node — Shape — QuadCurve

Конструкторы класса:

```
QuadCurve()
QuadCurve(double startX, double startY,
           double controlX, double controlY,
           double endX, double endY)
```

Первый конструктор создает объект со значениями по умолчанию. Пример:

```
QuadCurve quad = new QuadCurve();
quad.setStartX(50.0);
quad.setStartY(300.0);
quad.setEndX(300.0);
quad.setEndY(300.0);
quad.setControlX(100.0);
quad.setControlY(100.0);
quad.setFill(Color.TRANSPARENT);
quad.setStroke(Color.DARKGREEN);
quad.setStrokeWidth(3.0);
```

Второй конструктор позволяет сразу указать координаты трех точек: начальной точки (параметры *startX* и *startY*), конечной точки (параметры *endX* и *endY*) и опорной точки (параметры *controlX* и *controlY*). Пример:

```
QuadCurve quad = new QuadCurve(50.0, 300.0, 100.0, 100.0,
                                300.0, 300.0);
quad.setFill(Color.TRANSPARENT);
quad.setStroke(Color.RED);
quad.setStrokeWidth(3.0);
```

Управлять характеристиками фигуры позволяют следующие свойства:

❑ *startX* — координата начальной точки по оси *x*. Методы доступа:

```
public final void setStartX(double value)
public final double getStartX()
public final DoubleProperty startXProperty()
```

❑ *startY* — координата начальной точки по оси *y*. Методы доступа:

```
public final void setStartY(double value)
public final double getStartY()
public final DoubleProperty startYProperty()
```

❑ **endX** — координата конечной точки по оси *x*. Методы доступа:

```
public final void setEndX(double value)
public final double getEndX()
public final DoubleProperty endXProperty()
```

❑ **endY** — координата конечной точки по оси *y*. Методы доступа:

```
public final void setEndY(double value)
public final double getEndY()
public final DoubleProperty endYProperty()
```

❑ **controlX** — координата опорной точки по оси *x*. Методы доступа:

```
public final void setControlX(double value)
public final double getControlX()
public final DoubleProperty controlXProperty()
```

❑ **controlY** — координата опорной точки по оси *y*. Методы доступа:

```
public final void setControlY(double value)
public final double getControlY()
public final DoubleProperty controlYProperty()
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH05_13_QuadCurve`, приведенном в каталоге `CH05_graphics` сопровождающего книгу электронного архива (см. *приложение*).

5.14. Класс *Rectangle*: прямоугольник

Класс `Rectangle` описывает прямоугольник с обычными или закругленными углами.

Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.shape.Rectangle;
```

Иерархия наследования:

```
Object — Node — Shape — Rectangle
```

Конструкторы класса:

```
Rectangle()
Rectangle(double width, double height)
Rectangle(double x, double y, double width, double height)
Rectangle(double width, double height, Paint fill)
```

Первый конструктор создает пустой объект. Пример создания прямоугольника черного цвета без обводки:

```
Rectangle rect = new Rectangle();
rect.setX(10.0);
rect.setY(150.0);
```

```
rect.setWidth(200.0);  
rect.setHeight(100.0);
```

Второй конструктор позволяет указать ширину и высоту прямоугольника. Координаты левого верхнего угла будут равны 0.0. Пример создания красного прямоугольника без обводки:

```
Rectangle rect2 = new Rectangle(200.0, 100.0);  
rect2.setX(10.0);  
rect2.setY(10.0);  
rect2.setFill(Color.RED);
```

Третий конструктор позволяет указать координаты левого верхнего угла, а также ширину и высоту прямоугольника. Пример создания квадрата с заливкой линейным градиентом и черной обводкой шириной 3 px:

```
Rectangle rect3 = new Rectangle(250.0, 150.0, 100.0, 100.0);  
rect3.setFill(LinearGradient.valueOf(  
    "linear-gradient(from 0% 0% to 50% 0%, reflect, " +  
    "#000000 0%, #ffffff 100%)");  
rect3.setStroke(Color.BLACK);  
rect3.setStrokeWidth(3.0);
```

Четвертый конструктор задает ширину и высоту прямоугольника, а также цвет заливки. Координаты левого верхнего угла будут равны 0.0. Пример создания квадрата без заливки с черной обводкой шириной 3 px и со скругленными углами:

```
Rectangle rect4 = new Rectangle(100.0, 100.0, Color.TRANSPARENT);  
rect4.setX(250.0);  
rect4.setY(10.0);  
rect4.setStroke(Color.BLACK);  
rect4.setStrokeWidth(3.0);  
rect4.setArcWidth(20.0);  
rect4.setArcHeight(20.0);
```

Управлять характеристиками фигуры позволяют следующие свойства:

□ **x** — координата левого верхнего угла по оси x. Методы доступа:

```
public final void setX(double value)  
public final double getX()  
public final DoubleProperty xProperty()
```

□ **y** — координата левого верхнего угла по оси y. Методы доступа:

```
public final void setY(double value)  
public final double getY()  
public final DoubleProperty yProperty()
```

□ **width** — ширина прямоугольника. Методы доступа:

```
public final void setWidth(double value)  
public final double getWidth()  
public final DoubleProperty widthProperty()
```


❑ **height** — **высота прямоугольника**. Методы доступа:

```
public final void setHeight(double value)
public final double getHeight()
public final DoubleProperty heightProperty()
```

❑ **arcWidth** — **горизонтальный диаметр дуги для скругления углов**. Методы доступа:

```
public final void setArcWidth(double value)
public final double getArcWidth()
public final DoubleProperty arcWidthProperty()
```

❑ **arcHeight** — **вертикальный диаметр дуги для скругления углов**. Методы доступа:

```
public final void setArcHeight(double value)
public final double getArcHeight()
public final DoubleProperty arcHeightProperty()
```

Обратите внимание: углы будут скруглены, только если оба свойства: **и arcWidth, и arcHeight** — больше 0.

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH05_14_Rectangle`, приведенном в каталоге `CH05_graphics` сопровождающего книгу электронного архива (см. *приложение*).

5.15. Класс *Ellipse*: эллипс

Класс `Ellipse` описывает эллипс. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.shape.Ellipse;
```

Иерархия наследования:

`Object` — `Node` — `Shape` — `Ellipse`

Конструкторы класса:

```
Ellipse()
Ellipse(double radiusX, double radiusY)
Ellipse(double centerX, double centerY,
        double radiusX, double radiusY)
```

Первый конструктор создает пустой объект. Пример:

```
Ellipse ellipse = new Ellipse();
ellipse.setCenterX(110.0);
ellipse.setCenterY(110.0);
ellipse.setRadiusX(100.0);
ellipse.setRadiusY(50.0);
```

Второй конструктор позволяет указать радиус по горизонтали (параметр `radiusX`) и по вертикали (параметр `radiusY`). Координаты центра эллипса будут равны 0. Пример:

```
Ellipse ellipse2 = new Ellipse(50.0, 100.0);
ellipse2.setCenterX(300.0);
ellipse2.setCenterY(110.0);
ellipse2.setFill(Color.RED);
ellipse2.setStroke(Color.BLACK);
ellipse2.setStrokeWidth(3.0);
```

Третий конструктор позволяет указать координаты центра эллипса (параметры `centerX` и `centerY`), а также радиус по горизонтали (параметр `radiusX`) и по вертикали (параметр `radiusY`). Пример создания круга без заливки с обводкой:

```
Ellipse ellipse3 = new Ellipse(110.0, 300.0, 70.0, 70.0);
ellipse3.setFill(Color.TRANSPARENT);
ellipse3.setStroke(Color.CHOCOLATE);
ellipse3.setStrokeWidth(3.0);
```

Управлять характеристиками фигуры позволяют следующие свойства:

☐ `centerX` — координата центра эллипса по оси *x*. Методы доступа:

```
public final void setCenterX(double value)
public final double getCenterX()
public final DoubleProperty centerXProperty()
```

☐ `centerY` — координата центра эллипса по оси *y*. Методы доступа:

```
public final void setCenterY(double value)
public final double getCenterY()
public final DoubleProperty centerYProperty()
```

☐ `radiusX` — радиус по горизонтали. Методы доступа:

```
public final void setRadiusX(double value)
public final double getRadiusX()
public final DoubleProperty radiusXProperty()
```

☐ `radiusY` — радиус по вертикали. Методы доступа:

```
public final void setRadiusY(double value)
public final double getRadiusY()
public final DoubleProperty radiusYProperty()
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH05_15_Ellipse`, приведенном в каталоге `CH05_graphics` сопровождающего книгу электронного архива (см. *приложение*).

5.16. Класс *Circle*: круг

Класс `Circle` описывает круг. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.shape.Circle;
```

Иерархия наследования:

Object — Node — Shape — Circle

Конструкторы класса:

```
Circle()  
Circle(double radius)  
Circle(double centerX, double centerY, double radius)  
Circle(double centerX, double centerY, double radius, Paint fill)  
Circle(double radius, Paint fill)
```

Первый конструктор создает пустой объект. Пример:

```
Circle circle = new Circle();  
circle.setCenterX(110.0);  
circle.setCenterY(110.0);  
circle.setRadius(100.0);
```

Второй конструктор позволяет указать радиус круга. Координаты центра круга будут равны 0. Пример создания красного круга радиусом 100 px с обводкой черного цвета:

```
Circle circle2 = new Circle(100.0);  
circle2.setCenterX(350.0);  
circle2.setCenterY(110.0);  
circle2.setFill(Color.RED);  
circle2.setStroke(Color.BLACK);  
circle2.setStrokeWidth(3.0);
```

Третий конструктор позволяет задать координаты центра круга и его радиус. Пример заливки круга радиальным градиентом:

```
Circle circle3 = new Circle(110.0, 350.0, 70.0);  
Stop[] stops = new Stop[] {  
    new Stop(0.0, Color.LIGHTGRAY), new Stop(1.0, Color.BLACK)  
};  
circle3.setFill(new RadialGradient(240.0, 0.6, 0.5, 0.5, 0.6, true,  
    CycleMethod.NO_CYCLE, stops));
```

Четвертый конструктор дополнительно позволяет указать цвет заливки. Пример создания прозрачного круга с обводкой:

```
Circle circle4 = new Circle(350.0, 350.0, 70.0, Color.TRANSPARENT);  
circle4.setStroke(Color.CHOCOLATE);  
circle4.setStrokeWidth(3.0);
```

В пятом конструкторе указывается радиус круга и цвет заливки. Координаты центра круга будут равны 0. Пример:

```
Circle circle5 = new Circle(50.0, Color.BLUE);  
circle5.setCenterX(240.0);  
circle5.setCenterY(250.0);
```

Управлять характеристиками фигуры позволяют следующие свойства:

❑ `centerX` — координата центра круга по оси `x`. Методы доступа:

```
public final void setCenterX(double value)
public final double getCenterX()
public final DoubleProperty centerXProperty()
```

❑ `centerY` — координата центра круга по оси `y`. Методы доступа:

```
public final void setCenterY(double value)
public final double getCenterY()
public final DoubleProperty centerYProperty()
```

❑ `radius` — радиус. Методы доступа:

```
public final void setRadius(double value)
public final double getRadius()
public final DoubleProperty radiusProperty()
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH05_16_Circle`, приведенном в каталоге `CH05_graphics` сопровождающего книгу электронного архива (см. приложение).

5.17. Класс *Polygon*: многоугольник

Класс `Polygon` описывает многоугольник. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.shape.Polygon;
```

Иерархия наследования:

`Object` — `Node` — `Shape` — `Polygon`

Конструкторы класса:

```
Polygon()
Polygon(double... points)
```

Первый конструктор создает пустой объект. Пример создания треугольника:

```
Polygon polygon = new Polygon();
polygon.getPoints().addAll(
    30.0, 30.0,
    150.0, 80.0,
    30.0, 130.0
);
```

Второй конструктор позволяет сразу задать координаты углов. Значения указываются через запятую или в виде массива. Четные индексы задают координаты по оси `x`, а нечетные — по оси `y`. Каждая точка будет соединена линией с последующей точкой внутри списка. Последняя точка будет соединена линией с первой точкой, образуя замкнутый контур. Если вам не нужен замкнутый контур, то воспользуйтесь классом `Polyline` (см. *разд. 5.11*). Пример создания четырехугольника с заливкой линейным градиентом и обводкой:

```
Polygon polygon2 = new Polygon(
    200.0, 50.0,
    300.0, 10.0,
    300.0, 150.0,
    200.0, 110.0
);
polygon2.setFill(LinearGradient.valueOf(
    "linear-gradient(from 0% 0% to 100% 0%, " +
    "#333333 0%, #eeeeee 100%)"));
polygon2.setStroke(Color.BLACK);
polygon2.setStrokeWidth(3.0);
```

Получить или добавить координаты углов позволяет метод `getPoints()`. Формат метода:

```
public final ObservableList<Double> getPoints()
```

Пример добавления координат с помощью конструктора и метода `addAll()` из класса `ObservableList<E>`:

```
Polygon polygon3 = new Polygon(new double[] {
    100.0, 280.0,
    200.0, 200.0,
    300.0, 280.0,
    300.0, 400.0
});
polygon3.getPoints().addAll(new Double[] {100.0, 400.0});
polygon3.setFill(Color.CHOCOLATE);
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH05_17_Polygon`, приведенном в каталоге `CH05_graphics` сопровождающего книгу электронного архива (см. приложение).

5.18. Класс *Text*: текст

Класс `Text` описывает текст. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.text.Text;
```

Иерархия наследования:

```
Object — Node — Shape — Text
```

5.18.1. Создание объекта

Конструкторы класса `Text`:

```
Text()
Text(String text)
Text(double x, double y, String text)
```

Первый конструктор создает пустой объект. Пример:

```
Text text = new Text();
text.setText("Строка");
text.setX(20.0);
text.setY(20.0);
```

Второй конструктор позволяет указать текст. Координаты текста будут равны 0. Пример отображения текста красного цвета на нескольких строках (символом-разделителем абзацев является символ \n):

```
Text text2 = new Text("Строка1\nСтрока2\nСтрока3");
text2.setX(20.0);
text2.setY(50.0);
text2.setFill(Color.RED);
```

Третий конструктор позволяет сразу указать координаты и текст. Пример указания размера шрифта:

```
Text text3 = new Text(20.0, 150.0, "Строка");
text3.setFill(Color.GREEN);
text3.setFont(new Font(24.0));
```

Управлять характеристиками фигуры позволяют следующие свойства из класса Text:

❑ **text** — отображаемый текст. Методы доступа:

```
public final void setText(String value)
public final String getText()
public final StringProperty textProperty()
```

❑ **x** — координата по оси x. Методы доступа:

```
public final void setX(double value)
public final double getX()
public final DoubleProperty xProperty()
```

❑ **y** — координата по оси y. Методы доступа:

```
public final void setY(double value)
public final double getY()
public final DoubleProperty yProperty()
```

❑ **textOrigin** — определяет начало системы координат. Значение по умолчанию: VPos.BASELINE. Методы доступа:

```
public final void setTextOrigin(VPos value)
public final VPos getTextOrigin()
public final ObjectProperty<VPos> textOriginProperty()
```

В качестве значения параметра value можно указать следующие константы из перечисления VPos:

```
import javafx.geometry.VPos;
```

- TOP — по верху;
- CENTER — по центру;
- BASELINE — по базовой линии (значение по умолчанию);
- BOTTOM — по низу.

Пример смещения начала системы координат в левый верхний угол:

```
Text text4 = new Text(130.0, 300.0, "VPos.TOP");
text4.setTextOrigin(VPos.TOP);
```

- **baselineOffset** — смещение базовой линии относительно значения `layoutBounds.minY`. Методы доступа:

```
public final double getBaselineOffset()
public final ReadOnlyDoubleProperty baselineOffsetProperty()
```

Обратите внимание: метод `getBaselineOffset()` может возвращать значение константы `BASELINE_OFFSET_SAME_AS_HEIGHT` из класса `Node`:

```
System.out.println(Node.BASELINE_OFFSET_SAME_AS_HEIGHT);
// -Infinity
```

- **boundsType** — способ вычисления границ текста. Значение по умолчанию: `TextBoundsType.LOGICAL`. Методы доступа:

```
public final void setBoundsType(TextBoundsType value)
public final TextBoundsType getBoundsType()
public final ObjectProperty<TextBoundsType>
    boundsTypeProperty()
```

В качестве значения параметра `value` можно указать константы `LOGICAL`, `VISUAL` или `LOGICAL_VERTICAL_CENTER` из перечисления `TextBoundsType`. Инструкция импорта:

```
import javafx.scene.text.TextBoundsType;
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH05_18_Text`, приведенном в каталоге `CH05_graphics` сопровождающего книгу электронного архива (см. приложение).

5.18.2. Подчеркивание и зачеркивание текста

Отобразить текст подчеркнутым или зачеркнутым позволяют следующие свойства из класса `Text`:

- **underline** — если свойство имеет значение `true`, то текст будет подчеркнут. Значение по умолчанию: `false`. Методы доступа:

```
public final void setUnderline(boolean value)
public final boolean isUnderline()
public final BooleanProperty underlineProperty()
```

Пример:

```
Text text9 = new Text(30.0, 400.0, "Подчеркнутый текст");
text9.setUnderline(true);
```

- **strikethrough** — если свойство имеет значение `true`, то текст будет перечеркнут. Значение по умолчанию: `false`. Методы доступа:

```
public final void setStrikethrough(boolean value)
public final boolean isStrikethrough()
public final BooleanProperty strikethroughProperty()
```

Пример:

```
Text text10 = new Text(30.0, 450.0, "Перечеркнутый текст");
text10.setStrikethrough(true);
```

Получить траекторию, соответствующую подчеркнутому фрагменту от позиции `start` до `end` (не включая символ с этим индексом), позволяет метод `underlineShape()`. Метод доступен, начиная с JavaFX 9. Формат метода:

```
public final PathElement[] underlineShape(int start, int end)
```

Подчеркнем фрагмент текста красной линией:

```
Text text11 = new Text(250.0, 400.0, "Первая строка\nпвторая строка");
Path path = new Path(text11.underlineShape(0, 20));
path.setStroke(Color.RED);
path.setStrokeWidth(0.5);
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH05_18_Text`, приведенном в каталоге `CH05_graphics` сопровождающего книгу электронного архива (см. приложение).

5.18.3. Класс *Font*: изменение характеристик шрифта

Управлять характеристиками шрифта позволяют следующие свойства из класса `Text`:

- **font** — задает характеристики шрифта. Методы доступа:

```
public final void setFont(Font value)
public final Font getFont()
public final ObjectProperty<Font> fontProperty()
```

Пример указания размера шрифта:

```
Text text = new Text(20.0, 50.0, "Font(24.0)");
text.setFont(new Font(24.0));
```

- **fontSmoothingType** — тип сглаживания шрифта. Можно указать константы `GRAY` (значение по умолчанию) или `LCD` из перечисления `FontSmoothingType`. Методы доступа:


```
import javafx.scene.text.FontSmoothingType;
public final void setFontSmoothingType(FontSmoothingType value)
public final FontSmoothingType getFontSmoothingType()
public final ObjectProperty<FontSmoothingType>
    fontSmoothingTypeProperty()
```

Характеристики шрифта задаются с помощью класса `Font`. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.text.Font;
```

Конструкторы класса `Font`:

```
Font(double size)
Font(String name, double size)
```

Первый конструктор позволяет указать только размер шрифта, используемого по умолчанию. Пример:

```
text.setFont(new Font(24.0));
```

Получить характеристики шрифта, используемого по умолчанию, можно с помощью статического метода `getDefault()`. Формат метода:

```
public static Font getDefault()
```

Пример:

```
System.out.println(Font.getDefault());
// Font[name=System Regular, family=System, style=Regular, size=12.0]
```

Второй конструктор задает название шрифта и его размер. Пример:

```
text2.setFont(new Font("Verdana", 12.0));
```

Получить список всех семейств шрифтов, доступных приложению, можно с помощью статического метода `getFamilies()`. Формат метода:

```
public static List<String> getFamilies()
```

Получить список всех названий шрифтов, доступных приложению, позволяет статический метод `getFontNames()`. Форматы метода:

```
public static List<String> getFontNames()
public static List<String> getFontNames(String family)
```

Если метод `getFamilies()` возвращает только название семейства шрифтов (например, `Verdana`), то метод `getFontNames()` вернет все названия шрифтов внутри всех семейств, например: `Verdana`, `Verdana Bold`, `Verdana Bold Italic`, `Verdana Italic`. Второй формат метода `getFontNames()` позволяет получить список шрифтов указанного семейства. Пример:

```
System.out.println(Font.getFontNames("Verdana"));
// [Verdana, Verdana Italic, Verdana Bold Italic, Verdana Bold]
```

Создать объект шрифта помимо конструкторов позволяют также следующие статические методы из класса `Font`:

- ❑ `loadFont()` и `loadFonts()` — позволяют загрузить шрифт из файла. Метод `loadFonts()` доступен, начиная с JavaFX 9. Форматы методов:

```
public static Font loadFont(String urlStr, double size)
public static Font loadFont(InputStream in, double size)
public static Font[] loadFonts(String urlStr, double size)
public static Font[] loadFonts(InputStream in, double size)
```

- ❑ `font()` — создает объект шрифта с различными настройками. Форматы метода:

```
public static Font font(double size)
public static Font font(String family)
public static Font font(String family, double size)
public static Font font(String family, FontWeight weight,
                        double size)
public static Font font(String family, FontPosture posture,
                        double size)
public static Font font(String family, FontWeight weight,
                        FontPosture posture, double size)
```

Параметр `size` задает размер шрифта, а параметр `family` — название семейства шрифтов. Параметр `posture` определяет, является ли шрифт курсивным. В качестве значения можно указать константы: `REGULAR` (обычный, значение по умолчанию) и `ITALIC` (курсивный) из перечисления `FontPosture`. Инструкция импорта:

```
import javafx.scene.text.FontPosture;
```

Параметр `weight` задает жирность шрифта. В качестве значения можно указать константы: `THIN`, `EXTRA_LIGHT`, `LIGHT`, `NORMAL`, `MEDIUM`, `SEMI_BOLD`, `BOLD`, `EXTRA_BOLD` или `BLACK` из перечисления `FontWeight`. Инструкция импорта:

```
import javafx.scene.text.FontWeight;
```

Пример указания полужирного курсивного шрифта:

```
Text text18 = new Text(20.0, 550.0, "Текст");
text18.setFont(Font.font("Verdana", FontWeight.BOLD,
                        FontPosture.ITALIC, 12.0));
```

Получить информацию об объекте шрифта позволяют следующие методы из класса `Font`:

- ❑ `getFamily()` — возвращает название семейства шрифта. Формат метода:

```
public final String getFamily()
```

Пример:

```
Font font = text18.getFont();
System.out.println(font.getFamily()); // Verdana
```

- ❑ `getName()` — возвращает название шрифта. Формат метода:

```
public final String getName()
```

Пример:

```
System.out.println(font.getName()); // Verdana Bold Italic
```

- ❑ `getSize()` — возвращает размер шрифта. Формат метода:

```
public final double getSize()
```

Пример:

```
System.out.println(font.getSize());    // 12.0
```

- ❑ `getStyle()` — возвращает стиль шрифта. Формат метода:

```
public final String getStyle()
```

Пример:

```
System.out.println(font.getStyle());    // Bold Italic
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH05_18_Text`, приведенном в каталоге `CH05_graphics` сопровождающего книгу электронного архива (см. приложение).

5.18.4. Выравнивание текста

Управлять выравниванием текста позволяют следующие свойства из класса `Text`:

- ❑ `wrappingWidth` — ограничение ширины области с текстом. Если ограничение не задано, то текст будет выводиться на одной строке. Значение по умолчанию: `0.0`. Методы доступа:

```
public final void setWrappingWidth(double value)
public final double getWrappingWidth()
public final DoubleProperty wrappingWidthProperty()
```

- ❑ `textAlignment` — выравнивание текста внутри области ограниченной ширины по горизонтали. Значение по умолчанию: `TextAlignment.LEFT`. Методы доступа:

```
public final void setTextAlignment(TextAlignment value)
public final TextAlignment getTextAlignment()
public final ObjectProperty<TextAlignment>
    textAlignmentProperty()
```

В качестве значения параметра `value` можно указать следующие константы из перечисления `TextAlignment`:

```
import javafx.scene.text.TextAlignment;
```

- `LEFT` — по левому краю области (значение по умолчанию);
- `RIGHT` — по правому краю области;
- `CENTER` — по центру области;
- `JUSTIFY` — по ширине области.

Пример выравнивания длинного текста по ширине области размером 400 px:

```
Text text4 = new Text(20.0, 350.0, longStr);
text4.setWrappingWidth(400.0);
text4.setTextAlignment(TextAlignment.JUSTIFY);
```

- `lineSpacing` — расстояние между строками. Значение по умолчанию: 0.0. Методы доступа:

```
public final void setLineSpacing(double spacing)
public final double getLineSpacing()
public final DoubleProperty lineSpacingProperty()
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH05_18_Text`, приведенном в каталоге `CH05_graphics` сопровождающего книгу электронного архива (см. приложение).

5.18.5. Выделение фрагмента текста

Начиная с JavaFX 9, в классе `Text` доступны следующие свойства, позволяющие управлять выделением фрагмента текста:

- `selectionStart` — начальная позиция выделения. Чтобы снять выделение, нужно указать значение `-1`. Методы доступа:

```
public final void setSelectionStart(int value)
public final int getSelectionStart()
public final IntegerProperty selectionStartProperty()
```

- `selectionEnd` — конечная позиция выделения. Методы доступа:

```
public final void setSelectionEnd(int value)
public final int getSelectionEnd()
public final IntegerProperty selectionEndProperty()
```

- `selectionFill` — цвет текста выделенного фрагмента. Методы доступа:

```
public final void setSelectionFill(Paint paint)
public final Paint getSelectionFill()
public final ObjectProperty<Paint> selectionFillProperty()
```

- `selectionShape` — траектория, описывающая выделенный фрагмент текста. Методы доступа:

```
public final PathElement[] getSelectionShape()
public final ReadOnlyObjectProperty<PathElement[]>
    selectionShapeProperty()
```

Пример выделения фрагмента текста:

```
Text text = new Text(50.0, 50.0, "Первая строка\nвторая строка");
text.setFill(Color.BLACK);
text.setSelectionStart(1);
text.setSelectionEnd(20);
text.setSelectionFill(Color.RED);
```

```
Path path = new Path(text.getSelectionShape());
path.setFill(Color.web("#ffff00", 0.4));
path.setStroke(Color.TRANSPARENT);
```

Получить траекторию, соответствующую фрагменту от позиции `start` до `end` (не включая символ с этим индексом), позволяет метод `rangeShape()`. Метод доступен, начиная с JavaFX 9. Формат метода:

```
public final PathElement[] rangeShape(int start, int end)
```

Выделим фрагмент текста:

```
Text text6 = new Text(50.0, 100.0, "Первая строка\nвторая строка");
Path path6 = new Path(text6.rangeShape(1, 20));
path6.setFill(Color.web("#ffff00", 0.4));
path6.setStroke(Color.TRANSPARENT);
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH05_18_Text`, приведенном в каталоге `CH05_graphics` сопровождающего книгу электронного архива (см. приложение).

5.18.6. Управление положением текстового курсора

Начиная с JavaFX 9, в классе `Text` доступны следующие свойства, позволяющие управлять положением текстового курсора:

- `caretPosition` — позиция текстового курсора (значение `-1`, если позиция не установлена). Методы доступа:

```
public final void setCaretPosition(int value)
public final int getCaretPosition()
public final IntegerProperty caretPositionProperty()
```

- `caretBias` — если свойство имеет значение `true` (значение по умолчанию), то текстовый курсор располагается перед символом, а если `false` — то после символа. Методы доступа:

```
public final void setCaretBias(boolean value)
public final boolean isCaretBias()
public final BooleanProperty caretBiasProperty()
```

- `caretShape` — траектория, описывающая текстовый курсор. Методы доступа:

```
public final PathElement[] getCaretShape()
public final ReadOnlyObjectProperty<PathElement[]>
    caretShapeProperty()
```

Пример:

```
Text text2 = new Text(250.0, 50.0, "Длинная строка");
text2.setFill(Color.BLACK);
text2.setCaretPosition(5);
text2.setCaretBias(true);

Path path2 = new Path(text2.getCaretShape());
path2.setStroke(Color.RED);
```

Получить траекторию, соответствующую заданной в первом параметре позиции текстового курсора, позволяет метод `caretShape()`. Формат метода:

```
public final PathElement[] caretShape(int charIndex, boolean caretBias)
```

Если параметр `caretBias` имеет значение `true`, то текстовый курсор располагается перед символом, а если `false` — то после символа. Пример:

```
Path path4 = new Path(text3.caretShape(0, true));  
path4.setStroke(Color.GREEN);
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH05_18_Text`, приведенном в каталоге `CH05_graphics` сопровождающего книгу электронного архива (см. приложение).

5.18.7. Класс *HitInfo*

Метод `hitTest()` из класса `Text` позволяет узнать индекс символа и позицию вставки по информации о координатах щелчка мышью. Метод доступен, начиная с JavaFX 9. Формат метода:

```
import javafx.scene.text.HitInfo;  
public final HitInfo hitTest(Point2D point)
```

Метод возвращает объект класса `HitInfo`, который содержит следующие методы:

❑ `getCharIndex()` — возвращает индекс символа. Формат метода:

```
public int getCharIndex()
```

❑ `getInsertionIndex()` — возвращает позицию вставки. Формат метода:

```
public int getInsertionIndex()
```

❑ `isLeading()` — возвращает значение `true`, если позиция щелчка ближе к началу символа, и `false` — если ближе к концу символа. Формат метода:

```
public boolean isLeading()
```

ЭЛЕКТРОННЫЙ АРХИВ

Пример использования метода `hitTest()` содержится в проекте `FX_CH05_18_Text`, приведенном в каталоге `CH05_graphics` сопровождающего книгу электронного архива (см. приложение).

5.18.8. Класс *TextFlow*

Контейнер, описываемый классом `TextFlow`, позволяет выравнивать текст, описываемый классом `Text`. Если текст не помещается внутри ширины контейнера, то он будет «перетекать» на следующую строку. При этом значения свойств, задающих координаты и выравнивание текста, из класса `Text` будут игнорироваться. Для принудительного переноса текста на новую строку предназначен символ `\n`. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.text.TextFlow;
```

Иерархия наследования:

Object — Node — Parent — Region — Pane — TextFlow

Создать экземпляр класса позволяют следующие конструкторы:

```
TextFlow()
TextFlow(Node... children)
```

Первый конструктор создает пустой контейнер, а второй конструктор позволяет сразу добавить узлы в контейнер. Добавить узлы после создания контейнера можно с помощью метода `getChildren()`. Пример создания заголовка с выравниванием по центру контейнера:

```
TextFlow paneTitle = new TextFlow();
paneTitle.setPrefWidth(400.0);
paneTitle.setPadding(new Insets(10.0));
paneTitle.setBackground(new Background(
    new BackgroundFill(Color.WHITE, null, null)));
paneTitle.setTextAlignment(TextAlignment.CENTER);
paneTitle.setLineSpacing(1.0);

Text text = new Text("Заголовок");
text.setFill(Color.RED);
text.setFont(
    Font.font("Verdana", FontWeight.BOLD, FontPosture.ITALIC, 24.0));
paneTitle.getChildren().add(text);
```

Управлять выравниванием текста внутри контейнера позволяют следующие свойства из класса `TextFlow`:

❑ `textAlignment` — выравнивание текста по горизонтали внутри контейнера. Значение по умолчанию: `TextAlignment.LEFT`. Методы доступа:

```
public final void setTextAlignment(TextAlignment value)
public final TextAlignment getTextAlignment()
public final ObjectProperty<TextAlignment>
    textAlignmentProperty()
```

В качестве значения параметра `value` можно указать следующие константы из перечисления `TextAlignment`:

```
import javafx.scene.text.TextAlignment;
```

- `LEFT` — по левому краю контейнера (значение по умолчанию);
- `RIGHT` — по правому краю контейнера;
- `CENTER` — по центру контейнера;
- `JUSTIFY` — по ширине контейнера;

- ❑ `lineSpacing` — расстояние между строками. Значение по умолчанию: 0.0. Методы доступа:

```
public final void setLineSpacing(double spacing)
public final double getLineSpacing()
public final DoubleProperty lineSpacingProperty()
```

Начиная с JavaFX 9, в классе `TextFlow` доступны следующие методы:

- ❑ `rangeShape()` — возвращает траекторию, соответствующую фрагменту от позиции `start` до `end` (не включая символ с этим индексом). Формат метода:

```
public final PathElement[] rangeShape(int start, int end)
```

- ❑ `caretShape()` — возвращает траекторию, соответствующую заданной в первом параметре позиции текстового курсора. Если параметр `leading` имеет значение `true`, то текстовый курсор располагается перед символом, а если `false` — то после символа. Формат метода:

```
public PathElement[] caretShape(int charIndex, boolean leading)
```

- ❑ `hitTest()` — позволяет узнать индекс символа и позицию вставки по информации о координатах щелчка мышью. Формат метода:

```
public final HitInfo hitTest(Point2D point)
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH05_18_TextFlow`, приведенном в каталоге `CH05_graphics` сопровождающего книгу электронного архива (см. *приложение*).

5.19. Класс *Path*: траектория

Класс `Path` описывает открытую или закрытую траекторию, состоящую из отдельных линий (прямых или кривых). Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.shape.Path;
```

Иерархия наследования:

```
Object — Node — Shape — Path
```

Конструкторы класса:

```
Path()
Path(PathElement... elements)
Path(Collection<? extends PathElement> elements)
```

Первый конструктор создает пустой объект. Пример открытой траектории с заливкой и обводкой:

```
Path p = new Path();
MoveTo moveTo = new MoveTo();
moveTo.setX(20.0);
```



```

moveTo.setY(20.0);
VLineTo line1 = new VLineTo();
line1.setY(200.0);
HLineTo line2 = new HLineTo();
line2.setX(200.0);
p.getElements().addAll(moveTo, line1, line2);
p.setFill(Color.RED);
p.setStroke(Color.BLACK);
p.setStrokeWidth(3.0);

```

Второй конструктор позволяет сразу указать элементы траектории через запятую или в виде массива. Пример закрытой траектории с заливкой и обводкой:

```

Path p2 = new Path(
    new MoveTo(250.0, 20.0),
    new VLineTo(200.0),
    new HLineTo(400.0),
    new ClosePath()
);
p2.setFill(Color.GREEN);
p2.setStroke(Color.BLACK);
p2.setStrokeWidth(3.0);

```

Третий конструктор в качестве параметра принимает коллекцию из элементов. Пример создания закрытой траектории с обводкой, но без заливки:

```

ArrayList<PathElement> list = new ArrayList<PathElement>();
list.add(new MoveTo(20.0, 300.0));
list.add(new LineTo(200.0, 350.0));
list.add(new LineTo(20.0, 450.0));
list.add(new ClosePath());
Path p3 = new Path(list);
p3.setStroke(Color.BLACK);
p3.setStrokeWidth(3.0);

```

**Получить ссылку на список с элементами траектории позволяет метод `getElements()`.
Формат метода:**

```

public final ObservableList<PathElement> getElements()

```

Для добавления элементов можно воспользоваться методами `add()` или `addAll()`:

```

Path p4 = new Path();
p4.getElements().add(new MoveTo(250.0, 300.0));
p4.getElements().addAll(new HLineTo(450.0), new VLineTo(450.0));

```

Класс `Path` содержит свойство `fillRule`, которое задает правила заливки внутренней части пути. В качестве значения можно указать константы: `NON_ZERO` (значение по умолчанию) и `EVEN_ODD` из перечисления `FillRule`. Методы доступа:

```

import javafx.scene.shape.FillRule;
public final void setFillRule(FillRule value)

```

```
public final FillRule getFillRule()
public final ObjectProperty<FillRule> fillRuleProperty()
```

Пример (обратите внимание: один объект может содержать несколько траекторий):

```
Path p6 = new Path(
    new MoveTo(100.0, 0.0),
    new LineTo(150.0, 100.0),
    new LineTo(50.0, 100.0),
    new ClosePath(),
    new MoveTo(140.0, 40.0),
    new LineTo(90.0, 140.0),
    new LineTo(190.0, 140.0),
    new ClosePath()
);
p6.setFill(Color.CORNFLOWERBLUE);
p6.setStroke(Color.BLACK);
p6.setStrokeWidth(3.0);
p6.setFillRule(FillRule.EVEN_ODD);
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH05_19_Path`, приведенном в каталоге `CH05_graphics` сопровождающего книгу электронного архива (см. приложение).

5.19.1. Класс *PathElement*

Абстрактный класс `PathElement` является базовым классом для всех классов, описывающих элементы траектории. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.shape.PathElement;
```

Иерархия наследования:

`Object` — `PathElement`

Класс `PathElement` содержит свойство `absolute`, которое определяет, являются ли координаты абсолютными (значение `true`, значение по умолчанию) или относительными (значение `false`). Методы доступа:

```
public final void setAbsolute(boolean value)
public final boolean isAbsolute()
public final BooleanProperty absoluteProperty()
```

Пример:

```
Path p5 = new Path();
p5.setStroke(Color.RED);
p5.getElements().add(new MoveTo(250.0, 350.0));
HLineTo hline = new HLineTo(400.0);
hline.setAbsolute(true);
```

```
VLineTo vline = new VLineTo(50.0);  
vline.setAbsolute(false);  
p5.getElements().addAll(hline, vline);
```

5.19.2. Классы *MoveTo* и *ClosePath*

Класс *MoveTo* позволяет переместить текущую позицию в точку с указанными координатами. Инструкция импорта:

```
// Модуль javafx.graphics  
import javafx.scene.shape.MoveTo;
```

Иерархия наследования:

Object — PathElement — MoveTo

Конструкторы класса:

```
MoveTo()  
MoveTo(double x, double y)
```

Первый конструктор создает объект с нулевыми координатами, а второй конструктор позволяет сразу указать координаты. Пример:

```
MoveTo moveTo = new MoveTo();  
moveTo.setX(20.0);  
moveTo.setY(20.0);
```

Управлять характеристиками объекта позволяют следующие свойства:

□ *x* — координата позиции по оси *x*. Методы доступа:

```
public final void setX(double value)  
public final double getX()  
public final DoubleProperty xProperty()
```

□ *y* — координата позиции по оси *y*. Методы доступа:

```
public final void setY(double value)  
public final double getY()  
public final DoubleProperty yProperty()
```

Класс *ClosePath* позволяет замкнуть текущую траекторию. Инструкция импорта:

```
// Модуль javafx.graphics  
import javafx.scene.shape.ClosePath;
```

Иерархия наследования:

Object — PathElement — ClosePath

Конструктор класса:

```
ClosePath()
```

Пример создания замкнутой траектории в виде треугольника:

```
Path p2 = new Path(  
    new MoveTo(250.0, 20.0),
```

```
new VLineTo(200.0),  
new HLineTo(400.0),  
new ClosePath()  
);
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH05_19_Path`, приведенном в каталоге `CH05_graphics` сопровождающего книгу электронного архива (см. *приложение*).

5.19.3. Класс *HLineTo*: горизонтальная линия

Класс `HLineTo` позволяет добавить горизонтальную линию. Инструкция импорта:

```
// Модуль javafx.graphics  
import javafx.scene.shape.HLineTo;
```

Иерархия наследования:

`Object` — `PathElement` — `HLineTo`

Конструкторы класса:

```
HLineTo()  
HLineTo(double x)
```

Пример:

```
HLineTo hline = new HLineTo();  
hline.setX(200.0);  
HLineTo hline2 = new HLineTo(400.0);
```

Класс `HLineTo` содержит свойство `x`, с помощью которого задается координата конечной точки по оси `x`. Методы доступа:

```
public final void setX(double value)  
public final double getX()  
public final DoubleProperty xProperty()
```

5.19.4. Класс *VLineTo*: вертикальная линия

Класс `VLineTo` позволяет добавить вертикальную линию. Инструкция импорта:

```
// Модуль javafx.graphics  
import javafx.scene.shape.VLineTo;
```

Иерархия наследования:

`Object` — `PathElement` — `VLineTo`

Конструкторы класса:

```
VLineTo()  
VLineTo(double y)
```

Пример:

```
VLineTo vline = new VLineTo();  
vline.setY(200.0);  
VLineTo vline2 = new VLineTo(200.0);
```

Класс `VLineTo` содержит свойство `y`, с помощью которого задается координата конечной точки по оси `y`. Методы доступа:

```
public final void setY(double value)  
public final double getY()  
public final DoubleProperty yProperty()
```

5.19.5. Класс *LineTo*: линия

Класс `LineTo` позволяет добавить прямую линию. Инструкция импорта:

```
// Модуль javafx.graphics  
import javafx.scene.shape.LineTo;
```

Иерархия наследования:

Object — PathElement — LineTo

Конструкторы класса:

```
LineTo()  
LineTo(double x, double y)
```

Пример:

```
LineTo line = new LineTo();  
line.setX(50.0);  
line.setY(20.0);  
LineTo line2 = new LineTo(250.0, 20.0);
```

Класс `LineTo` содержит следующие свойства:

❑ `x` — координата конца линии по оси `x`. Методы доступа:

```
public final void setX(double value)  
public final double getX()  
public final DoubleProperty xProperty()
```

❑ `y` — координата конца линии по оси `y`. Методы доступа:

```
public final void setY(double value)  
public final double getY()  
public final DoubleProperty yProperty()
```

Пример рисования треугольника:

```
Path p3 = new Path(  
    new MoveTo(20.0, 250.0),  
    new VLineTo(400.0),  
    new HLineTo(200.0),
```

```
        new LineTo(20.0, 250.0)
    );
    p3.setStroke(Color.BLACK);
    p3.setStrokeWidth(3.0);
```

5.19.6. Класс *ArcTo*: дуга

Класс *ArcTo* позволяет добавить дугу. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.shape.ArcTo;
```

Иерархия наследования:

Object — PathElement — ArcTo

Конструкторы класса:

```
ArcTo()
ArcTo(double radiusX, double radiusY, double xAxisRotation,
       double x, double y, boolean largeArcFlag, boolean sweepFlag)
```

Пример:

```
ArcTo arc = new ArcTo();
arc.setX(400.0);
arc.setY(100.0);
arc.setRadiusX(100.0);
arc.setRadiusY(100.0);
```

Класс *ArcTo* содержит следующие свойства:

- ❑ **radiusX** — радиус окружности по горизонтали. Методы доступа:

```
public final void setRadiusX(double value)
public final double getRadiusX()
public final DoubleProperty radiusXProperty()
```
- ❑ **radiusY** — радиус окружности по вертикали. Методы доступа:

```
public final void setRadiusY(double value)
public final double getRadiusY()
public final DoubleProperty radiusYProperty()
```
- ❑ **XAxisRotation** — угол поворота по оси x (в градусах). Методы доступа:

```
public final void setXAxisRotation(double value)
public final double getXAxisRotation()
public final DoubleProperty XAxisRotationProperty()
```
- ❑ **x** — координата конца дуги по оси x. Методы доступа:

```
public final void setX(double value)
public final double getX()
public final DoubleProperty xProperty()
```

□ `y` — координата конца дуги по оси `y`. Методы доступа:

```
public final void setY(double value)
public final double getY()
public final DoubleProperty yProperty()
```

□ `largeArcFlag` — если свойство имеет значение `true`, то будет использоваться большая по длине дуга. Значение по умолчанию: `false`. Методы доступа:

```
public final void setLargeArcFlag(boolean value)
public final boolean isLargeArcFlag()
public final BooleanProperty largeArcFlagProperty()
```

□ `sweepFlag` — если свойство имеет значение `true`, то дуга будет нарисована в положительном направлении угла (по часовой стрелке). Значение по умолчанию: `false`. Методы доступа:

```
public final void setSweepFlag(boolean value)
public final boolean isSweepFlag()
public final BooleanProperty sweepFlagProperty()
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH05_19_Path`, приведенном в каталоге `CH05_graphics` сопровождающего книгу электронного архива (см. приложение).

5.19.7. Класс *CubicCurveTo*: кубическая кривая Безье

Класс `CubicCurveTo` позволяет добавить кубическую кривую Безье. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.shape.CubicCurveTo;
```

Иерархия наследования:

```
Object — PathElement — CubicCurveTo
```

Конструкторы класса:

```
CubicCurveTo()
CubicCurveTo(double controlX1, double controlY1,
              double controlX2, double controlY2,
              double x, double y)
```

Пример:

```
CubicCurveTo cubic = new CubicCurveTo();
cubic.setControlX1(300.0);
cubic.setControlY1(250.0);
cubic.setControlX2(300.0);
cubic.setControlY2(50.0);
cubic.setX(400.0);
cubic.setY(150.0);
```

Класс `CubicCurveTo` содержит следующие свойства:

❑ `x` — координата конечной точки по оси `x`. Методы доступа:

```
public final void setX(double value)
public final double getX()
public final DoubleProperty xProperty()
```

❑ `y` — координата конечной точки по оси `y`. Методы доступа:

```
public final void setY(double value)
public final double getY()
public final DoubleProperty yProperty()
```

❑ `controlX1` — координата первой опорной точки по оси `x`. Методы доступа:

```
public final void setControlX1(double value)
public final double getControlX1()
public final DoubleProperty controlX1Property()
```

❑ `controlY1` — координата первой опорной точки по оси `y`. Методы доступа:

```
public final void setControlY1(double value)
public final double getControlY1()
public final DoubleProperty controlY1Property()
```

❑ `controlX2` — координата второй опорной точки по оси `x`. Методы доступа:

```
public final void setControlX2(double value)
public final double getControlX2()
public final DoubleProperty controlX2Property()
```

❑ `controlY2` — координата второй опорной точки по оси `y`. Методы доступа:

```
public final void setControlY2(double value)
public final double getControlY2()
public final DoubleProperty controlY2Property()
```

ЭЛЕКТРОННЫЙ АРХИВ

Пример использования класса `CubicCurveTo` содержится в проекте `FX_CH05_19_Path`, приведенном в каталоге `CH05_graphics` сопровождающего книгу электронного архива (см. приложение).

5.19.8. Класс *QuadCurveTo*: квадратичная кривая

Класс `QuadCurveTo` позволяет добавить квадратичную кривую. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.shape.QuadCurveTo;
```

Иерархия наследования:

`Object` — `PathElement` — `QuadCurveTo`

Конструкторы класса:

```
QuadCurveTo()  
QuadCurveTo(double controlX, double controlY,  
             double x, double y)
```

Пример:

```
QuadCurveTo quad = new QuadCurveTo();  
quad.setX(400.0);  
quad.setY(150.0);  
quad.setControlX(300.0);  
quad.setControlY(250.0);  
Path p = new Path(new MoveTo(100.0, 150.0), new HLineTo(200.0), quad);  
p.setStroke(Color.DARKBLUE);  
p.setStrokeWidth(3.0);
```

Класс `QuadCurveTo` содержит следующие свойства:

❑ `x` — координата конечной точки по оси `x`. Методы доступа:

```
public final void setX(double value)  
public final double getX()  
public final DoubleProperty xProperty()
```

❑ `y` — координата конечной точки по оси `y`. Методы доступа:

```
public final void setY(double value)  
public final double getY()  
public final DoubleProperty yProperty()
```

❑ `controlX` — координата опорной точки по оси `x`. Методы доступа:

```
public final void setControlX(double value)  
public final double getControlX()  
public final DoubleProperty controlXProperty()
```

❑ `controlY` — координата опорной точки по оси `y`. Методы доступа:

```
public final void setControlY(double value)  
public final double getControlY()  
public final DoubleProperty controlYProperty()
```

ЭЛЕКТРОННЫЙ АРХИВ

Пример использования класса `QuadCurveTo` содержится в проекте `FX_CH05_19_Path`, приведенном в каталоге `CH05_graphics` сопровождающего книгу электронного архива (см. приложение).

5.20. Класс ***SVGPath***

Класс `SVGPath` позволяет отобразить траекторию на основе строки в формате SVG. Полное описание формата строки можно найти в Интернете на странице <http://www.w3.org/TR/SVG/paths.html>. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.shape.SVGPath;
```

Иерархия наследования:

Object — Node — Shape — SVGPath

Конструктор класса:

```
SVGPath()
```

Класс SVGPath содержит следующие свойства:

- ❑ **content** — описание траектории (строка в формате SVG). Методы доступа:

```
public final void setContent(String value)
public final String getContent()
public final StringProperty contentProperty()
```

- ❑ **fillRule** — задает правила заливки внутренней части пути. В качестве значения можно указать константы: **NON_ZERO** (значение по умолчанию) и **EVEN_ODD** из перечисления **FillRule**. Методы доступа:

```
import javafx.scene.shape.FillRule;
public final void setFillRule(FillRule value)
public final FillRule getFillRule()
public final ObjectProperty<FillRule> fillRuleProperty()
```

В состав строки, описывающей траекторию, могут входить следующие основные элементы (полный список элементов и подробное описание параметров смотрите на странице <http://www.w3.org/TR/SVG/paths.html>):

- ❑ **М** или **m** — перемещает текущую позицию в точку с указанными координатами (аналог класса **MoveTo**). Формат:

```
M x y
```

- ❑ **Z** или **z** — позволяет замкнуть текущую траекторию (аналог класса **ClosePath**);

- ❑ **H** или **h** — добавляет горизонтальную линию (аналог класса **HLineTo**). Формат:

```
H x
```

- ❑ **V** или **v** — добавляет вертикальную линию (аналог класса **VLineTo**). Формат:

```
V y
```

Создадим открытую траекторию с заливкой и обводкой из горизонтальной и вертикальной линий:

```
SVGPath svg = new SVGPath();
svg.setContent("M 50 50 H 100 V 200");
svg.setFill(Color.RED);
svg.setStroke(Color.BLACK);
svg.setStrokeWidth(3.0);
Координаты x и y можно разделять запятой:
svg.setContent("M 50,50 H 100 V 200");
```

Между буквами и координатами можно вообще не указывать пробелы. Нарисуем замкнутую траекторию в форме треугольника:

```
SVGPath svg2 = new SVGPath();
svg2.setContent("M50 50H100V200Z");
svg2.setFill(Color.GREEN);
svg2.setStroke(Color.BLACK);
svg2.setStrokeWidth(3.0);
```

- **L** или **l** — добавляет прямую линию (аналог класса `LineTo`). Формат:

`L x y`

Пример рисования треугольника с заливкой, но без обводки:

```
SVGPath svg3 = new SVGPath();
svg3.setContent("M 100 100 L 50 200 L 150 200 Z");
svg3.setFill(Color.CHOCOLATE);
```

Если мы используем одну и ту же букву несколько раз подряд, то повторно букву можно и не указывать (в этом случае для наглядности координаты лучше разделить запятыми):

```
svg3.setContent("M 100 100 L 50,200 150,200 Z");
```

Если буква указана в верхнем регистре, то координаты задаются в абсолютных величинах, а если в нижнем регистре — то в относительных единицах. Кроме того, одна траектория может состоять из нескольких фрагментов. Для этого достаточно указать новое положение начальной точки с помощью буквы `м`. Пример рисования трех параллельных горизонтальных линий длиной 100 пх:

```
SVGPath svg4 = new SVGPath();
svg4.setContent("M 20 20 h 100 M 20 30 h 100 M 20 40 h 100");
svg4.setStroke(Color.BLACK);
svg4.setStrokeWidth(3.0);
```

- **A** или **a** — добавляет дугу (аналог класса `ArcTo`). Формат:

`A radiusX radiusY xAxisRotation largeArcFlag sweepFlag x y`

Описание этих параметров смотрите в *разд. 5.19.6*. В параметрах `largeArcFlag` и `sweepFlag` единица означает, что флаг установлен, а ноль — что сброшен. Пример рисования сектора (указаны относительные координаты):

```
SVGPath svg5 = new SVGPath();
svg5.setContent("M 200 200 h -100 a 100,100 0 1,0 100,-100 Z");
svg5.setFill(Color.NAVAJOWHITE);
```

- **C** или **c** — добавляет кубическую кривую Безье (аналог класса `CubicCurveTo`). Формат:

`C controlX1 controlY1 controlX2 controlY2 x y`

Описание этих параметров смотрите в *разд. 5.19.7*. Пример:

```
SVGPath svg9 = new SVGPath();
svg9.setContent("M 100 150 H 200 C 300,250 300,50 400,150");
```

```
svg9.setFill(Color.TRANSPARENT);  
svg9.setStroke(Color.BLACK);  
svg9.setStrokeWidth(3.0);
```

- **S** или **s** — добавляет кубическую кривую Безье. Используется только вторая опорная точка. Первая опорная точка добавляется автоматически таким образом, чтобы создать плавный переход между предыдущей кривой и текущей. Формат:

`S controlX2 controlY2 x y`

Пример:

```
SVGPath svg10 = new SVGPath();  
svg10.setContent("M 100 150 H 200 C 300,250 300,50 400,150 " +  
                "S 450,50 550,150");  
svg10.setFill(Color.TRANSPARENT);  
svg10.setStroke(Color.BLACK);  
svg10.setStrokeWidth(3.0);
```

- **Q** или **q** — добавляет квадратичную кривую (аналог класса `QuadCurveTo`). Формат:

`Q controlX controlY x y`

Описание этих параметров смотрите в *разд. 5.19.8*. Пример:

```
SVGPath svg11 = new SVGPath();  
svg11.setContent("M 100 150 H 200 Q 300,250 400,150");  
svg11.setFill(Color.TRANSPARENT);  
svg11.setStroke(Color.BLACK);  
svg11.setStrokeWidth(3.0);
```

- **T** или **t** — добавляет квадратичную кривую. Опорная точка добавляется автоматически таким образом, чтобы создать плавный переход между предыдущей кривой и текущей. Формат:

`T x y`

Пример:

```
SVGPath svg12 = new SVGPath();  
svg12.setContent("M 100 150 H 200 Q 300,250 400,150 T 600,150");  
svg12.setFill(Color.TRANSPARENT);  
svg12.setStroke(Color.BLACK);  
svg12.setStrokeWidth(3.0);
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH05_20_SVGPath`, приведенном в каталоге `CH05_graphics` сопровождающего книгу электронного архива (см. *приложение*).

ГЛАВА 6



Работа с изображениями

В этой главе мы рассмотрим работу с изображениями. Вы научитесь загружать изображение из файла, отображать его внутри окна или использовать в качестве фона узла, создавать новое изображение, преобразовывать форматы, а также создавать снимок (скриншот) сцены, только узла или всего экрана.

6.1. Класс *Image*: загрузка изображения из файла

Все изображения в JavaFX представлены классом `Image`. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.image.Image;
```

Конструкторы класса:

```
Image(InputStream is)
Image(InputStream is, double requestedWidth, double requestedHeight,
    boolean preserveRatio, boolean smooth)
Image(String url)
Image(String url, double requestedWidth, double requestedHeight,
    boolean preserveRatio, boolean smooth)
Image(String url, boolean backgroundLoading)
Image(String url, double requestedWidth, double requestedHeight,
    boolean preserveRatio, boolean smooth,
    boolean backgroundLoading)
```

Первые два конструктора позволяют загрузить изображение из открытого потока `InputStream`. Поддерживается загрузка изображений в форматах BMP, GIF, JPEG и PNG. Пример загрузки изображения (изображение в папке `src\img`):

```
Image image = null;
try {
    image = new Image(
        getClass().getResourceAsStream("/img/image1.jpg"));
}
```

```

} catch (Exception e) {
    System.out.println("Не удалось загрузить изображение1");
    return;
}

```

Если загрузить изображение не удалось, то будет сгенерировано исключение. Поэтому лучше выполнять загрузку внутри инструкции `try...catch`. Не лишней будет также проверка статуса загрузки изображения с помощью метода `isError()`:

```
System.out.println(image.isError() ? "Ошибка" : "Нет");
```

Остальные конструкторы в первом параметре принимают путь в виде строки. Это может быть абсолютный или относительный путь к локальному файлу или URL-адрес файла в Интернете. Пример (изображение в папке `src/img`):

```

Image image3 = null;
try {
    image3 = new Image(
        getClass().getResource("/img/image3.jpg").toExternalForm());
    if (image3.isError()) new RuntimeException();
} catch (Exception e) {
    System.out.println("Не удалось загрузить изображение3");
    return;
}

```

При загрузке можно сразу изменить размеры изображения. Для этого нужно указать требуемую ширину в параметре `requestedWidth`, а высоту — в параметре `requestedHeight`. Если в параметре `preserveRatio` указано значение `true`, то изменение размеров будет произведено с сохранением соотношения сторон, а если `false` — то при неправильно указанных размерах изображение будет искажено. Например, если мы хотим, чтобы высота изображения была равна 600 px и при этом соблюдалось соотношение сторон, то можно написать пример следующим образом:

```

image2 = new Image(getClass().getResourceAsStream("/img/image2.jpg"),
    0, 600, true, true);

```

В этом примере вместо неизвестной ширины было указано значение 0, а в параметре `preserveRatio` — значение `true`. Если в параметре `smooth` задано значение `true`, то при изменении размеров будет использоваться сглаживание.

Узнать размеры загруженного изображения позволяют следующие свойства:

- ❑ `width` — ширина загруженного изображения или значение 0.0, если загрузка не выполнена. Методы доступа:

```

public final double getWidth()
public final ReadOnlyDoubleProperty widthProperty()

```

- ❑ `height` — высота загруженного изображения или значение 0.0, если загрузка не выполнена. Методы доступа:

```

public final double getHeight()
public final ReadOnlyDoubleProperty heightProperty()

```

Пример:

```
System.out.println(image2.getWidth() + " " + image2.getHeight());
```

Загрузка изображений из Интернета может выполняться продолжительное время. Чтобы загрузка производилась в фоновом потоке, а не в текущем, нужно присвоить параметру `backgroundLoading` значение `true`. Узнать статус загрузки позволяют следующие свойства из класса `Image`:

- `progress` — процент загрузки изображения: от 0.0 (для 0 %) до 1.0 (для 100 %). Методы доступа:

```
public final double getProgress()
public final ReadOnlyDoubleProperty progressProperty
```

Пример назначения обработчика:

```
Image image5 = new Image("http://localhost/img/image.jpg", true);
image5.progressProperty().addListener(
    (obj, oldValue, newValue) -> {
        System.out.println("Изменение процента " + newValue);
        if (image5.getProgress() == 1.0) {
            System.out.print("Загрузка завершена ");
            if (!image5.isError()) {
                System.out.println("удачно");
            }
            else {
                System.out.println("неудачно");
            }
        }
    }
);
```

- `error` — содержит значение `true`, если при загрузке возникли ошибки, и `false` — в противном случае. Методы доступа:

```
public final boolean isError()
public final ReadOnlyBooleanProperty errorProperty()
```

- `exception` — если свойство `error` имеет значение `true`, то с помощью этого свойства можно получить информацию об ошибке. Методы доступа:

```
public final Exception getException()
public final ReadOnlyObjectProperty<Exception> exceptionProperty
```

Если загрузка в фоновом потоке выполняется долгое время, то ее можно прервать с помощью метода `cancel()`. Формат метода:

```
public void cancel()
```

Получить доступ к отдельным пикселям изображения позволяет метод `getPixelReader()`. Формат метода:

```
public final PixelReader getPixelReader()
```


Метод возвращает объект, реализующий интерфейс `PixelReader`, или значение `null`. Подробно интерфейс `PixelReader` мы рассмотрим в разд. 6.3, а сейчас просто получим цвет первого пиксела:

```
// import javafx.scene.image.PixelReader;
PixelReader pr = image.getPixelReader();
System.out.println(pr.getColor(0, 0));
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH06_01_Image`, приведенном в каталоге `CH06_images` сопровождающего книгу электронного архива (см. приложение).

6.2. Класс *ImageView*: отображение изображения в окне

Отобразить изображение в окне позволяет класс `ImageView`. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.image.ImageView;
```

Иерархия наследования:

Object — Node — `ImageView`

Конструкторы класса:

```
ImageView()
ImageView(Image image)
ImageView(String url)
```

Первый конструктор создает пустой объект. Пример:

```
Image image = null;
try {
    image = new Image(
        getClass().getResourceAsStream("/img/image1.jpg"));
    if (image.isError()) new RuntimeException();
} catch (Exception e) {
    System.out.println("Не удалось загрузить изображение");
    return;
}
ImageView iv = new ImageView();
iv.setImage(image);
```

Второй конструктор позволяет указать объект класса `Image`. Пример ограничения области отображения (изображение будет пропорционально уменьшено или увеличено, чтобы вписаться в эту область):

```
ImageView iv2 = new ImageView(image);
iv2.setFitWidth(200.0);
iv2.setFitHeight(200.0);
```

```
iv2.setPreserveRatio(true);  
iv2.setSmooth(true);  
iv2.setCache(true);
```

Обратите внимание: один и тот же объект класса `Image` можно указывать в нескольких компонентах `ImageView`.

Третий конструктор принимает путь в виде строки. Загрузим изображение и повернем компонент `ImageView` на 90 градусов:

```
ImageView iv3 = new ImageView(  
    getClass().getResource("/img/image1.jpg").toExternalForm());  
iv3.setRotate(90.0);
```

Класс `ImageView` содержит следующие свойства:

❑ **`image`** — содержит объект изображения или значение `null`. Методы доступа:

```
public final void setImage(Image value)  
public final Image getImage()  
public final ObjectProperty<Image> imageProperty()
```

Пример:

```
ImageView iv = new ImageView();  
iv.setImage(image);  
System.out.println(iv.getImage().getWidth());
```

❑ **`x`** — координата левого верхнего угла по оси `x`. Методы доступа:

```
public final void setX(double value)  
public final double getX()  
public final DoubleProperty xProperty()
```

❑ **`y`** — координата левого верхнего угла по оси `y`. Методы доступа:

```
public final void setY(double value)  
public final double getY()  
public final DoubleProperty yProperty()
```

❑ **`fitWidth`** — предпочтительная ширина отображения изображения. Если значение меньше или равно нулю, то используется исходная ширина изображения. Значение по умолчанию: 0.0. Методы доступа:

```
public final void setFitWidth(double value)  
public final double getFitWidth()  
public final DoubleProperty fitWidthProperty()
```

❑ **`fitHeight`** — предпочтительная высота отображения изображения. Если значение меньше или равно нулю, то используется исходная высота изображения. Значение по умолчанию: 0.0. Методы доступа:

```
public final void setFitHeight(double value)  
public final double getFitHeight()  
public final DoubleProperty fitHeightProperty()
```

- `preserveRatio` — если свойство имеет значение `true`, то изменение размеров изображения (чтобы вписаться в область размером `fitWidth` на `fitHeight`) будет производиться с сохранением пропорций. Значение по умолчанию: `false`. Методы доступа:

```
public final void setPreserveRatio(boolean value)
public final boolean isPreserveRatio()
public final BooleanProperty preserveRatioProperty()
```

- `smooth` — если свойство имеет значение `true`, то при изменении размеров изображения будет использоваться сглаживание. Значение по умолчанию: `false`. Методы доступа:

```
public final void setSmooth(boolean value)
public final boolean isSmooth()
public final BooleanProperty smoothProperty()
```

- `viewport` — позволяет указать, какую область изображения показывать. По умолчанию свойство имеет значение `null`, что приводит к отображению всего изображения. Методы доступа:

```
public final void setViewport(Rectangle2D value)
public final Rectangle2D getViewport()
public final ObjectProperty<Rectangle2D> viewportProperty()
```

Пример отображения области шириной 300 px и высотой 200 px:

```
ImageView iv2 = new ImageView();
iv2.setImage(im);
iv2.setX(20.0);
iv2.setY(500.0);
iv2.setViewport(new Rectangle2D(200.0, 100.0, 300.0, 200.0));
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH06_01_Image`, приведенном в каталоге `CH06_images` сопровождающего книгу электронного архива (см. приложение).

6.3. Интерфейс *PixelReader*: получение цвета пикселей

Получить информацию об отдельном пикселе изображения позволяют методы из интерфейса `PixelReader`. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.image.PixelReader;
```

Ссылку на объект, реализующий интерфейс `PixelReader`, возвращает метод `getPixelReader()` из класса `Image`. Обратите внимание, если изображение не было загружено, то метод вернет значение `null`. Формат метода:

```
public final PixelReader getPixelReader()
```

Пример:

```
// Image im = new Image(...)  
PixelReader pr = im.getPixelReader();
```

Интерфейс `PixelReader` содержит следующие методы:

- `getPixelFormat()` — возвращает объект `PixelFormat` с информацией о формате изображения. Формат метода:

```
public PixelFormat getPixelFormat()
```

Например, получить информацию о типе изображения позволяет метод `getType()` из класса `PixelFormat`. Формат метода:

```
public PixelFormat.Type getType()
```

Перечисление `PixelFormat.Type` содержит константы: `BYTE_BGRA`, `BYTE_BGRA_PRE`, `BYTE_INDEXED`, `BYTE_RGB`, `INT_ARGB` и `INT_ARGB_PRE`. Пример вывода типа изображения:

```
System.out.println(pr.getPixelFormat().getType());
```

- `getColor()` — возвращает цвет пиксела (в виде объекта класса `Color`), расположенного по указанным координатам. Формат метода:

```
public Color getColor(int x, int y)
```

Пример:

```
Color color = pr.getColor(10, 10);  
System.out.println(color.getRed() + " " + color.getGreen() +  
    " " + color.getBlue() + " " + color.getOpacity());  
System.out.println(color.toString()); // 0xa48c72ff
```

- `getArgb()` — возвращает цвет пиксела (в виде целого 32-битного числа), расположенного по указанным координатам. Формат метода:

```
public int getArgb(int x, int y)
```

Пример:

```
System.out.println(pr.getArgb(10, 10)); // -5993358  
int argb = pr.getArgb(10, 10);  
int alpha = ((argb >> 24) & 0xff);  
int red = ((argb >> 16) & 0xff);  
int green = ((argb >> 8) & 0xff);  
int blue = ((argb >> 0) & 0xff);  
System.out.println(red + " " + green + " " + blue + " " + alpha);
```

- `getPixels()` — записывает информацию о пикселах в буфер `buffer` в различных форматах. Форматы метода:

```
import javafx.scene.image.WritablePixelFormat;  
public void getPixels(int x, int y, int width, int height,  
    WritablePixelFormat<IntBuffer> pixelFormat,  
    int[] buffer, int offset, int scanlineStride)
```

```

public void getPixels(int x, int y, int width, int height,
    WritablePixelFormat<ByteBuffer> pixelformat,
    byte[] buffer, int offset, int scanlineStride)
public <T extends Buffer> void getPixels(int x, int y,
    int width, int height,
    WritablePixelFormat<T> pixelformat,
    T buffer, int scanlineStride)

```

Указать значение для параметра `pixelformat` позволяют следующие статические методы из класса `PixelFormat`:

```

public static WritablePixelFormat<IntBuffer>
    getIntArgbInstance()
public static WritablePixelFormat<IntBuffer>
    getIntArgbPreInstance()
public static WritablePixelFormat<ByteBuffer>
    getByteBgraInstance()
public static WritablePixelFormat<ByteBuffer>
    getByteBgraPreInstance()

```

Пример:

```

int[] buffer = new int[100];
pr.getPixels(0, 0, 10, 10,
    WritablePixelFormat.getIntArgbInstance(), buffer, 0, 10);
System.out.println(Arrays.toString(buffer));

byte[] buffer2 = new byte[400];
pr.getPixels(0, 0, 10, 10,
    WritablePixelFormat.getByteBgraInstance(), buffer2, 0, 10 * 4);
System.out.println(Arrays.toString(buffer2));

IntBuffer buffer3 = IntBuffer.allocate(100);
pr.getPixels(0, 0, 10, 10,
    WritablePixelFormat.getIntArgbInstance(), buffer3, 10);
System.out.println(Arrays.toString(buffer3.array()));

```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH06_01_Image`, приведенном в каталоге `CH06_images` сопровождающего книгу электронного архива (см. *приложение*).

6.4. Класс *WritableImage*: создание нового изображения

Итак, мы научились загружать изображение из файла, отображать его и получать цвет отдельного пиксела. А как изменить цвет пикселей и как создать изображение из программы? Ведь ни один из рассмотренных классов не имеет методов для из-

менения цвета пикселей. Для этого нужно воспользоваться классом `WritableImage`. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.image.WritableImage;
```

Иерархия наследования:

Object — Image — WritableImage

Как видно из иерархии, класс `WritableImage` является наследником класса `Image`, а значит, мы можем использовать объект класса `WritableImage` везде, где ожидается объект класса `Image`. Например, отобразить его в окне с помощью класса `ImageView`.

Создать новое изображение позволяют следующие конструкторы:

```
WritableImage(int width, int height)
WritableImage(PixelReader reader, int width, int height)
WritableImage(PixelReader reader,
               int x, int y, int width, int height)
```

Первый конструктор создает новое изображение шириной `width` и высотой `height` в формате `PixelFormat.BYTE_BGRA_PRE`. Цвет каждого пиксела будет равен значению `0x00000000`, т. е. изображение полностью прозрачное. Создадим новое изображение, зальем его цветом `Color.CHOCOLATE` и отобразим в окне:

```
// Image im = new Image(...)
WritableImage wim = new WritableImage((int)im.getWidth(),
                                       (int)im.getHeight());

PixelWriter pw = wim.getPixelWriter();
for (int x = 0, w = (int)wim.getWidth(); x < w; x++) {
    for (int y = 0, h = (int)wim.getHeight(); y < h; y++) {
        pw.setColor(x, y, Color.CHOCOLATE);
    }
}

ImageView iv = new ImageView(wim);
```

Второй и третий конструкторы позволяют создать новое изображение на основе другого изображения. Параметры `x` и `y` задают координаты начала копирования (во втором конструкторе координаты равны нулю), а параметры `width` и `height` — ширину и высоту. Пример создания копии изображения:

```
// Image im = new Image(...)
WritableImage wim2 = new WritableImage(im.getPixelReader(),
                                       (int)im.getWidth(), (int)im.getHeight());
ImageView iv2 = new ImageView(wim2);
```

Пример копирования лишь определенной области исходного изображения:

```
WritableImage wim3 = new WritableImage(im.getPixelReader(),
                                       200, 100, 300, 200);
ImageView iv3 = new ImageView(wim3);
```

Класс `WritableImage` наследует все методы из класса `Image` (см. *разд. 6.1*) и добавляет только метод `getPixelWriter()`, который возвращает объект, реализующий интерфейс `PixelWriter` (см. *разд. 6.5*). Формат метода:

```
public final PixelWriter getPixelWriter()
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH06_01_Image`, приведенном в каталоге `CH06_images` сопровождающего книгу электронного архива (см. *приложение*).

6.5. Интерфейс *PixelWriter*: изменение цвета пикселей

Изменить цвет отдельного пиксела изображения позволяют методы из интерфейса `PixelWriter`. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.image.PixelWriter;
```

Ссылку на объект, реализующий интерфейс `PixelWriter`, возвращает метод `getPixelWriter()` из класса `WritableImage`. Пример создания изображения и изменения цвета первого пиксела:

```
WritableImage wim = new WritableImage(300, 200);
PixelWriter pw = wim.getPixelWriter();
pw.setColor(0, 0, Color.CHOCOLATE);
```

Интерфейс `PixelWriter` содержит следующие методы:

- ❑ `getPixelFormat()` — возвращает объект `PixelFormat` с информацией о формате изображения. Формат метода:

```
public PixelFormat getPixelFormat()
```

Например, получить информацию о типе изображения позволяет метод `getType()` из класса `PixelFormat`. Формат метода:

```
public PixelFormat.Type getType()
```

Перечисление `PixelFormat.Type` содержит константы: `BYTE_BGRA`, `BYTE_BGRA_PRE`, `BYTE_INDEXED`, `BYTE_RGB`, `INT_ARGB` и `INT_ARGB_PRE`. Пример вывода типа изображения:

```
PixelWriter pw = wim.getPixelWriter();
System.out.println(pw.getPixelFormat().getType());
```

- ❑ `setColor()` — задает цвет пиксела на основе объекта класса `Color`. Формат метода:

```
public void setColor(int x, int y, Color c)
```

Пример:

```
pw.setColor(0, 0, Color.CHOCOLATE);
```

- `setArgb()` — задает цвет пиксела на основе целочисленного значения в формате `INT_ARGB`. **Формат метода:**

```
public void setArgb(int x, int y, int argb)
```

Пример:

```
pw.setArgb(10, 10, -6452363);
int argb = -6452363;
int alpha = ((argb >> 24) & 0xff);
int red = ((argb >> 16) & 0xff);
int green = ((argb >> 8) & 0xff);
int blue = ((argb >> 0) & 0xff);
System.out.println(red + " " + green + " " + blue + " " +
    alpha); // 157 139 117 255
```

- `setPixels()` — записывает данные из буфера или из изображения. **Форматы метода:**

```
public void setPixels(int x, int y, int width, int height,
    PixelFormat<IntBuffer> pixelformat,
    int[] buffer, int offset, int scanlineStride)
public void setPixels(int x, int y, int width, int height,
    PixelFormat<ByteBuffer> pixelformat,
    byte[] buffer, int offset, int scanlineStride)
public <T extends Buffer> void setPixels(
    int x, int y, int width, int height,
    PixelFormat<T> pixelformat,
    T buffer, int scanlineStride)
public void setPixels(int dstx, int dsty,
    int width, int height,
    PixelReader reader, int srcx, int srcy)
```

Пример записи в буфер и считывания из буфера:

```
// Image im = new Image(...)
PixelReader pr = im.getPixelReader();
WritableImage wim2 = new WritableImage(500, 400);

int[] buffer = new int[100];
pr.getPixels(0, 0, 10, 10,
    WritablePixelFormat.getIntArgbInstance(), buffer, 0, 10);
pw2.setPixels(0, 0, 10, 10,
    WritablePixelFormat.getIntArgbInstance(),
    buffer, 0, 10);

byte[] buffer2 = new byte[400];
pr.getPixels(0, 0, 10, 10,
    WritablePixelFormat.getBytesBgraInstance(),
    buffer2, 0, 10 * 4);
```



```

pw2.setPixels(50, 50, 10, 10,
    WritablePixelFormat.getByteBgraInstance(),
    buffer2, 0, 10 * 4);

IntBuffer buffer3 = IntBuffer.allocate(100);
pr.getPixels(0, 0, 10, 10,
    WritablePixelFormat.getIntArgbInstance(), buffer3, 10);
pw2.setPixels(100, 100, 10, 10,
    WritablePixelFormat.getIntArgbInstance(),
    buffer3, 10);

```

Пример использования четвертого формата (создание копии изображения):

```

WritableImage wim3 = new WritableImage((int)im.getWidth(),
    (int)im.getHeight());
PixelWriter pw3 = wim3.getPixelWriter();
pw3.setPixels(0, 0, (int)im.getWidth(), (int)im.getHeight(),
    im.getPixelReader(), 0, 0);

```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH06_01_Image`, приведенном в каталоге `CH06_images` сопровождающего книгу электронного архива (см. *приложение*).

6.6. Класс *SwingFXUtils*

Все рассмотренные в этой главе классы не содержат методов, позволяющих сохранить изображение в файл. Более того, класс `Image` не поддерживает сериализацию, поэтому мы не сможем сохранить объект этого класса в файл, и нам придется разрабатывать свой собственный формат. Однако разработчики добавили возможность преобразования объекта класса `Image` в объект класса `BufferedImage` из пакета `java.awt.image`, а также возможность выполнять обратную операцию. Для этого предназначены следующие статические методы из класса `SwingFXUtils`:

```

// Модуль javafx.swing
import javafx.embed.swing.SwingFXUtils;
// Модуль java.desktop
import java.awt.image.BufferedImage;

```

- `fromFXImage()` — преобразует объект класса `Image` в объект класса `BufferedImage`. Возвращает объект изображения или значение `null`. Формат метода:

```

public static BufferedImage fromFXImage(Image img,
    BufferedImage bimg)

```

Пример:

```

WritableImage wim = new WritableImage(200, 200);
// Что-то рисуем на изображении (фрагмент опущен)
BufferedImage bim = SwingFXUtils.fromFXImage(wim, null);

```

□ `toFXImage()` — преобразует объект класса `BufferedImage` в объект класса `WritableImage`. Формат метода:

```
public static WritableImage toFXImage(BufferedImage bim,
                                     WritableImage wimg)
```

Пример обратного преобразования:

```
WritableImage wim2 = SwingFXUtils.toFXImage(bim, null);
```

Какие же нам дает возможности преобразование объекта класса `Image` в объект класса `BufferedImage`? Во-первых, мы можем сохранить изображение в файл, например, используя статический метод `write()` из класса `ImageIO`. Один из форматов метода:

```
import java.io.File;
import javax.imageio.ImageIO;
import java.io.IOException;
public static boolean write(RenderedImage im, String formatName,
                           File output) throws IOException
```

Получим поддерживаемые классом `ImageIO` форматы записи:

```
System.out.println(Arrays.toString(ImageIO.getWriterFormatNames()));
// Java 8:
// [JPG, jpg, bmp, BMP, gif, GIF, WBMP, png, PNG, wbmp, jpeg, JPEG]
// Java 11:
// [JPG, jpg, tiff, bmp, BMP, gif, GIF, WBMP, png, PNG, JPEG, tif,
//  TIF, TIFF, wbmp, jpeg]
```

При сохранении следует учитывать формат данных внутри объекта `BufferedImage`. Не нужно забывать, что форматы файлов JPEG и BMP не поддерживают альфа-канал, поэтому при сохранении в этих форматах вначале следует преобразовать объект `BufferedImage` в формат без альфа-канала, иначе цвета на сохраненном изображении будут искажены, или при сохранении изображения получите ошибку.

Пример сохранения изображения в формате PNG:

```
try {
    boolean status = ImageIO.write(bim, "png", new File("img.png"));
    if (status) System.out.println("Сохранено");
    else System.out.println("Ошибка при сохранении");
} catch (IOException e) {
    System.out.println("Не удалось записать в файл");
}
```

Во-вторых, мы можем рисовать на таком изображении, а также выполнять различные преобразования. Например, выведем текст на изображение:

```
Graphics2D g = (Graphics2D) bim.getGraphics();
g.setFont(new java.awt.Font("Verdana", java.awt.Font.BOLD, 20));
g.setColor(java.awt.Color.BLACK);
```

```
g.setRenderingHint(RenderingHints.KEY_TEXT_ANTIALIASING,  
                    RenderingHints.VALUE_TEXT_ANTIALIAS_ON);  
g.drawString("Текст", 20, 30);
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH06_06_SwingFXUtils`, приведенном в каталоге `CH06_images` сопровождающего книгу электронного архива (см. *приложение*).

6.7. Класс *ImagePattern*

Объект класса `ImagePattern` можно использовать везде, где ожидается класс `Paint`. Например, в классах `Scene` и `Shape` есть метод `setFill()`, который позволяет установить изображение в качестве фона сцены или фигуры. Формат метода:

```
public final void setFill(Paint value)
```

ПРИМЕЧАНИЕ

Использовать изображение в качестве фона контейнеров и компонентов позволяет класс `BackgroundImage` (см. *разд. 3.5.4*).

Инструкция импорта класса `ImagePattern`:

```
// Модуль javafx.graphics  
import javafx.scene.paint.ImagePattern;
```

Иерархия наследования:

`Object` — `Paint` — `ImagePattern`

Конструкторы класса:

```
ImagePattern(Image image)  
ImagePattern(Image image, double x, double y,  
              double width, double height, boolean proportional)
```

Первый конструктор создает объект, в котором изображение будет соответствовать размеру области. Если размеры области станут изменяться, то изображение будет растягиваться или сжиматься без соблюдения исходных пропорций. Пример:

```
// Image img = new Image(...)  
Rectangle rect = new Rectangle(500.0, 100.0);  
ImagePattern imp = new ImagePattern(img);  
rect.setFill(imp);
```

Второй конструктор в параметрах `x` и `y` позволяет указать начальные координаты, а в параметрах `width` и `height` — размеры изображения. Если в параметре `proportional` указано значение `true`, то размеры задаются в относительных единицах (от 0.0 до 1.0), а если `false` — то в абсолютных единицах (в пикселах). Предыдущий пример полностью эквивалентен следующему коду (изображение растягивается по размеру области):

```
Rectangle rect3 = new Rectangle(500.0, 100.0);  
rect3.setFill(new ImagePattern(img, 0, 0, 1.0, 1.0, true));
```

Если мы хотим, чтобы при изменении размеров области изображение повторялось и заполняло за счет повторов всю область, то нужно указать абсолютные размеры изображения (если указать бесшовную текстуру, то фон всегда будет равномерно заполнен за счет повторов этой текстуры по горизонтали и вертикали):

```
Rectangle rect2 = new Rectangle(500.0, 100.0);  
rect2.setFill(new ImagePattern(img, 0, 0,  
    img.getWidth(), img.getHeight(), false));
```

Получить значения отдельных компонентов позволяют методы: `getImage()`, `getX()`, `getY()`, `getWidth()`, `getHeight()` и `isProportional()`. Форматы методов:

```
public final Image getImage()  
public final double getX()  
public final double getY()  
public final double getWidth()  
public final double getHeight()  
public final boolean isProportional()
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH06_07_ImagePattern`, приведенном в каталоге `CH06_images` сопровождающего книгу электронного архива (см. приложение).

6.8. Создание снимков сцены, узла и экрана

С помощью метода `snapshot()` из классов `Node` и `Scene` мы можем создать снимок отдельного узла или всей сцены. Форматы метода в классе `Node`:

```
public WritableImage snapshot(SnapshotParameters params,  
    WritableImage image)  
public void snapshot(Callback<SnapshotResult, Void> callback,  
    SnapshotParameters params, WritableImage image)
```

Пример создания снимка узла (обратите внимание: узел должен принадлежать сцене, чтобы были учтены все параметры):

```
SnapshotParameters sp = new SnapshotParameters();  
sp.setFill(Color.TRANSPARENT);  
WritableImage wim = rect.snapshot(sp, null);
```

Форматы метода в классе `Scene`:

```
public WritableImage snapshot(WritableImage image)  
public void snapshot(Callback<SnapshotResult, Void> callback,  
    WritableImage image)
```

Пример создания снимка сцены (видимой части сцены внутри окна):

```
WritableImage wim = scene.snapshot(null);
```

В библиотеке JavaFX до версии 11 не было методов, позволяющих сделать снимок всего окна или экрана. Если очень хочется сделать снимок экрана, то можно воспользоваться методом `createScreenCapture()` из класса `java.awt.Robot`. Формат метода:

```
// Модуль java.desktop
import java.awt.Robot;
import java.awt.Toolkit;
import java.awt.Rectangle;
public BufferedImage createScreenCapture(Rectangle screenRect)
```

Пример создания снимка всего экрана (без указателя мыши):

```
try {
    BufferedImage bim = (new Robot()).createScreenCapture(
        new java.awt.Rectangle(
            Toolkit.getDefaultToolkit().getScreenSize()
        )
    );
    WritableImage wim = SwingFXUtils.toFXImage(bim, null);
    newWindow(wim); // Показывает снимок в новом окне
} catch (Exception e) {
    e.printStackTrace();
}
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH06_08_snapshot`, приведенном в каталоге `CH06_images` сопровождающего книгу электронного архива (см. *приложение*).

Начиная с JavaFX 11, в состав библиотеки входит класс `Robot`, с помощью которого можно получить цвет любого пиксела на экране, а также сделать снимок указанной области основного экрана. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.robot.Robot;
```

Иерархия наследования:

`Object` — `Robot`

Конструктор класса:

`Robot()`

Методы класса `Robot`:

□ `getPixelColor()` — возвращает цвет пиксела. Форматы метода:

```
public Color getPixelColor(double x, double y)
public Color getPixelColor(Point2D location)
```

Пример:

```
Robot robot = new Robot();
System.out.println(robot.getPixelColor(10.0, 10.0));
System.out.println(robot.getPixelColor(new Point2D(10.0, 10.0)));
```

- ❑ `getScreenCapture()` — возвращает снимок указанной области основного экрана. Получить характеристики экрана позволяет класс `Screen` (см. *разд. 2.4.1*). Форматы метода:

```
public WritableImage getScreenCapture(WritableImage image,
                                     double x, double y, double width, double height)
public WritableImage getScreenCapture(WritableImage image,
                                     double x, double y, double width, double height,
                                     boolean scaleToFit)
public WritableImage getScreenCapture(WritableImage image,
                                     Rectangle2D region)
public WritableImage getScreenCapture(WritableImage image,
                                     Rectangle2D region, boolean scaleToFit)
```

В параметре `image` можно указать объект изображения или значение `null`, при котором будет создан новый объект изображения. Параметры `x` и `y` задают начальные координаты области, а параметры `width` и `height` — ширину и высоту области. Характеристики области можно также указать в виде объекта класса `Rectangle2D` в параметре `region`. Если в параметре `scaleToFit` указано значение `true`, то масштаб возвращаемого изображения будет таким, чтобы соответствовать характеристикам области, а если `false` — то реальные размеры возвращаемого изображения будут зависеть от разрешения экрана. Если параметр `scaleToFit` не указан, то он имеет значение `true`.

Пример создания снимка всего экрана (без указателя мыши):

```
Robot robot = new Robot();
Screen screen = Screen.getPrimary();
Rectangle2D bounds = screen.getBounds();
WritableImage wim = robot.getScreenCapture(null, bounds, true);
newWindow(wim); // Показывает снимок в новом окне
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH06_08_Robot`, приведенном в каталоге `CH06_images` сопровождающего книгу электронного архива (см. *приложение*).

ГЛАВА 7



Canvas API

Помимо создания графических объектов, которые мы можем добавлять на сцену, как и другие узлы, JavaFX предоставляет компонент `Canvas`, позволяющий программно рисовать на поверхности, которую мы будем называть *холстом*. Можно нарисовать любую геометрическую фигуру с заливкой или обводкой, вывести текст различными шрифтами, добавить на холст изображение, а также применить к графическим объектам эффекты или трансформации. Холст позволяет обрабатывать все события мыши и клавиатуры, поэтому его можно использовать для создания своих собственных компонентов произвольной формы. Следует учитывать, что отрисовка выполняется видеокартой, а не процессором, поэтому размеры холста ограничены размерами текстуры (8192 px).

7.1. Класс *Canvas*

Класс `Canvas` описывает холст, на котором мы можем рисовать. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.canvas.Canvas;
```

Иерархия наследования:

`Object` — `Node` — `Canvas`

Как видно из иерархии, класс `Canvas` наследует все методы из класса `Node`, т. е. является узлом. Конструкторы класса:

```
Canvas()
Canvas(double width, double height)
```

Первый конструктор создает холст с настройками по умолчанию. Создадим холст, укажем его размеры и зальем зеленым цветом:

```
Canvas canvas = new Canvas();
canvas.setWidth(400.0);
canvas.setHeight(150.0);
```



```
GraphicsContext gc = canvas.getGraphicsContext2D();
gc.setFill(Color.GREEN);
gc.fillRect(0.0, 0.0, canvas.getWidth(), canvas.getHeight());
```

Второй конструктор позволяет сразу задать размеры холста. Создадим холст белого цвета с рамкой черного цвета:

```
Canvas canvas2 = new Canvas(400.0, 150.0);
GraphicsContext gc2 = canvas2.getGraphicsContext2D();
gc2.setFill(Color.WHITE);
gc2.setStroke(Color.BLACK);
gc2.fillRect(0.0, 0.0, canvas2.getWidth(), canvas2.getHeight());
gc2.strokeRect(1.0, 1.0, canvas.getWidth() - 2.0,
              canvas.getHeight() - 2.0);
```

Класс `Canvas` содержит следующие свойства:

□ `width` — ширина холста. Методы доступа:

```
public final void setWidth(double value)
public final double getWidth()
public final DoubleProperty widthProperty()
```

□ `height` — высота холста. Методы доступа:

```
public final void setHeight(double value)
public final double getHeight()
public final DoubleProperty heightProperty()
```

ПРИМЕЧАНИЕ

Как уже отмечалось, максимальный размер холста ограничен размером текстуры (8192 px).

7.2. Класс *GraphicsContext*

Класс `GraphicsContext` описывает средства, позволяющие рисовать на холсте. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.canvas.GraphicsContext;
```

Получить ссылку на объект `GraphicsContext`, связанный с холстом, позволяет метод `getGraphicsContext2D()` из класса `Canvas`. Формат метода:

```
public GraphicsContext getGraphicsContext2D()
```

Пример:

```
Canvas canvas = new Canvas(400.0, 300.0);
GraphicsContext gc = canvas.getGraphicsContext2D();
```

Получить ссылку на объект холста позволяет метод `getCanvas()`. Формат метода:

```
public Canvas getCanvas()
```

7.2.1. Изменение характеристик заливки

Управлять характеристиками фона позволяют следующие методы из класса `GraphicsContext`:

- ❑ `setFill()` — задает характеристики фона. Формат метода:

```
public void setFill(Paint value)
```

- ❑ `getFill()` — возвращает характеристики фона. Формат метода:

```
public Paint getFill()
```

Если характеристики фона не заданы, то используется черный цвет. В качестве значения параметра `value` можно указать:

- ❑ объект класса `Color` — задает сплошной цвет фона. Пример:

```
gc.setFill(Color.WHITE);
gc.fillRect(0.0, 0.0, 400.0, 300.0);
gc.setFill(Color.BLACK);
gc.fillRect(10.0, 10.0, 40.0, 40.0);
```

- ❑ объект класса `LinearGradient` — заливка линейным градиентом. Пример:

```
Stop[] stops = new Stop[] {
    new Stop(0, Color.BLACK), new Stop(1, Color.WHITE)
};
gc.setFill(new LinearGradient(0, 0, 1, 0, true,
                             CycleMethod.NO_CYCLE, stops));
gc.fillRect(60.0, 10.0, 40.0, 40.0);
```

- ❑ объект класса `RadialGradient` — радиальный градиент. Пример:

```
gc.setFill(new RadialGradient(240.0, 0.7, 0.5, 0.5, 0.5,
                              true, CycleMethod.REFLECT, stops));
gc.fillRect(110.0, 10.0, 40.0, 40.0);
```

- ❑ объект класса `ImagePattern` — устанавливает фоновое изображение. Пример:

```
WritableImage im = new WritableImage(10, 10);
PixelWriter pw = im.getPixelWriter();
for (int x = 0, w = (int) im.getWidth(); x < w; x++) {
    for (int y = 0, h = (int) im.getHeight(); y < h; y++) {
        if (x > 3 && x < 8)
            pw.setColor(x, y, Color.WHITE);
        else pw.setColor(x, y, Color.CHOCOLATE);
    }
}
gc.setFill(new ImagePattern(im, 0.0, 0.0,
                           im.getWidth(), im.getHeight(), false));
gc.fillRect(160.0, 10.0, 40.0, 40.0);
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH07_01_Canvas`, приведенном в каталоге `CH07_canvas` сопровождающего книгу электронного архива (см. *приложение*).

7.2.2. Изменение характеристик обводки

Управлять характеристиками обводки позволяют следующие методы из класса `GraphicsContext`:

- `setStroke()` — задает цвет обводки. По умолчанию используется черный цвет. **Формат метода:**

```
public void setStroke(Paint value)
```

В качестве значения параметра `value` можно указать: сплошной цвет (класс `Color`), линейный градиент (класс `LinearGradient`), радиальный градиент (класс `RadialGradient`) или изображение (класс `ImagePattern`). Пример указания красного цвета:

```
gc.setStroke(Color.RED);  
gc.strokeRect(10.0, 60.0, 40.0, 40.0);
```

- `getStroke()` — возвращает цвет обводки. **Формат метода:**

```
public Paint getStroke()
```

- `setLineWidth()` — задает ширину (толщину) обводки. Значение по умолчанию: 1.0. **Формат метода:**

```
public void setLineWidth(double lw)
```

Пример:

```
gc.setLineWidth(5.0);
```

- `getLineWidth()` — возвращает ширину (толщину) обводки. **Формат метода:**

```
public double getLineWidth()
```

- `setLineDashOffset()` — задает смещение начала пунктирной обводки. Значение по умолчанию: 0.0. **Формат метода:**

```
public void setLineDashOffset(double dashOffset)
```

- `getLineDashOffset()` — возвращает смещение начала пунктирной обводки. **Формат метода:**

```
public double getLineDashOffset()
```

- `setLineDashes()` — задает значения для пунктирной линии. Значения указываются через запятую или в виде массива. Четные индексы задают длину штриха, а нечетные — длину пропуска. **Формат метода:**

```
public void setLineDashes(double... dashes)
```

Пример рисования пунктирных линий:

```
gc.setStroke(Color.BLUE);
gc.setLineWidth(8.0);
gc.setLineDashes(15.0, 40.0);
gc.strokeLine(20.0, 120.0, 300.0, 120.0);
gc.setLineDashOffset(40.0);
gc.strokeLine(20.0, 140.0, 300.0, 140.0);
```

Чтобы сделать следующую линию опять сплошной, достаточно передать в метод `setLineDashes()` пустое значение, пустой массив или значение `null`. Пример:

```
gc.setLineDashOffset(0.0);
gc.setLineDashes();
gc.strokeLine(20.0, 160.0, 300.0, 160.0);
```

- `getLineDashes()` — возвращает массив со значениями для пунктирной линии или значение `null`. Формат метода:

```
public double[] getLineDashes()
```

- `setLineCap()` — задает форму окончания линии. Значение по умолчанию: `StrokeLineCap.SQUARE`. Формат метода:

```
public void setLineCap(StrokeLineCap value)
```

В качестве значения параметра `value` можно указать следующие константы из перечисления `StrokeLineCap`:

```
import javafx.scene.shape.StrokeLineCap;
```

- `SQUARE` — квадратные концы (прибавляются к длине линии), значение по умолчанию;
- `BUTT` — концы никак не оформляются;
- `ROUND` — закругленные концы (прибавляются к длине линии).

Пример:

```
gc.setStroke(Color.BLACK);
gc.setLineWidth(15.0);
gc.setLineCap(StrokeLineCap.ROUND);
gc.strokeLine(20.0, 180.0, 300.0, 180.0);
gc.setLineCap(StrokeLineCap.SQUARE);
gc.strokeLine(20.0, 200.0, 300.0, 200.0);
gc.setLineCap(StrokeLineCap.BUTT);
gc.strokeLine(20.0, 220.0, 300.0, 220.0);
```

- `getLineCap()` — возвращает форму окончания линии. Формат метода:

```
public StrokeLineCap getLineCap()
```

- `setLineJoin()` — задает форму окончания в месте соединения двух линий обводки. Значение по умолчанию: `StrokeLineJoin.MITER`. Формат метода:

```
public void setLineJoin(StrokeLineJoin value)
```

В качестве значения параметра `value` можно указать следующие константы из перечисления `StrokeLineJoin`:

```
import javafx.scene.shape.StrokeLineJoin;
```

- `MITER` — обычные углы (значение по умолчанию);
- `BEVEL` — скошенные углы;
- `ROUND` — закругленные углы.

Пример:

```
gc.setStroke(Color.RED);
gc.setLineWidth(15.0);
gc.setLineJoin(StrokeLineJoin.MITER);
gc.strokePolyline(new double[] {100.0, 30.0, 100.0},
                  new double[] {250.0, 290.0, 290.0}, 3);
gc.setLineJoin(StrokeLineJoin.ROUND);
gc.strokePolyline(new double[] {200.0, 120.0, 200.0},
                  new double[] {250.0, 290.0, 290.0}, 3);
gc.setLineJoin(StrokeLineJoin.BEVEL);
gc.strokePolyline(new double[] {300.0, 220.0, 300.0},
                  new double[] {250.0, 290.0, 290.0}, 3);
```

- `getLineJoin()` — возвращает форму окончания в месте соединения двух линий обводки. **Формат метода:**

```
public StrokeLineJoin getLineJoin()
```

- `setMiterLimit()` — задает ограничение длины угла при использовании стиля `StrokeLineJoin.MITER`. Значение по умолчанию: 10.0. Если значение превышено, то угол будет скошенным. **Формат метода:**

```
public void setMiterLimit(double ml)
```

- `getMiterLimit()` — возвращает ограничение длины угла. **Формат метода:**

```
public double getMiterLimit()
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH07_01_Canvas`, приведенном в каталоге `CH07_canvas` сопровождающего книгу электронного архива (см. приложение).

7.2.3. Изменение цвета отдельных пикселей

С помощью метода `getPixelWriter()` из класса `GraphicsContext` мы можем получить ссылку на объект, реализующий интерфейс `PixelWriter`, и изменять цвет любых пикселей на холсте. **Формат метода:**

```
public PixelWriter getPixelWriter()
```

Напомню, что методы интерфейса `PixelWriter` мы уже рассматривали в предыдущей главе (см. *разд. 6.5*). Пример рисования красного прямоугольника на холсте:

- `strokeArc()` — рисует дугу или сектор, используя характеристики обводки (см. *разд. 7.2.2*). Формат метода:

```
public void strokeArc(double x, double y, double w, double h,
    double startAngle, double arcExtent, ArcType closure)
```

Параметры `x` и `y` задают координаты точки вставки, параметры `w` и `h` — ширину и высоту дуги, параметр `startAngle` — начальный угол в градусах, параметр `arcExtent` — длину дуги в градусах, начиная с угла `startAngle`, а параметр `closure` — тип дуги. В параметре `closure` можно указать следующие константы из перечисления `ArcType`:

```
import javafx.scene.shape.ArcType;
```

- `OPEN` — концы дуги не соединяются;
- `CHORD` — концы дуги соединяются прямой линией;
- `ROUND` — концы дуги соединяются через центр окружности.

Пример:

```
gc.setStroke(Color.DARKGREEN);
gc.setLineWidth(3.0);
gc.strokeArc(10.0, 150.0, 100.0, 100.0, 45.0, 270.0,
    ArcType.OPEN);
```

- `fillArc()` — рисует сектор, используя характеристики заливки (см. *разд. 7.2.1*). Формат метода:

```
public void fillArc(double x, double y, double w, double h,
    double startAngle, double arcExtent, ArcType closure)
```

Смысл параметров точно такой же, как и у метода `strokeArc()`. Пример:

```
gc.setFill(Color.KHAKI);
gc.fillArc(300.0, 300.0, 100.0, 100.0, 45.0, 270.0,
    ArcType.ROUND);
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH07_01_Canvas`, приведенном в каталоге `CH07_canvas` сопровождающего книгу электронного архива (см. *приложение*).

7.2.5. Рисование траектории

Управлять рисованием траектории позволяют следующие методы из класса `GraphicsContext`:

- `beginPath()` — начинает новую траекторию. Формат метода:

```
public void beginPath()
```

- `moveTo()` — позволяет переместить текущую позицию в точку с указанными координатами. Формат метода:

```
public void moveTo(double x0, double y0)
```

- ❑ `closePath()` — позволяет замкнуть текущую траекторию. Формат метода:

```
public void closePath()
```

- ❑ `stroke()` — прорисовывает текущую траекторию, используя характеристики обводки (см. *разд. 7.2.2*). Формат метода:

```
public void stroke()
```

- ❑ `fill()` — прорисовывает текущую траекторию, используя характеристики заливки (см. *разд. 7.2.1*). Формат метода:

```
public void fill()
```

Пример рисования замкнутого треугольника с обводкой и заливкой:

```
gc.setFill(Color.DARKGREEN);  
gc.setStroke(Color.RED);  
gc.setLineWidth(3.0);  
gc.beginPath();  
gc.moveTo(20.0, 20.0);  
gc.lineTo(100.0, 20.0);  
gc.lineTo(100.0, 80.0);  
gc.closePath();  
gc.stroke();  
gc.fill();
```

- ❑ `setFillRule()` — задает правила заливки внутренней части траектории. В качестве значения можно указать константы: `NON_ZERO` (значение по умолчанию) и `EVEN_ODD` из перечисления `FillRule`. Формат метода:

```
import javafx.scene.shape.FillRule;  
public void setFillRule(FillRule fillRule)
```

- ❑ `getFillRule()` — возвращает правила заливки внутренней части траектории. Формат метода:

```
public FillRule getFillRule()
```

- ❑ `lineTo()` — добавляет прямую линию к текущей траектории. Формат метода:

```
public void lineTo(double x1, double y1)
```

Пример:

```
gc.lineTo(100.0, 20.0);
```

- ❑ `arcTo()` — добавляет дугу к текущей траектории. Формат метода:

```
public void arcTo(double x1, double y1,  
                 double x2, double y2, double radius)
```

Представьте, что у нас есть три точки с координатами (x_0, y_0) , (x_1, y_1) и (x_2, y_2) и круг радиусом `radius`. Координаты первой точки совпадают с координатами текущей позиции, например, заданными с помощью метода `moveTo()`. Проведите прямую линию между первой и второй точками, а также между второй и

третьей точками. Теперь подставьте круг таким образом, чтобы он касался этих двух линий. Дуга пройдет от первой точки касания до второй точки касания. Причем первая точка будет соединена прямой линией с первой точкой касания. Пример рисования дуги:

```
gc.setStroke(Color.BLACK);
gc.setLineWidth(5.0);
gc.beginPath();
gc.moveTo(100.0, 100.0);
gc.arcTo(200.0, 0.0, 300.0, 100.0, 100.0);
gc.stroke();
```

- **arc()** — добавляет дугу к текущей траектории. Формат метода:

```
public void arc(double centerX, double centerY,
               double radiusX, double radiusY,
               double startAngle, double length)
```

Параметры `centerX` и `centerY` задают координаты центра эллипса, параметры `radiusX` и `radiusY` — радиус эллипса по горизонтали и вертикали, параметр `startAngle` — начальный угол в градусах, а параметр `length` — длину дуги в градусах, начиная с угла `startAngle`. Пример:

```
gc.setStroke(Color.BLACK);
gc.setLineWidth(5.0);
gc.beginPath();
gc.moveTo(10.0, 200.0);
gc.arc(50.0, 200.0, 40.0, 40.0, 180.0, 180.0);
gc.arc(130.0, 200.0, 40.0, 40.0, 180.0, -180.0);
gc.stroke();
```

- **bezierCurveTo()** — добавляет кубическую кривую Безье к текущей траектории. Формат метода:

```
public void bezierCurveTo(double xc1, double yc1,
                        double xc2, double yc2, double x1, double y1)
```

Параметры `xc1` и `yc1` задают координаты первой опорной точки, параметры `xc2` и `yc2` — координаты второй опорной точки, а параметры `x1` и `y1` — координаты конечной точки. Пример:

```
gc.setStroke(Color.BLACK);
gc.setLineWidth(5.0);
gc.beginPath();
gc.moveTo(50.0, 300.0);
gc.bezierCurveTo(200.0, 400.0, 200.0, 200.0, 350.0, 300.0);
gc.stroke();
```

- **quadraticCurveTo()** — добавляет квадратичную кривую к текущей траектории. Формат метода:

```
public void quadraticCurveTo(double xc, double yc,
                           double x1, double y1)
```

Параметры `xc` и `yc` задают координаты опорной точки, а параметры `x1` и `y1` — координаты конечной точки. Пример:

```
gc.setStroke(Color.BLACK);
gc.setLineWidth(5.0);
gc.beginPath();
gc.moveTo(50.0, 400.0);
gc.quadraticCurveTo(200.0, 300.0, 350.0, 400.0);
gc.stroke();
```

- `appendSVGPath()` — добавляет траекторию на основе строки в формате SVG к текущей траектории. Описание основных параметров формата SVG можно найти в *разд. 5.20*. Формат метода:

```
public void appendSVGPath(String svgpath)
```

Пример рисования треугольника:

```
gc.setStroke(Color.BLACK);
gc.setLineWidth(5.0);
gc.beginPath();
gc.moveTo(200.0, 370.0);
gc.appendSVGPath("L 150 420 L 250 420");
gc.closePath();
gc.stroke();
```

- `rect()` — добавляет к траектории прямоугольник. Формат метода:

```
public void rect(double x, double y, double w, double h)
```

Этот метод удобно использовать совместно с методом `clip()` для добавления маски:

```
gc.beginPath();
gc.rect(100.0, 100.0, 200.0, 200.0);
gc.clip();
```

- `clip()` — создает маску на основе текущего пути. Область внутри пути будет видна, а вне — скрыта. Формат метода:

```
public void clip()
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH07_01_Canvas`, приведенном в каталоге `CH07_canvas` сопровождающего книгу электронного архива (см. *приложение*).

7.2.6. Рисование фигур

Нарисовать на холсте различные фигуры позволяют следующие методы из класса `GraphicsContext`:

- `fillRect()` — рисует прямоугольник, используя характеристики заливки (см. *разд. 7.2.1*). Формат метода:

```
public void fillRect(double x, double y, double w, double h)
```

Параметры x и y задают координаты левого верхнего угла прямоугольника, а параметры w и h — ширину и высоту. Пример:

```
gc.setFill(Color.BLUE);  
gc.fillRect(20.0, 20.0, 40.0, 40.0);
```

- ❑ `fillRoundRect()` — рисует прямоугольник со скругленными углами, используя характеристики заливки (см. *разд. 7.2.1*). Формат метода:

```
public void fillRoundRect(double x, double y,  
    double w, double h, double arcWidth, double arcHeight)
```

Пример:

```
gc.setFill(Color.GREEN);  
gc.fillRoundRect(80.0, 20.0, 40.0, 40.0, 20.0, 20.0);
```

- ❑ `strokeRect()` — рисует прямоугольник, используя характеристики обводки (см. *разд. 7.2.2*). Формат метода:

```
public void strokeRect(double x, double y, double w, double h)
```

Пример:

```
gc.setStroke(Color.RED);  
gc.strokeRect(140.0, 20.0, 40.0, 40.0);
```

- ❑ `strokeRoundRect()` — рисует прямоугольник со скругленными углами, используя характеристики обводки (см. *разд. 7.2.2*). Формат метода:

```
public void strokeRoundRect(double x, double y,  
    double w, double h, double arcWidth, double arcHeight)
```

Пример:

```
gc.setStroke(Color.PURPLE);  
gc.strokeRoundRect(200.0, 20.0, 40.0, 40.0, 20.0, 20.0);
```

- ❑ `fillOval()` — рисует окружность, используя характеристики заливки (см. *разд. 7.2.1*). Формат метода:

```
public void fillOval(double x, double y, double w, double h)
```

Параметры x и y задают координаты левого верхнего угла прямоугольника, внутри которого вписана окружность, а параметры w и h — ширину и высоту.

Пример:

```
gc.setFill(Color.BLUE);  
gc.fillOval(20.0, 100.0, 100.0, 40.0);
```

- ❑ `strokeOval()` — рисует окружность, используя характеристики обводки (см. *разд. 7.2.2*). Формат метода:

```
public void strokeOval(double x, double y, double w, double h)
```

Пример:

```
gc.setStroke(Color.RED);  
gc.setLineWidth(3.0);  
gc.strokeOval(200.0, 100.0, 40.0, 100.0);
```

- ❑ `fillPolygon()` — рисует многоугольник, используя характеристики заливки (см. *разд. 7.2.1*). Формат метода:

```
public void fillPolygon(double[] xPoints, double[] yPoints,  
                        int nPoints)
```

В массиве `xPoints` задаются координаты точек по оси X, а в массиве `yPoints` — по оси Y. Параметр `nPoints` задает количество точек. Пример рисования треугольника:

```
gc.setFill(Color.BLACK);  
gc.fillPolygon(new double[] {100.0, 20.0, 100.0},  
               new double[] {180.0, 220.0, 220.0}, 3);
```

- ❑ `strokePolygon()` — рисует многоугольник, используя характеристики обводки (см. *разд. 7.2.2*). Формат метода:

```
public void strokePolygon(double[] xPoints, double[] yPoints,  
                          int nPoints)
```

Пример:

```
gc.setStroke(Color.RED);  
gc.setLineWidth(3.0);  
gc.strokePolygon(new double[] {100.0, 20.0, 100.0},  
                 new double[] {250.0, 290.0, 290.0}, 3);
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH07_01_Canvas`, приведенном в каталоге `CH07_canvas` сопровождающего книгу электронного архива (см. *приложение*).

7.2.7. Вывод текста

Управлять характеристиками текста позволяют следующие методы из класса `GraphicsContext`:

- ❑ `setFont()` — задает характеристики шрифта с помощью класса `Font` (см. *разд. 5.18.3*). Формат метода:

```
public void setFont(Font f)
```

Пример:

```
gc.setFill(Color.BLACK);  
gc.setFont(new Font(24.0));  
gc.fillText("Font(24.0)", 20.0, 40.0);
```

- ❑ `getFont()` — возвращает характеристики шрифта. Формат метода:

```
public Font getFont()
```

Выведем настройки шрифта по умолчанию:

```
System.out.println(gc.getFont());  
// Font{name=System Regular, family=System, style=Regular,  
// size=12.0}
```

- `setFontSmoothingType()` — задает тип сглаживания шрифта. Можно указать константы: `GRAY` (значение по умолчанию) или `LCD` из перечисления `FontSmoothingType`. Формат метода:

```
import javafx.scene.text.FontSmoothingType;
public void setFontSmoothingType(FontSmoothingType value)
```

Пример:

```
gc.setFontSmoothingType(FontSmoothingType.GRAY);
```

- `getFontSmoothingType()` — возвращает тип сглаживания шрифта. Формат метода:

```
public FontSmoothingType getFontSmoothingType()
```

- `setTextAlign()` — задает горизонтальное выравнивание текста относительно начальной точки. Значение по умолчанию: `TextAlignment.LEFT`. Формат метода:

```
import javafx.scene.text.TextAlignment;
public void setTextAlign(TextAlignment align)
```

В качестве значения параметра `align` можно указать константы `LEFT`, `RIGHT` или `CENTER` из перечисления `TextAlignment`. Пример:

```
gc.setFill(Color.BLACK);
gc.setFont(Font.font("Verdana", FontWeight.NORMAL,
    FontPosture.REGULAR, 12.0));
gc.setTextAlign(TextAlignment.RIGHT);
gc.fillText("TextAlignment.RIGHT", 150.0, 80.0);
```

- `getTextAlign()` — возвращает горизонтальное выравнивание текста относительно начальной точки. Формат метода:

```
public TextAlignment getTextAlign()
```

- `setTextBaseline()` — задает вертикальное выравнивание текста относительно начальной точки. Значение по умолчанию: `VPos.BASELINE`. Формат метода:

```
import javafx.geometry.VPos;
public void setTextBaseline(VPos value)
```

В качестве значения параметра `value` можно указать константы `TOP`, `CENTER`, `BASLINE` или `BOTTOM` из перечисления `VPos`. Пример выравнивания по верху:

```
gc.setTextBaseline(VPos.TOP);
gc.fillText("VPos.TOP", 50.0, 140.0);
```

- `getTextBaseline()` — возвращает вертикальное выравнивание текста относительно начальной точки. Формат метода:

```
public VPos getTextBaseline()
```

Вывести текст позволяют следующие методы из класса `GraphicsContext`:

- `fillText()` — выводит текст, используя характеристики заливки (см. *разд. 7.2.1*). Форматы метода:

```
public void fillText(String text, double x, double y)
public void fillText(String text, double x, double y,
                    double maxWidth)
```

Параметр `text` задает выводимый текст (разделителем строк является символ `\n`), параметры `x` и `y` — координаты начальной точки, а параметр `maxWidth` — максимальную ширину поля. Пример ограничения ширины поля:

```
// String longStr = ...;
gc.fillText(longStr, 20.0, 270.0, 350.0);
```

Если текст не помещается в указанную ширину, то он будет сжат таким образом, чтобы поместиться. В результате текст может стать абсолютно не читаемым. Если необходимо ограничить ширину и при этом обеспечить, чтобы текст не сжимался, а просто обрезался, то следует наложить маску:

```
gc.save();
gc.beginPath();
gc.rect(15.0, 280.0, 350.0, 30.0);
gc.clip();
gc.fillText(longStr, 20.0, 300.0);
gc.restore();
```

Метод `save()` сохраняет в стеке текущие значения различных характеристик. Метод `rect()` определяет границы маски, а метод `clip()` применяет маску. После вывода текста вызывается метод `restore()`, который восстанавливает значения различных характеристик, сохраненных ранее методом `save()`;

- `strokeText()` — выводит текст, используя характеристики обводки (см. *разд. 7.2.2*).
Форматы метода:

```
public void strokeText(String text, double x, double y)
public void strokeText(String text, double x, double y,
                    double maxWidth)
```

Пример:

```
gc.setStroke(Color.RED);
gc.setLineWidth(3.0);
gc.setFont(new Font(96.0));
gc.strokeText("OK", 20.0, 390.0);
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH07_01_Canvas`, приведенном в каталоге `CH07_canvas` сопровождающего книгу электронного архива (см. *приложение*).

7.2.8. Вывод изображения

Вывести изображение на холст позволяет метод `drawImage()` из класса `GraphicsContext`.
Форматы метода:

```
public void drawImage(Image img, double x, double y)
public void drawImage(Image img,
                    double x, double y, double w, double h)
```

```
public void drawImage(Image img,
                     double sx, double sy, double sw, double sh,
                     double dx, double dy, double dw, double dh)
```

Первый формат выводит изображение полностью в позицию с координатами x и y .
Пример:

```
WritableImage img = new WritableImage(100, 100);
// Фрагмент опущен
gc.drawImage(img, 20.0, 20.0);
```

Второй формат позволяет ограничить область вывода шириной (параметр w) и высотой (параметр h). При этом изображение может быть уменьшено или увеличено таким образом, чтобы вписаться в область. Если пропорции области не совпадают с пропорциями изображения, то изображение будет растянуто или сжато без соблюдения пропорций. Пример:

```
gc.drawImage(img, 20.0, 150.0, 50.0, 50.0);
gc.drawImage(img, 20.0, 230.0, 50.0, 100.0);
gc.drawImage(img, 20.0, 340.0, 100.0, 50.0);
```

Третий формат берет из изображения прямоугольную область (sx , sy , sw , sh) и вписывает ее в прямоугольную область (dx , dy , dw , dh) на холсте. Пример:

```
gc.drawImage(img, 0.0, 0.0, 50.0, 50.0,
             200.0, 20.0, 50.0, 50.0);
gc.drawImage(img, 0.0, 0.0, 50.0, 50.0,
             200.0, 100.0, 100.0, 100.0);
```

Начиная с JavaFX 12, доступны методы `setImageSmoothing()` и `isImageSmoothing()`, которые управляют сглаживанием изображения при масштабировании. По умолчанию сглаживание включено. Форматы методов:

```
public void setImageSmoothing(boolean imageSmoothing)
public boolean isImageSmoothing()
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH07_01_Canvas`, приведенном в каталоге `CH07_canvas` сопровождающего книгу электронного архива (см. приложение).

7.2.9. Очистка прямоугольной области или всего холста

Стереть какую-либо прямоугольную область на холсте позволяет метод `clearRect()` из класса `GraphicsContext`. Формат метода:

```
public void clearRect(double x, double y, double w, double h)
```

Пример очистки всего холста:

```
gc.clearRect(0.0, 0.0, canvas.getWidth(), canvas.getHeight());
```

7.2.10. Сохранение и восстановление состояния

Значения основных характеристик можно сохранить в стек, выполнить какую-либо операцию рисования, а затем восстановить эти значения из стека. Для этого предназначены следующие методы из класса `GraphicsContext`:

- ❑ `save()` — сохраняет значения основных характеристик (в частности, характеристики заливки, обводки, текста и др.) в стек. Полный список сохраняемых характеристик можно найти в документации. Формат метода:

```
public void save()
```

- ❑ `restore()` — восстанавливает значения основных характеристик из стека. Формат метода:

```
public void restore()
```

Например, в предыдущих разделах мы ограничивали область вывода длинного текста маской с помощью следующего кода:

```
gc.save();  
gc.beginPath();  
gc.rect(15.0, 280.0, 350.0, 30.0);  
gc.clip();  
gc.fillText(longStr, 20.0, 300.0);  
gc.restore();
```

Вначале мы сохранили значения в стек, применили маску к тексту, а затем восстановили значения из стека. Если убрать последнюю инструкцию восстановления значений, то все последующие операции рисования будут отображаться только внутри маски, а не на всей поверхности холста.

7.2.11. Применение эффектов и трансформаций

Применить различные эффекты и преобразования позволяют следующие методы из класса `GraphicsContext`:

- ❑ `setGlobalAlpha()` — задает степень непрозрачности: вещественное число от 0.0 (полностью прозрачный) до 1.0 (полностью непрозрачный). Значение по умолчанию: 1.0. Формат метода:

```
public void setGlobalAlpha(double alpha)
```

Пример:

```
gc.save();  
gc.setFill(Color.BLUE);  
gc.setGlobalAlpha(0.5);  
gc.fillRect(10.0, 10.0, 20.0, 20.0);  
gc.restore();
```

- ❑ `getGlobalAlpha()` — возвращает степень непрозрачности. Формат метода:

```
public double getGlobalAlpha()
```


- ❑ `setGlobalBlendMode()` — задает режим наложения. Напомню, что режимы наложения мы уже рассматривали в *разд. 5.6*. Значение по умолчанию: `BlendMode.SRC_OVER`. Формат метода:

```
import javafx.scene.effect.BlendMode;
public void setGlobalBlendMode(BlendMode op)
```

Пример:

```
gc.setFill(Color.GREEN);
gc.fillRect(300.0, 300.0, 40.0, 40.0);
gc.save();
gc.setFill(Color.BLACK);
gc.setGlobalBlendMode(BlendMode.OVERLAY);
gc.fillRect(320.0, 280.0, 40.0, 40.0);
gc.restore();
```

- ❑ `getGlobalBlendMode()` — возвращает режим наложения. Формат метода:

```
public BlendMode getGlobalBlendMode()
```

- ❑ `scale()` — изменяет масштабирование. Формат метода:

```
public void scale(double x, double y)
```

Пример увеличения масштаба в два раза:

```
gc.save();
gc.scale(2.0, 2.0);
gc.fillRect(10.0, 40.0, 40.0, 40.0);
gc.restore();
```

- ❑ `rotate()` — применяет трансформацию вращения. По умолчанию вращение выполняется относительно левого верхнего угла холста. Формат метода:

```
public void rotate(double degrees)
```

Пример вращения прямоугольника на 45 градусов относительно левого верхнего угла холста:

```
gc.save();
gc.rotate(45.0);
gc.fillRect(40.0, -20.0, 40.0, 40.0);
gc.restore();
```

- ❑ `translate()` — сдвигает систему координат. Формат метода:

```
public void translate(double x, double y)
```

Пример вращения прямоугольника на 45 градусов относительно точки вставки:

```
gc.save();
gc.setStroke(Color.GREEN);
gc.translate(200.0, 200.0);
gc.rotate(45.0);
gc.strokeRect(0, 0, 40.0, 40.0);
gc.restore();
```

- `setTransform()` — задает матрицу трансформации. Форматы метода:

```
import javafx.scene.transform.Affine;
public void setTransform(double mxx, double myx,
                        double mxy, double myy,
                        double mxt, double myt)
public void setTransform(Affine xform)
```

Пример вращения прямоугольника на 45 градусов относительно центра прямоугольника:

```
// import javafx.scene.transform.Rotate;
gc.save();
gc.setFill(Color.BLUE);
double x = 50.0, y = 50.0, width = 40.0, height = 40.0;
Rotate r = new Rotate(45.0, x + width / 2, y + height / 2);
gc.setTransform(r.getMxx(), r.getMyx(),
               r.getMxy(), r.getMyy(),
               r.getTx(), r.getTy());
gc.fillRect(x, y, width, height);
gc.restore();
```

ПРИМЕЧАНИЕ

Подробно трансформации и эффекты мы рассмотрим в следующей главе.

- `getTransform()` — возвращает текущую матрицу трансформации. Форматы метода:

```
public Affine getTransform()
public Affine getTransform(Affine xform)
```

Выведем матрицу трансформации по умолчанию:

```
System.out.println(gc.getTransform());
/*
Affine [
    1.0, 0.0, 0.0, 0.0
    0.0, 1.0, 0.0, 0.0
    0.0, 0.0, 1.0, 0.0
]
*/
```

- `transform()` — изменяет текущую матрицу трансформации. Форматы метода:

```
public void transform(double mxx, double myx,
                     double mxy, double myy,
                     double mxt, double myt)
public void transform(Affine xform)
```

Пример увеличения масштаба в два раза:

```
gc.save();
gc.transform(2.0, 0.0, 0.0, 2.0, 0.0, 0.0);
```

```
gc.setStroke(Color.BLUE);  
gc.strokeRect(140.0, 140.0, 40.0, 40.0);  
gc.restore();
```

В этом примере инструкцию:

```
gc.transform(2.0, 0.0, 0.0, 2.0, 0.0, 0.0);
```

лучше заменить более понятной инструкцией:

```
gc.scale(2.0, 2.0);
```

- ❑ `setEffect()` — задает эффект для последующих операций рисования. Чтобы отключить эффект, достаточно передать значение `null` вместо эффекта. Формат метода:

```
public void setEffect(Effect e)
```

Выведем прямоугольник с эффектом тени:

```
gc.save();  
gc.setEffect(new DropShadow());  
gc.setFill(Color.CORAL);  
gc.fillRect(50.0, 300.0, 40.0, 40.0);  
gc.restore();
```

- ❑ `getEffect()` — возвращает текущий эффект или значение `null`. Формат метода:

```
public Effect getEffect(Effect e)
```

- ❑ `applyEffect()` — применяет эффект ко всему холсту. Формат метода:

```
public void applyEffect(Effect e)
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH07_01_Canvas`, приведенном в каталоге `CH07_canvas` сопровождающего книгу электронного архива (см. *приложение*).

ГЛАВА 8



Трансформации и эффекты

В предыдущей главе мы научились применять различные трансформации и эффекты к элементам холста. Например: сдвигать, масштабировать, вращать, накладывать тень и др. Применить трансформации и эффекты можно не только к элементам холста, но и к обычным узлам. Давайте рассмотрим трансформации и эффекты подробно.

ПРИМЕЧАНИЕ

Изменение степени непрозрачности узла и режима наложения узла также можно отнести к трансформациям и эффектам. Напомню, что изменение непрозрачности мы уже рассматривали в разд. 3.7, а режимы наложения — в разд. 5.6, поэтому не будем повторяться.

8.1. Трансформации

Применить к узлу трансформации смещения, масштабирования и вращения можно с помощью специализированных методов из класса `Node`, особо не вникая в детали реализации. Если же мы хотим разобраться в деталях и научиться применять произвольные трансформации, то необходимо будет изучить класс `Transform` и его производные классы: `Affine`, `Translate`, `Scale`, `Rotate` и `Shear`.

8.1.1. Методы класса *Node*

Управлять трансформациями узлов позволяют следующие свойства из класса `Node`:

□ `translateX` — смещение по оси `X`. Методы доступа:

```
public final void setTranslateX(double value)
public final double getTranslateX()
public final DoubleProperty translateXProperty()
```

□ `translateY` — смещение по оси `Y`. Методы доступа:

```
public final void setTranslateY(double value)
public final double getTranslateY()
public final DoubleProperty translateYProperty()
```

Методы `setTranslateX()` и `setTranslateY()` смещают узел на некоторое расстояние относительно текущего местоположения. Их очень удобно использовать для создания анимации. Значения смещений прибавляются к исходным координатам, заданным методами `setLayoutX()`, `setLayoutY()` или `relocate()`, и не изменяют исходные координаты. Чтобы вернуть узел на первоначальное место, достаточно присвоить смещению значение 0.0. Пример:

```
Rectangle rect = new Rectangle(100.0, 100.0);
rect.setFill(Color.RED);
rect.setTranslateX(10.0);
rect.setTranslateY(10.0);
```

❑ **translateZ — смещение по оси z. Методы доступа:**

```
public final void setTranslateZ(double value)
public final double getTranslateZ()
public final DoubleProperty translateZProperty()
```

❑ **scaleX — масштаб по оси x. Методы доступа:**

```
public final void setScaleX(double value)
public final double getScaleX()
public final DoubleProperty scaleXProperty()
```

❑ **scaleY — масштаб по оси y. Методы доступа:**

```
public final void setScaleY(double value)
public final double getScaleY()
public final DoubleProperty scaleYProperty()
```

Пример увеличения масштаба в два раза:

```
Rectangle rect2 = new Rectangle(50.0, 50.0);
rect2.setFill(Color.BLACK);
rect2.setScaleX(2.0);
rect2.setScaleY(2.0);
```

❑ **scaleZ — масштаб по оси z. Методы доступа:**

```
public final void setScaleZ(double value)
public final double getScaleZ()
public final DoubleProperty scaleZProperty()
```

❑ **rotate — задает угол поворота в градусах относительно центра узла. Методы доступа:**

```
public final void setRotate(double value)
public final double getRotate()
public final DoubleProperty rotateProperty()
```

Пример поворота прямоугольника на 45 градусов:

```
Rectangle rect3 = new Rectangle(50.0, 50.0);
rect3.setFill(Color.GREEN);
rect3.setRotate(45.0);
```

По умолчанию вращение осуществляется вокруг оси *z*. Если необходимо вращать вокруг другой оси, то предварительно нужно указать ось с помощью свойства `rotationAxis`. Методы доступа:

```
import javafx.scene.transform.Rotate;
public final void setRotationAxis(Point3D value)
public final Point3D getRotationAxis()
public final ObjectProperty<Point3D> rotationAxisProperty()
```

В параметре `value` можно указать константы: `X_AXIS`, `Y_AXIS` или `Z_AXIS` из класса `Rotate` или объект класса `Point3D`. Значение по умолчанию: `Rotate.Z_AXIS`. Пример вращения вокруг оси *x*:

```
Rectangle rect3_2 = new Rectangle(50.0, 50.0);
rect3_2.setFill(Color.GREEN);
rect3_2.setRotationAxis(Rotate.X_AXIS);
rect3_2.setRotate(45.0);
```

Чтобы одновременно вращать по нескольким осям, можно воспользоваться классом `Rotate`. Пример:

```
Rectangle rect3_3 = new Rectangle(50.0, 50.0);
rect3_3.setFill(Color.GREEN);
Rotate rx = new Rotate(30.0, Rotate.X_AXIS);
Rotate ry = new Rotate(50.0, Rotate.Y_AXIS);
Rotate rz = new Rotate(30.0, Rotate.Z_AXIS);
rect3_3.getTransforms().addAll(rx, ry, rz);
```

Метод `getTransforms()` возвращает ссылку на список с трансформациями. В этот список мы можем добавить нужные трансформации. Формат метода:

```
public final ObservableList<Transform> getTransforms()
```

Пример добавления трансформации сдвига по оси *x*:

```
Rectangle rect4 = new Rectangle(100.0, 100.0);
Shear sh = new Shear();
sh.setX(0.5);
rect4.getTransforms().add(sh);
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH08_01_Transform`, приведенном в каталоге `CH08_effects` сопровождающего книгу электронного архива (см. приложение).

8.1.2. Классы *Transform* и *Affine*

Абстрактный класс `Transform` является базовым классом для всех классов, реализующих трансформации. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.transform.Transform;
```

Трансформация задается с помощью матрицы. Типы матриц описываются в перечислении `javafx.scene.transform.MatrixType` (содержит константы: `MT_2D_2x3`, `MT_2D_3x3`, `MT_3D_3x4` и `MT_3D_4x4`). Смещение, масштабирование, вращение и сдвиг реализуются с помощью матрицы `MT_3D_3x4`. Матрица имеет следующий вид:

```
mxx mxy mxz tx
myx myy myz ty
mzx mzy mzz tz
```

Вычисление координат с учетом трансформации производится так:

```
x = mxx * x + mxy * y + mxz * z + tx
y = myx * x + myy * y + myz * z + ty
z = mzx * x + mzy * y + mzz * z + tz
```

Создать матрицу `MT_3D_3x4` позволяет класс `Affine`. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.transform.Affine;
```

Иерархия наследования:

Object — Transform — Affine

Создать объект позволяют следующие конструкторы:

```
Affine()
Affine(double[] matrix, MatrixType type, int offset)
Affine(double mxx, double mxy, double tx,
        double myx, double myy, double ty)
Affine(double mxx, double mxy, double mxz, double tx,
        double myx, double myy, double myz, double ty,
        double mzx, double mzy, double mzz, double tz)
Affine(Transform transform)
```

Первый конструктор создает матрицу без трансформации:

```
Affine m = new Affine();
System.out.println(m.toString());
/*
Affine [
    1.0, 0.0, 0.0, 0.0
    0.0, 1.0, 0.0, 0.0
    0.0, 0.0, 1.0, 0.0
] */
```

Если мы подставим значения в формулы вычисления координат, то можем убедиться, что трансформаций нет, и координаты не меняются:

```
// x = mxx * x + mxy * y + mxz * z + tx
x      = 1.0 * x + 0.0 * y + 0.0 * z + 0.0; // x == x
```

Остальные конструкторы заполняют матрицу конкретными значениями. Создадим трансформацию смещения на 100 px по осям X и Y:

```

Affine m2 = new Affine(1.0, 0.0, 100.0, 0.0, 1.0, 100.0);
System.out.println(m2.toString());
/*
Affine [
    1.0, 0.0, 0.0, 100.0
    0.0, 1.0, 0.0, 100.0
    0.0, 0.0, 1.0, 0.0
] */

```

Создать экземпляр класса Affine позволяет также статический метод affine() из класса Transform. Форматы метода:

```

public static Affine affine(double mxx, double myx,
                           double mxy, double myy,
                           double tx, double ty)

public static Affine affine(
    double mxx, double mxy, double mxz, double tx,
    double myx, double myy, double myz, double ty,
    double mzx, double mzy, double mzz, double tz)

```

Создадим трансформацию масштабирования в два раза по осям X и Y:

```

Affine m3 = Transform.affine(2.0, 0.0, 0.0, 2.0, 0.0, 0.0);
System.out.println(m3.toString());
/*
Affine [
    2.0, 0.0, 0.0, 0.0
    0.0, 2.0, 0.0, 0.0
    0.0, 0.0, 1.0, 0.0
] */

```

Задать значения отдельных элементов матрицы после создания объекта позволяют следующие методы из класса Affine:

```

// mxx mxy mxz tx
public final void setMxx(double value)
public final void setMxy(double value)
public final void setMxz(double value)
public final void setTx(double value)
// myx myy myz ty
public final void setMyx(double value)
public final void setMyy(double value)
public final void setMyz(double value)
public final void setTy(double value)
// mzx mzy mzz tz
public final void setMzx(double value)
public final void setMzy(double value)
public final void setMzz(double value)
public final void setTz(double value)

```


Получить значения отдельных элементов матрицы позволяют следующие методы из класса `Transform` (методы наследуются всеми производными классами):

```
// mxx mxy mxz tx
public double getMxx()
public double getMxy()
public double getMxz()
public double getTx()
// myx myy myz ty
public double getMyx()
public double getMyy()
public double getMyz()
public double getTy()
// mzx mzy mzz tz
public double getMzx()
public double getMzy()
public double getMzz()
public double getTz()
```

Получить координаты с учетом значений матрицы трансформации позволяют следующие методы из класса `Transform` (методы наследуются всеми производными классами):

```
public Point2D transform(double x, double y)
public Point2D transform(Point2D point)
public Point3D transform(double x, double y, double z)
public Point3D transform(Point3D point)
public Bounds transform(Bounds bounds)
```

Пример:

```
Affine m2 = new Affine(1.0, 0.0, 100.0, 0.0, 1.0, 100.0);
System.out.println(m2.transform(new Point2D(5.0, 10.0)));
// Point2D [x = 105.0, y = 110.0]
```

Классы `Transform` и `Affine` содержат множество вспомогательных методов, которые будут полезны разработчикам трансформаций. За подробной информацией обращайтесь к документации.

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH08_01_Transform`, приведенном в каталоге `CH08_effects` сопровождающего книгу электронного архива (см. приложение).

8.1.3. Класс *Translate*: смещение

Класс `Translate` описывает трансформацию смещения. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.transform.Translate;
```

Иерархия наследования:

`Object` — `Transform` — `Translate`

Создаваемая матрица выглядит следующим образом:

```
1   0   0   x
0   1   0   y
0   0   1   z
```

Конструкторы класса:

```
Translate()
Translate(double x, double y)
Translate(double x, double y, double z)
```

Пример смещения по оси X на 100 px, а по оси Y — на 50 px:

```
Translate tr2 = new Translate(100.0, 50.0);
System.out.println(tr2.toString());
// Translate [x=100.0, y=50.0, z=0.0]
Rectangle rect2 = new Rectangle(50.0, 50.0);
rect2.setFill(Color.GREEN);
rect2.getTransforms().add(tr2);
```

Создать объект позволяет также статический метод `translate()` из класса `Transform`. Формат метода:

```
public static Translate translate(double x, double y)
```

Пример:

```
Translate tr4 = Transform.translate(300.0, 150.0);
```

Управлять смещением позволяют следующие свойства:

□ **x** — смещение по оси X. Методы доступа:

```
public final void setX(double value)
public final double getX()
public final DoubleProperty xProperty()
```

□ **y** — смещение по оси Y. Методы доступа:

```
public final void setY(double value)
public final double getY()
public final DoubleProperty yProperty()
```

□ **z** — смещение по оси Z. Методы доступа:

```
public final void setZ(double value)
public final double getZ()
public final DoubleProperty zProperty()
```

Не забывайте, что применить трансформацию смещения к узлу можно с помощью методов: `setTranslateX()`, `setTranslateY()` и `setTranslateZ()` из класса `Node` (см. разд. 8.1.1).

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH08_01_Transform`, приведенном в каталоге `CH08_effects` сопровождающего книгу электронного архива (см. приложение).

8.1.4. Класс *Scale*: масштабирование

Класс *Scale* описывает трансформацию масштабирования. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.transform.Scale;
```

Иерархия наследования:

Object — Transform — Scale

Создаваемая матрица выглядит следующим образом:

```
x    0    0    ((1 - x) * pivotX)
0    y    0    ((1 - y) * pivotY)
0    0    z    ((1 - z) * pivotZ)
```

Конструкторы класса:

```
Scale()
Scale(double x, double y)
Scale(double x, double y, double pivotX, double pivotY)
Scale(double x, double y, double z)
Scale(double x, double y, double z,
      double pivotX, double pivotY, double pivotZ)
```

Выполним масштабирование прямоугольника в два раза относительно центра прямоугольника:

```
Scale tr5 = new Scale(2.0, 2.0, 1.0, 25.0, 25.0, 0.0);
System.out.println(tr5.toString());
// Scale [x=2.0, y=2.0, z=1.0, pivotX=25.0, pivotY=25.0, pivotZ=0.0]
Rectangle rect5 = new Rectangle(50.0, 50.0);
rect5.setFill(Color.BLACK);
rect5.getTransforms().add(tr5);
```

Создать объект позволяет также статический метод `scale()` из класса *Transform*. Форматы метода:

```
public static Scale scale(double x, double y)
public static Scale scale(double x, double y,
                        double pivotX, double pivotY)
```

Управлять масштабированием позволяют следующие свойства:

❑ **x** — масштаб по оси *x*. Значение по умолчанию: 1.0. Методы доступа:

```
public final void setX(double value)
public final double getX()
public final DoubleProperty xProperty()
```

❑ **y** — масштаб по оси *y*. Значение по умолчанию: 1.0. Методы доступа:

```
public final void setY(double value)
public final double getY()
public final DoubleProperty yProperty()
```

□ **z** — масштаб по оси **z**. Значение по умолчанию: 1.0. Методы доступа:

```
public final void setZ(double value)
public final double getZ()
public final DoubleProperty zProperty()
```

□ **pivotX** — координата опорной точки (точки, относительно которой производится масштабирование) по оси **x**. Значение по умолчанию: 0.0. Методы доступа:

```
public final void setPivotX(double value)
public final double getPivotX()
public final DoubleProperty pivotXProperty()
```

□ **pivotY** — координата опорной точки по оси **y**. Значение по умолчанию: 0.0. Методы доступа:

```
public final void setPivotY(double value)
public final double getPivotY()
public final DoubleProperty pivotYProperty()
```

□ **pivotZ** — координата опорной точки по оси **z**. Значение по умолчанию: 0.0. Методы доступа:

```
public final void setPivotZ(double value)
public final double getPivotZ()
public final DoubleProperty pivotZProperty()
```

Не забывайте, что применить трансформацию масштабирования к узлу можно с помощью методов: `setScaleX()`, `setScaleY()` и `setScaleZ()` из класса `Node` (см. *разд. 8.1.1*), не создавая объект класса `Scale` явным образом. Однако вы не сможете этим способом задать координаты опорной точки.

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH08_01_Transform`, приведенном в каталоге `CH08_effects` сопровождающего книгу электронного архива (см. *приложение*).

8.1.5. Класс *Rotate*: вращение

Класс `Rotate` описывает трансформацию вращения. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.transform.Rotate;
```

Иерархия наследования:

Object — Transform — Rotate

Матрица вращения вокруг оси **z** выглядит следующим образом:

```
// theta = Math.toRadians(angle);
cos(theta)  -sin(theta)  0    (x - x * cos(theta) + y * sin(theta))
sin(theta)   cos(theta)  0    (y - x * sin(theta) - y * cos(theta))
0            0          1            z
```

Конструкторы класса:

```
Rotate()  
Rotate(double angle)  
Rotate(double angle, double pivotX, double pivotY)  
Rotate(double angle, double pivotX, double pivotY, double pivotZ)  
Rotate(double angle, Point3D axis)  
Rotate(double angle, double pivotX, double pivotY, double pivotZ,  
        Point3D axis)
```

Пример вращения прямоугольника на 45 градусов относительно центра прямоугольника:

```
Rotate tr3 = new Rotate(45.0, 25.0, 25.0);  
System.out.println(tr3.toString());  
// Rotate [angle=45.0, pivotX=25.0, pivotY=25.0, pivotZ=0.0,  
// axis=Point3D [x = 0.0, y = 0.0, z = 1.0]]  
Rectangle rect3 = new Rectangle(50.0, 50.0);  
rect3.setFill(Color.BLACK);  
rect3.getTransforms().add(tr3);
```

Создать объект позволяет также статический метод `rotate()` из класса `Transform`. Формат метода:

```
public static Rotate rotate(double angle, double pivotX, double pivotY)
```

Пример вращения на 45 градусов относительно начала координат:

```
Rotate tr7 = Transform.rotate(45.0, 0.0, 0.0);
```

Управлять вращением позволяют следующие свойства:

- ❑ **angle** — угол вращения в градусах. Значение по умолчанию: 0.0. Методы доступа:

```
public final void setAngle(double value)  
public final double getAngle()  
public final DoubleProperty angleProperty()
```

- ❑ **axis** — задает ось, вокруг которой выполняется вращение. В параметре `value` можно указать константы: `X_AXIS`, `Y_AXIS` или `Z_AXIS` из класса `Rotate` или объект класса `Point3D`. Значение по умолчанию: `Rotate.Z_AXIS`. Методы доступа:

```
public final void setAxis(Point3D value)  
public final Point3D getAxis()  
public final ObjectProperty<Point3D> axisProperty()
```

Выведем значения констант:

```
System.out.println(Rotate.X_AXIS);  
// Point3D [x = 1.0, y = 0.0, z = 0.0]  
System.out.println(Rotate.Y_AXIS);  
// Point3D [x = 0.0, y = 1.0, z = 0.0]  
System.out.println(Rotate.Z_AXIS);  
// Point3D [x = 0.0, y = 0.0, z = 1.0]
```

- ❑ `pivotX` — координата опорной точки (точки, относительно которой производится вращение) по оси `x`. Значение по умолчанию: `0.0`. Методы доступа:

```
public final void setPivotX(double value)
public final double getPivotX()
public final DoubleProperty pivotXProperty()
```

- ❑ `pivotY` — координата опорной точки по оси `y`. Значение по умолчанию: `0.0`. Методы доступа:

```
public final void setPivotY(double value)
public final double getPivotY()
public final DoubleProperty pivotYProperty()
```

- ❑ `pivotZ` — координата опорной точки по оси `z`. Значение по умолчанию: `0.0`. Методы доступа:

```
public final void setPivotZ(double value)
public final double getPivotZ()
public final DoubleProperty pivotZProperty()
```

Не забывайте, что применить трансформацию вращения к узлу можно с помощью метода `setRotate()` из класса `Node` (см. *разд. 8.1.1*), не создавая объект класса `Rotate` явным образом. Однако вы не сможете этим способом задать координаты опорной точки.

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH08_01_Transform`, приведенном в каталоге `CH08_effects` сопровождающего книгу электронного архива (см. *приложение*).

8.1.6. Класс *Shear*: сдвиг

Класс `Shear` описывает трансформацию сдвига. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.transform.Shear;
```

Иерархия наследования:

`Object` — `Transform` — `Shear`

Матрица выглядит следующим образом:

```
1   x   0   (-x * pivotY)
y   1   0   (-y * pivotX)
0   0   1           0
```

Конструкторы класса:

```
Shear()
Shear(double x, double y)
Shear(double x, double y, double pivotX, double pivotY)
```

Пример сдвига по оси x:

```
Shear tr2 = new Shear(-0.5, 0);
System.out.println(tr2.toString());
// Shear [x=-0.5, y=0.0, pivotX=0.0, pivotY=0.0]
```

Создать объект позволяет также статический метод `shear()` из класса `Transform`.
Форматы метода:

```
public static Shear shear(double x, double y)
public static Shear shear(double x, double y,
                          double pivotX, double pivotY)
```

Пример сдвига по оси y:

```
Shear tr5 = Transform.shear(0.0, -0.5, 25.0, 25.0);
```

Управлять трансформацией позволяют следующие свойства:

- ❑ `x` — сдвиг по оси `x` (вещественное число от -1.0 до 1.0). Значение по умолчанию: `0.0`. Методы доступа:

```
public final void setX(double value)
public final double getX()
public final DoubleProperty xProperty()
```

- ❑ `y` — сдвиг по оси `y` (вещественное число от -1.0 до 1.0). Значение по умолчанию: `0.0`. Методы доступа:

```
public final void setY(double value)
public final double getY()
public final DoubleProperty yProperty()
```

- ❑ `pivotX` — координата опорной точки по оси `x`. Значение по умолчанию: `0.0`. Методы доступа:

```
public final void setPivotX(double value)
public final double getPivotX()
public final DoubleProperty pivotXProperty()
```

- ❑ `pivotY` — координата опорной точки по оси `y`. Значение по умолчанию: `0.0`. Методы доступа:

```
public final void setPivotY(double value)
public final double getPivotY()
public final DoubleProperty pivotYProperty()
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH08_01_Transform`, приведенном в каталоге `CH08_effects` сопровождающего книгу электронного архива (см. приложение).

8.2. Эффекты

Помимо трансформации мы можем применить к узлу различные эффекты — например, добавить внешнюю или внутреннюю тень, создать зеркальное отражение и др.

Базовым классом для всех классов, реализующих эффекты, является абстрактный класс `Effect`. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.effect.Effect;
```

Проверить возможность использования эффектов позволяет следующий код:

```
// import javafx.application.ConditionalFeature;
// import javafx.application.Platform;
if (Platform.isSupported(ConditionalFeature.EFFECT)) {
    System.out.println("Эффекты поддерживаются");
}
else {
    System.out.println("Нет поддержки");
}
```

8.2.1. Методы класса *Node*

Управлять эффектами позволяет свойство `effect` из класса `Node`. Методы доступа:

```
public final void setEffect(Effect value)
public final Effect getEffect()
public final ObjectProperty<Effect> effectProperty()
```

Пример добавления внешней тени к прямоугольнику:

```
Rectangle rect = new Rectangle(50.0, 50.0);
rect.setFill(Color.RED);
rect.setEffect(new DropShadow());
```

ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH08_02_Effect`, приведенном в каталоге `CH08_effects` сопровождающего книгу электронного архива (см. приложение).

8.2.2. Класс *DropShadow*: внешняя тень

Класс `DropShadow` реализует эффект отбрасывания внешней тени. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.effect.DropShadow;
```

Иерархия наследования:

```
Object — Effect — DropShadow
```


Конструкторы класса:

```
DropShadow()  
DropShadow(double radius, Color color)  
DropShadow(double radius, double offsetX, double offsetY,  
            Color color)  
DropShadow(BlurType blurType, Color color, double radius,  
            double spread, double offsetX, double offsetY)
```

Пример создания тени со смещением:

```
DropShadow effect3 = new DropShadow(20.0, 10.0, 10.0, Color.BLACK);  
Rectangle rect3 = new Rectangle(50.0, 50.0);  
rect3.setFill(Color.RED);  
rect3.setEffect(effect3);
```

Управлять эффектом позволяют следующие свойства:

- ❑ **blurType** — задает алгоритм, используемый для размытия тени. В качестве значения можно указать константы: `ONE_PASS_BOX`, `TWO_PASS_BOX`, `THREE_PASS_BOX` (значение по умолчанию) или `GAUSSIAN` из перечисления `BlurType`. Методы доступа:

```
import javafx.scene.effect.BlurType;  
public final void setBlurType(BlurType value)  
public final BlurType getBlurType()  
public final ObjectProperty<BlurType> blurTypeProperty()
```

- ❑ **color** — задает цвет тени. Значение по умолчанию: `Color.BLACK`. Методы доступа:

```
public final void setColor(Color value)  
public final Color getColor()  
public final ObjectProperty<Color> colorProperty()
```

- ❑ **radius** — радиус размытия тени. Можно указать значение от 0.0 до 127.0. Значение по умолчанию: 10.0. Методы доступа:

```
public final void setRadius(double value)  
public final double getRadius()  
public final DoubleProperty radiusProperty()
```

Установка значения радиуса автоматически изменяет значения свойств `width` и `height` по формуле: $2 * radius + 1$. Методы доступа к свойствам `width` и `height`:

```
public final void setWidth(double value)  
public final double getWidth()  
public final DoubleProperty widthProperty()  
public final void setHeight(double value)  
public final double getHeight()  
public final DoubleProperty heightProperty()
```

- ❑ **offsetX** — смещение тени по оси x. Значение по умолчанию: 0.0. Методы доступа:

```
public final void setOffsetX(double value)  
public final double getOffsetX()  
public final DoubleProperty offsetXProperty()
```

- **offsetY** — смещение тени по оси *Y*. Значение по умолчанию: 0.0. Методы доступа:
- ```
public final void setOffsetY(double value)
public final double getOffsetY()
public final DoubleProperty offsetYProperty()
```
- **spread** — если свойство имеет значение 0.0, то тень полностью определяется алгоритмом размытия, а если 1.0 — то размытие отсутствует. Промежуточные значения между 0.0 и 1.0 позволяют контролировать степень размытия. Значение по умолчанию: 0.0. Методы доступа:
- ```
public final void setSpread(double value)
public final double getSpread()
public final DoubleProperty spreadProperty()
```
- **input** — задает входные данные для эффекта. Если свойство имеет значение `null` (значение по умолчанию), то в качестве входных данных используется изображение узла, к которому применяется эффект. Методы доступа:
- ```
public final void setInput(Effect value)
public final Effect getInput()
public final ObjectProperty<Effect> inputProperty()
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH08_02_Effect`, приведенном в каталоге `CH08_effects` сопровождающего книгу электронного архива (см. приложение).

## 8.2.3. Класс *InnerShadow*: внутренняя тень

Класс `InnerShadow` реализует эффект внутренней тени. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.effect.InnerShadow;
```

Иерархия наследования:

`Object` — `Effect` — `InnerShadow`

Конструкторы класса:

```
InnerShadow()
InnerShadow(double radius, Color color)
InnerShadow(double radius, double offsetX, double offsetY,
 Color color)
InnerShadow(BlurType blurType, Color color, double radius,
 double choke, double offsetX, double offsetY)
```

Пример создания внутренней тени:

```
InnerShadow effect2 = new InnerShadow(40.0, Color.BLACK);
Rectangle rect2 = new Rectangle(50.0, 50.0);
rect2.setFill(Color.RED);
rect2.setEffect(effect2);
```

Управлять эффектом позволяют следующие свойства:

- **blurType** — задает алгоритм, используемый для размытия тени. В качестве значения можно указать константы: `ONE_PASS_BOX`, `TWO_PASS_BOX`, `THREE_PASS_BOX` (значение по умолчанию) или `GAUSSIAN` из перечисления `BlurType`. Методы доступа:

```
import javafx.scene.effect.BlurType;
public final void setBlurType(BlurType value)
public final BlurType getBlurType()
public final ObjectProperty<BlurType> blurTypeProperty()
```

- **color** — задает цвет тени. Значение по умолчанию: `Color.BLACK`. Методы доступа:

```
public final void setColor(Color value)
public final Color getColor()
public final ObjectProperty<Color> colorProperty()
```

- **radius** — радиус размытия тени. Можно указать значение от 0.0 до 127.0. Значение по умолчанию: 10.0. Методы доступа:

```
public final void setRadius(double value)
public final double getRadius()
public final DoubleProperty radiusProperty()
```

Установка значения радиуса автоматически изменяет значения свойств `width` и `height` по формуле:  $2 * radius + 1$ . Методы доступа к свойствам `width` и `height`:

```
public final void setWidth(double value)
public final double getWidth()
public final DoubleProperty widthProperty()
public final void setHeight(double value)
public final double getHeight()
public final DoubleProperty heightProperty()
```

- **offsetX** — смещение тени по оси *x*. Значение по умолчанию: 0.0. Методы доступа:

```
public final void setOffsetX(double value)
public final double getOffsetX()
public final DoubleProperty offsetXProperty()
```

- **offsetY** — смещение тени по оси *y*. Значение по умолчанию: 0.0. Методы доступа:

```
public final void setOffsetY(double value)
public final double getOffsetY()
public final DoubleProperty offsetYProperty()
```

- **choke** — если свойство имеет значение 0.0, то тень полностью определяется алгоритмом размытия, а если 1.0 — то размытие отсутствует. Промежуточные значения между 0.0 и 1.0 позволяют контролировать степень размытия. Значение по умолчанию: 0.0. Методы доступа:

```
public final void setChoke(double value)
public final double getChoke()
public final DoubleProperty chokeProperty()
```

- `input` — задает входные данные для эффекта. Если свойство имеет значение `null` (значение по умолчанию), то в качестве входных данных используется изображение узла, к которому применяется эффект. Методы доступа:

```
public final void setInput (Effect value)
public final Effect getInput()
public final ObjectProperty<Effect> inputProperty()
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH08_02_Effect`, приведенном в каталоге `CH08_effects` сопровождающего книгу электронного архива (см. приложение).

## 8.2.4. Класс *Shadow*: тень

Класс `Shadow` позволяет создать тень. В отличие от классов `DropShadow` и `InnerShadow`, класс `Shadow` просто превращает узел в тень, а не добавляет тень к узлу. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.effect.Shadow;
```

Иерархия наследования:

`Object` — `Effect` — `Shadow`

Конструкторы класса:

```
Shadow()
Shadow(double radius, Color color)
Shadow(BlurType blurType, Color color, double radius)
```

Пример создания тени синего цвета:

```
Shadow effect2 = new Shadow(40.0, Color.BLUE);
Rectangle rect2 = new Rectangle(50.0, 50.0);
rect2.setFill(Color.RED);
rect2.setEffect(effect2);
```

Управлять эффектом позволяют следующие свойства:

- `blurType` — задает алгоритм, используемый для размытия тени. В качестве значения можно указать константы: `ONE_PASS_BOX`, `TWO_PASS_BOX`, `THREE_PASS_BOX` (значение по умолчанию) или `GAUSSIAN` из перечисления `BlurType`. Методы доступа:

```
import javafx.scene.effect.BlurType;
public final void setBlurType(BlurType value)
public final BlurType getBlurType()
public final ObjectProperty<BlurType> blurTypeProperty()
```

- `color` — задает цвет тени. Значение по умолчанию: `Color.BLACK`. Методы доступа:

```
public final void setColor(Color value)
public final Color getColor()
public final ObjectProperty<Color> colorProperty()
```

- **radius** — радиус размытия тени. Можно указать значение от 0.0 до 127.0. Значение по умолчанию: 10.0. Методы доступа:

```
public final void setRadius(double value)
public final double getRadius()
public final DoubleProperty radiusProperty()
```

Установка значения радиуса автоматически изменяет значения свойств **width** и **height** по формуле:  $2 * \text{radius} + 1$ . Методы доступа к свойствам **width** и **height**:

```
public final void setWidth(double value)
public final double getWidth()
public final DoubleProperty widthProperty()
public final void setHeight(double value)
public final double getHeight()
public final DoubleProperty heightProperty()
```

- **input** — задает входные данные для эффекта. Если свойство имеет значение **null** (значение по умолчанию), то в качестве входных данных используется изображение узла, к которому применяется эффект. Методы доступа:

```
public final void setInput(Effekt value)
public final Effect getInput()
public final ObjectProperty<Effect> inputProperty()
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH08_02_Effect`, приведенном в каталоге `CH08_effects` сопровождающего книгу электронного архива (см. приложение).

## 8.2.5. Класс *Reflection*: зеркальное отражение

Класс `Reflection` реализует эффект зеркального отражения. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.effect.Reflection;
```

Иерархия наследования:

```
Object — Effect — Reflection
```

Конструкторы класса:

```
Reflection()
Reflection(double topOffset, double fraction, double topOpacity,
 double bottomOpacity)
```

Пример добавления зеркального отражения к тексту:

```
Reflection effect3 = new Reflection(-20.0, 0.7, 0.5, 0.0);
Text text = new Text("JavaFX");
text.setFont(new Font(40.0));
text.setEffect(effect3);
```

Управлять эффектом позволяют следующие свойства:

- ❑ **topOffset** — смещение отражения от нижней границы узла. Значение по умолчанию: 0.0. Методы доступа:

```
public final void setTopOffset(double value)
public final double getTopOffset()
public final DoubleProperty topOffsetProperty()
```

- ❑ **fraction** — часть узла, видимая в отражении. В качестве значения указывается вещественное число от 0.0 (для 0 %) до 1.0 (для 100 %). Значение по умолчанию: 0.75. Методы доступа:

```
public final void setFraction(double value)
public final double getFraction()
public final DoubleProperty fractionProperty()
```

- ❑ **topOpacity** — степень непрозрачности верхней части отражения. В качестве значения указывается вещественное число от 0.0 до 1.0. Значение по умолчанию: 0.5. Методы доступа:

```
public final void setTopOpacity(double value)
public final double getTopOpacity()
public final DoubleProperty topOpacityProperty()
```

- ❑ **bottomOpacity** — степень непрозрачности нижней части отражения. В качестве значения указывается вещественное число от 0.0 до 1.0. Значение по умолчанию: 0.0. Методы доступа:

```
public final void setBottomOpacity(double value)
public final double getBottomOpacity()
public final DoubleProperty bottomOpacityProperty()
```

- ❑ **input** — задает входные данные для эффекта. Если свойство имеет значение `null` (значение по умолчанию), то в качестве входных данных используется изображение узла, к которому применяется эффект. Методы доступа:

```
public final void setInput(Effect value)
public final Effect getInput()
public final ObjectProperty<Effect> inputProperty()
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH08_02_Effect`, приведенном в каталоге `CH08_effects` сопровождающего книгу электронного архива (см. *приложение*).

## 8.2.6. Класс *GaussianBlur*: размытие по Гауссу

Класс `GaussianBlur` реализует эффект размытия по Гауссу. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.effect.GaussianBlur;
```

### Иерархия наследования:

Object — Effect — GaussianBlur

### Конструкторы класса:

```
GaussianBlur()
GaussianBlur(double radius)
```

### Пример добавления эффекта к тексту:

```
Text text = new Text("JavaFX");
text.setFont(new Font(40.0));
text.setEffect(new GaussianBlur());
```

### Управлять эффектом позволяют следующие свойства:

- ❑ **radius** — задает радиус размытия. В качестве значения указывается вещественное число от 0.0 до 63.0. Значение по умолчанию: 10.0. Методы доступа:

```
public final void setRadius(double value)
public final double getRadius()
public final DoubleProperty radiusProperty()
```

- ❑ **input** — задает входные данные для эффекта. Если свойство имеет значение null (значение по умолчанию), то в качестве входных данных используется изображение узла, к которому применяется эффект. Методы доступа:

```
public final void setInput(Effect value)
public final Effect getInput()
public final ObjectProperty<Effect> inputProperty()
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH08_02_Effect`, приведенном в каталоге `CH08_effects` сопровождающего книгу электронного архива (см. *приложение*).

## 8.2.7. Класс *MotionBlur*: размытие в движении

Класс `MotionBlur` реализует эффект размытия в движении. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.effect.MotionBlur;
```

### Иерархия наследования:

Object — Effect — MotionBlur

### Конструкторы класса:

```
MotionBlur()
MotionBlur(double angle, double radius)
```

### Пример:

```
Text text = new Text("JavaFX");
text.setFont(new Font(40.0));
text.setEffect(new MotionBlur());
```

Управлять эффектом позволяют следующие свойства:

- **radius** — задает радиус размытия. В качестве значения указывается вещественное число от 0.0 до 63.0. Значение по умолчанию: 10.0. Методы доступа:

```
public final void setRadius(double value)
public final double getRadius()
public final DoubleProperty radiusProperty()
```

- **angle** — задает угол размытия в градусах. Значение по умолчанию: 0.0. Методы доступа:

```
public final void setAngle(double value)
public final double getAngle()
public final DoubleProperty angleProperty()
```

- **input** — задает входные данные для эффекта. Если свойство имеет значение `null` (значение по умолчанию), то в качестве входных данных используется изображение узла, к которому применяется эффект. Методы доступа:

```
public final void setInput(Effect value)
public final Effect getInput()
public final ObjectProperty<Effect> inputProperty()
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH08_02_Effect`, приведенном в каталоге `CH08_effects` сопровождающего книгу электронного архива (см. *приложение*).

## 8.2.8. Класс *BoxBlur*: размытие

Класс `BoxBlur` реализует эффект размытия. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.effect.BoxBlur;
```

Иерархия наследования:

`Object` — `Effect` — `BoxBlur`

Конструкторы класса:

```
BoxBlur()
BoxBlur(double width, double height, int iterations)
```

Пример:

```
Text text = new Text("JavaFX");
text.setFont(new Font(40.0));
text.setEffect(new BoxBlur(5.0, 5.0, 3));
```

Управлять эффектом позволяют следующие свойства:

- **width** — горизонтальный размер эффекта размытия. В качестве значения указывается вещественное число от 0.0 до 255.0. Значение по умолчанию: 5.0. Методы доступа:



```
public final void setWidth(double value)
public final double getWidth()
public final DoubleProperty widthProperty()
```

- ❑ **height** — вертикальный размер эффекта размытия. В качестве значения указывается вещественное число от 0.0 до 255.0. Значение по умолчанию: 5.0. Методы доступа:

```
public final void setHeight(double value)
public final double getHeight()
public final DoubleProperty heightProperty()
```

- ❑ **iterations** — количество применений эффекта. В качестве значения указывается целое число от 0 до 3. Значение по умолчанию: 1. Методы доступа:

```
public final void setIterations(int value)
public final int getIterations()
public final IntegerProperty iterationsProperty()
```

- ❑ **input** — задает входные данные для эффекта. Если свойство имеет значение null (значение по умолчанию), то в качестве входных данных используется изображение узла, к которому применяется эффект. Методы доступа:

```
public final void setInput(Effect value)
public final Effect getInput()
public final ObjectProperty<Effect> inputProperty()
```

### **ЭЛЕКТРОННЫЙ АРХИВ**

Полный текст программы содержится в проекте `FX_CH08_02_Effect`, приведенном в каталоге `CH08_effects` сопровождающего книгу электронного архива (см. *приложение*).

## **8.2.9. Класс *Bloom*: свечение**

Класс `Bloom` реализует эффект свечения. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.effect.Bloom;
```

**Иерархия наследования:**

Object — Effect — Bloom

**Конструкторы класса:**

```
Bloom()
Bloom(double threshold)
```

**Пример:**

```
Bloom effect = new Bloom();
Text text = new Text("JavaFX");
text.setFill(Color.WHITE);
text.setFont(Font.font("Verdana", FontWeight.BOLD, 40.0));
text.setEffect(effect);
```

Управлять эффектом позволяют следующие свойства:

- **threshold** — порог свечения. В качестве значения указывается вещественное число от 0.0 до 1.0. Значение по умолчанию: 0.3. Методы доступа:

```
public final void setThreshold(double value)
public final double getThreshold()
public final DoubleProperty thresholdProperty()
```

- **input** — задает входные данные для эффекта. Если свойство имеет значение `null` (значение по умолчанию), то в качестве входных данных используется изображение узла, к которому применяется эффект. Методы доступа:

```
public final void setInput(Effect value)
public final Effect getInput()
public final ObjectProperty<Effect> inputProperty()
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH08_02_Effect`, приведенном в каталоге `CH08_effects` сопровождающего книгу электронного архива (см. приложение).

## 8.2.10. Класс *Glow*: свечение

Класс `Glow` реализует эффект свечения. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.effect.Glow;
```

Иерархия наследования:

`Object` — `Effect` — `Glow`

Конструкторы класса:

```
Glow()
Glow(double level)
```

Пример:

```
Glow effect2 = new Glow(0.9);
Text text2 = new Text("JavaFX");
text2.setFill(Color.WHITE);
text2.setFont(Font.font("Verdana", FontWeight.BOLD, 40.0));
text2.setEffect(effect2);
```

Управлять эффектом позволяют следующие свойства:

- **level** — интенсивность свечения. В качестве значения указывается вещественное число от 0.0 до 1.0. Значение по умолчанию: 0.3. Методы доступа:

```
public final void setLevel(double value)
public final double getLevel()
public final DoubleProperty levelProperty()
```

- `input` — задает входные данные для эффекта. Если свойство имеет значение `null` (значение по умолчанию), то в качестве входных данных используется изображение узла, к которому применяется эффект. Методы доступа:

```
public final void setInput(Effect value)
public final Effect getInput()
public final ObjectProperty<Effect> inputProperty()
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH08_02_Effect`, приведенном в каталоге `CH08_effects` сопровождающего книгу электронного архива (см. *приложение*).

## 8.2.11. Класс *PerspectiveTransform*: трансформация перспективы

Класс `PerspectiveTransform` реализует эффект трансформации перспективы. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.effect.PerspectiveTransform;
```

Иерархия наследования:

`Object` — `Effect` — `PerspectiveTransform`

Конструкторы класса:

```
PerspectiveTransform()
PerspectiveTransform(double ulx, double uly, double urx, double ury,
 double lrx, double lry, double llx, double lly)
```

Во втором конструкторе вначале указываются координаты левого верхнего угла четырехугольника, а затем все остальные координаты углов по часовой стрелке. Пример:

```
PerspectiveTransform effect2 = new PerspectiveTransform(
 50.0, 10.0, 200.0, 10.0, 250.0, 100.0, 0.0, 100.0);
Text text2 = new Text("JavaFX");
text2.setFill(Color.BLACK);
text2.setFont(Font.font("Verdana", FontWeight.BOLD, 40.0));
text2.setEffect(effect2);
```

Управлять эффектом позволяют следующие свойства:

- `ulx` — координата левого верхнего угла по оси `x`. Значение по умолчанию: `0.0`. Методы доступа:

```
public final void setUlx(double value)
public final double getUlx()
public final DoubleProperty ulxProperty()
```

- `uly` — координата левого верхнего угла по оси `y`. Значение по умолчанию: `0.0`. Методы доступа:

```
public final void setUly(double value)
public final double getUly()
public final DoubleProperty ulyProperty()
```

- ❑ **llx** — координата левого нижнего угла по оси *x*. Значение по умолчанию: 0.0.  
**Методы доступа:**

```
public final void setLlx(double value)
public final double getLlx()
public final DoubleProperty llxProperty()
```

- ❑ **lly** — координата левого нижнего угла по оси *y*. Значение по умолчанию: 0.0.  
**Методы доступа:**

```
public final void setLly(double value)
public final double getLly()
public final DoubleProperty llyProperty()
```

- ❑ **urx** — координата правого верхнего угла по оси *x*. Значение по умолчанию: 0.0.  
**Методы доступа:**

```
public final void setUrx(double value)
public final double getUrx()
public final DoubleProperty urxProperty()
```

- ❑ **ury** — координата правого верхнего угла по оси *y*. Значение по умолчанию: 0.0.  
**Методы доступа:**

```
public final void setUry(double value)
public final double getUry()
public final DoubleProperty uryProperty()
```

- ❑ **lrx** — координата правого нижнего угла по оси *x*. Значение по умолчанию: 0.0.  
**Методы доступа:**

```
public final void setLrx(double value)
public final double getLrx()
public final DoubleProperty lrxProperty()
```

- ❑ **lry** — координата правого нижнего угла по оси *y*. Значение по умолчанию: 0.0.  
**Методы доступа:**

```
public final void setLry(double value)
public final double getLry()
public final DoubleProperty lryProperty()
```

- ❑ **input** — задает входные данные для эффекта. Если свойство имеет значение `null` (значение по умолчанию), то в качестве входных данных используется изображение узла, к которому применяется эффект. **Методы доступа:**

```
public final void setInput(Effect value)
public final Effect getInput()
public final ObjectProperty<Effect> inputProperty()
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH08_02_Effect`, приведенном в каталоге `CH08_effects` сопровождающего книгу электронного архива (см. *приложение*).

## 8.2.12. Класс *ColorAdjust*: изменение цветового тона, насыщенности, яркости и контраста

Класс `ColorAdjust` реализует возможность изменения цветового тона, насыщенности, яркости и контраста. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.effect.ColorAdjust;
```

Иерархия наследования:

`Object` – `Effect` – `ColorAdjust`

Конструкторы класса:

```
ColorAdjust()
ColorAdjust(double hue, double saturation, double brightness,
 double contrast)
```

Изменим цвет и яркость текста:

```
ColorAdjust effect2 = new ColorAdjust(-0.5, 0.0, -0.5, 0.0);
Text text = new Text("JavaFX");
text.setFill(Color.BLUE);
text.setFont(Font.font("Verdana", FontWeight.BOLD, 40.0));
text.setEffect(effect2);
```

Управлять эффектом позволяют следующие свойства:

- **hue** — регулировка цветового тона. В качестве значения указывается вещественное число от  $-1.0$  до  $1.0$ . Значение по умолчанию:  $0.0$ . Методы доступа:

```
public final void setHue(double value)
public final double getHue()
public final DoubleProperty hueProperty()
```

- **saturation** — регулировка насыщенности. В качестве значения указывается вещественное число от  $-1.0$  до  $1.0$ . Значение по умолчанию:  $0.0$ . Методы доступа:

```
public final void setSaturation(double value)
public final double getSaturation()
public final DoubleProperty saturationProperty()
```

- **brightness** — регулировка яркости. В качестве значения указывается вещественное число от  $-1.0$  до  $1.0$ . Значение по умолчанию:  $0.0$ . Методы доступа:

```
public final void setBrightness(double value)
public final double getBrightness()
public final DoubleProperty brightnessProperty()
```

- **contrast** — регулировка контраста. В качестве значения указывается вещественное число от  $-1.0$  до  $1.0$ . Значение по умолчанию:  $0.0$ . Методы доступа:

```
public final void setContrast(double value)
public final double getContrast()
public final DoubleProperty contrastProperty()
```

- **input** — задает входные данные для эффекта. Если свойство имеет значение `null` (значение по умолчанию), то в качестве входных данных используется изображение узла, к которому применяется эффект. Методы доступа:

```
public final void setInput(Effect value)
public final Effect getInput()
public final ObjectProperty<Effect> inputProperty()
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH08_02_Effect`, приведенном в каталоге `CH08_effects` сопровождающего книгу электронного архива (см. приложение).

## 8.2.13. Класс *SepiaTone*: эффект состаривания изображения

Класс `SepiaTone` реализует эффект состаривания изображения. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.effect.SepiaTone;
```

Иерархия наследования:

`Object` — `Effect` — `SepiaTone`

Конструкторы класса:

```
SepiaTone()
SepiaTone(double level)
```

Пример:

```
// Image image = ...;
SepiaTone effect = new SepiaTone();
ImageView iv = new ImageView(image);
iv.setFitWidth(400.0);
iv.setPreserveRatio(true);
iv.setEffect(effect);
```

Управлять эффектом позволяют следующие свойства:

- **level** — задает интенсивность эффекта. В качестве значения указывается вещественное число от  $0.0$  до  $1.0$ . Значение по умолчанию:  $1.0$ . Методы доступа:

```
public final void setLevel(double value)
public final double getLevel()
public final DoubleProperty levelProperty()
```

- `input` — задает входные данные для эффекта. Если свойство имеет значение `null` (значение по умолчанию), то в качестве входных данных используется изображение узла, к которому применяется эффект. Методы доступа:

```
public final void setInput(Effect value)
public final Effect getInput()
public final ObjectProperty<Effect> inputProperty()
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH08_02_Effect`, приведенном в каталоге `CH08_effects` сопровождающего книгу электронного архива (см. *приложение*).

## 8.2.14. Класс *Lighting*: эффект освещения источником света

Класс `Lighting` реализует эффект освещения источником света. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.effect.Lighting;
```

Иерархия наследования:

```
Object — Effect — Lighting
```

Конструкторы класса:

```
Lighting()
Lighting(Light light)
```

Пример:

```
// import javafx.scene.effect.Light;
Lighting effect = new Lighting();
Light.Distant light = new Light.Distant();
effect.setLight(light);
Text text = new Text("JavaFX");
text.setFill(Color.WHITE);
text.setFont(Font.font("Verdana", FontWeight.BOLD, 40.0));
text.setEffect(effect);
```

Управлять эффектом позволяют следующие свойства:

- `light` — задает источник освещения (объект класса `Light`). По умолчанию используется объект класса `Light.Distant`. Методы доступа:

```
import javafx.scene.effect.Light;
public final void setLight(Light value)
public final Light getLight()
public final ObjectProperty<Light> lightProperty()
```

- `diffuseConstant` — рассеивание света. В качестве значения указывается вещественное число от 0.0 до 2.0. Значение по умолчанию: 1.0. Методы доступа:

```
public final void setDiffuseConstant(double value)
public final double getDiffuseConstant()
public final DoubleProperty diffuseConstantProperty()
```

- ❑ **specularConstant** — отражение света в центре. В качестве значения указывается вещественное число от 0.0 до 2.0. Значение по умолчанию: 0.3. Методы доступа:

```
public final void setSpecularConstant(double value)
public final double getSpecularConstant()
public final DoubleProperty specularConstantProperty()
```

- ❑ **specularExponent** — отражение света по краям. В качестве значения указывается вещественное число от 0.0 до 40.0. Значение по умолчанию: 20.0. Методы доступа:

```
public final void setSpecularExponent(double value)
public final double getSpecularExponent()
public final DoubleProperty specularExponentProperty()
```

- ❑ **surfaceScale** — коэффициент масштаба поверхности. В качестве значения указывается вещественное число от 0.0 до 10.0. Значение по умолчанию: 1.5. Методы доступа:

```
public final void setSurfaceScale(double value)
public final double getSurfaceScale()
public final DoubleProperty surfaceScaleProperty()
```

- ❑ **bumpInput** — задает входные данные для создания рельефа поверхности. По умолчанию используется эффект *Shadow* с радиусом размытия тени 10.0. Методы доступа:

```
public final void setBumpInput(Effect value)
public final Effect getBumpInput()
public final ObjectProperty<Effect> bumpInputProperty()
```

- ❑ **contentInput** — задает входные данные для эффекта. Если свойство имеет значение *null* (значение по умолчанию), то в качестве входных данных используется изображение узла, к которому применяется эффект. Методы доступа:

```
public final void setContentInput(Effect value)
public final Effect getContentInput()
public final ObjectProperty<Effect> contentInputProperty()
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH08_02_Effect`, приведенном в каталоге `CH08_effects` сопровождающего книгу электронного архива (см. приложение).

## 8.2.15. Класс *Light*: источник освещения

Источник освещения в эффекте *Lighting* задается с помощью объекта класса *Light*, а точнее сказать, в виде объекта статических классов: *Light.Distant*, *Light.Point* и *Light.Spot*. Инструкция импорта класса *Light*:



```
// Модуль javafx.graphics
import javafx.scene.effect.Light;
```

**Класс `Light` содержит свойство `color`, которое задает цвет источника света. По умолчанию используется белый цвет. Методы доступа:**

```
public final void setColor(Color value)
public final Color getColor()
public final ObjectProperty<Color> colorProperty()
```

**Класс `Light.Distant` описывает равномерно светящийся удаленный источник света (эффект солнца). В эффекте `Lighting` этот источник используется по умолчанию. Иерархия наследования:**

`Object` — `Light` — `Light.Distant`

**Конструкторы класса `Light.Distant`:**

```
Distant()
Distant(double azimuth, double elevation, Color color)
```

**Пример:**

```
Light.Distant light2 = new Light.Distant(45.0, 45.0, Color.BISQUE);
Lighting effect2 = new Lighting(light2);
Text text2 = new Text("JavaFX");
text2.setFill(Color.WHITE);
text2.setFont(Font.font("Verdana", FontWeight.BOLD, 40.0));
text2.setEffect(effect2);
```

**Управлять источником света позволяют следующие свойства из класса `Light.Distant`:**

❑ **`azimuth` — задает азимут (угол направления в градусах в плоскости `XY`). Значение по умолчанию: 45.0. Методы доступа:**

```
public final void setAzimuth(double value)
public final double getAzimuth()
public final DoubleProperty azimuthProperty()
```

❑ **`elevation` — задает высоту (угол направления в градусах в плоскости `YZ`). Значение по умолчанию: 45.0. Методы доступа:**

```
public final void setElevation(double value)
public final double getElevation()
public final DoubleProperty elevationProperty()
```

**Класс `Light.Point` описывает точечный источник света (эффект фонарика, светящего сверху вниз под прямым углом). Иерархия наследования:**

`Object` — `Light` — `Light.Point`

**Конструкторы класса `Light.Point`:**

```
Point()
Point(double x, double y, double z, Color color)
```

**Пример:**

```
Light.Point light2 = new Light.Point(100.0, 50.0, 100.0, Color.WHITE);
Lighting effect2 = new Lighting(light2);
Text text2 = new Text("JavaFX");
text2.setFill(Color.BLUE);
text2.setFont(Font.font("Verdana", FontWeight.BOLD, 80.0));
text2.setEffect(effect2);
```

Управлять источником света позволяют следующие свойства из класса `Light.Point`:

- ☐ **x** — положение источника света по оси **x**. Значение по умолчанию: 0.0. Методы доступа:

```
public final void setX(double value)
public final double getX()
public final DoubleProperty xProperty()
```

- ☐ **y** — положение источника света по оси **y**. Значение по умолчанию: 0.0. Методы доступа:

```
public final void setY(double value)
public final double getY()
public final DoubleProperty yProperty()
```

- ☐ **z** — положение источника света по оси **z**. Значение по умолчанию: 0.0. Методы доступа:

```
public final void setZ(double value)
public final double getZ()
public final DoubleProperty zProperty()
```

Класс `Light.Spot` расширяет класс `Light.Point` и описывает точечный источник света (эффект фонарика, светящего под разными углами). Иерархия наследования:

Object — Light — Light.Point — Light.Spot

**Конструкторы класса `Light.Spot`:**

```
Spot()
Spot(double x, double y, double z, double specularExponent, Color color)
```

**Пример:**

```
Light.Spot light = new Light.Spot();
light.setX(100.0);
light.setY(50.0);
light.setZ(100.0);
light.setColor(Color.BISQUE);
light.setPointsAtX(100.0);
light.setPointsAtY(50.0);
light.setPointsAtZ(0.0);
light.setSpecularExponent(1.0);
Lighting effect = new Lighting(light);
Text text = new Text("JavaFX");
```

```
text.setFill(Color.RED);
text.setFont(Font.font("Verdana", FontWeight.BOLD, 80.0));
text.setEffect(effect);
```

Управлять источником света позволяют следующие свойства из класса `Light.Spot`:

- ❑ `pointsAtX` — координата вектора направления источника света по оси *x*. Значение по умолчанию: 0.0. Методы доступа:

```
public final void setPointsAtX(double value)
public final double getPointsAtX()
public final DoubleProperty pointsAtXProperty()
```

- ❑ `pointsAtY` — координата вектора направления источника света по оси *y*. Значение по умолчанию: 0.0. Методы доступа:

```
public final void setPointsAtY(double value)
public final double getPointsAtY()
public final DoubleProperty pointsAtYProperty()
```

- ❑ `pointsAtZ` — координата вектора направления источника света по оси *z*. Значение по умолчанию: 0.0. Методы доступа:

```
public final void setPointsAtZ(double value)
public final double getPointsAtZ()
public final DoubleProperty pointsAtZProperty()
```

- ❑ `specularExponent` — управляет фокусом. В качестве значения указывается вещественное число от 0.0 до 4.0. Значение по умолчанию: 1.0. Методы доступа:

```
public final void setSpecularExponent(double value)
public final double getSpecularExponent()
public final DoubleProperty specularExponentProperty()
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH08_02_Effect`, приведенном в каталоге `CH08_effects` сопровождающего книгу электронного архива (см. приложение).

## 8.2.16. Класс *Blend*: смешивание

Класс `Blend` позволяет получить результат смешивания верхнего ввода с нижним, используя режимы наложения, определенные в перечислении `BlendMode` (см. разд. 5.6). Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.effect.Blend;
```

Иерархия наследования:

Object — Effect — Blend

Конструкторы класса:

```
Blend()
Blend(BlendMode mode)
Blend(BlendMode mode, Effect bottomInput, Effect topInput)
```

Пример наложения цвета в режиме `BlendMode.OVERLAY` на прямоугольник:

```
ColorInput colorInput = new ColorInput();
colorInput.setPaint(Color.WHITE);
colorInput.setX(50.0);
colorInput.setY(50.0);
colorInput.setWidth(70.0);
colorInput.setHeight(70.0);

Blend effect = new Blend();
effect.setMode(BlendMode.OVERLAY);
effect.setTopInput(colorInput);

Rectangle rect = new Rectangle(100.0, 100.0);
rect.setFill(Color.GREEN);
rect.setEffect(effect);
```

Управлять эффектом позволяют следующие свойства:

- `mode` — задает режим смешивания верхнего ввода с нижним. В качестве значения указываются константы из перечисления `BlendMode` (см. *разд. 5.6*). Значение по умолчанию: `BlendMode.SRC_OVER`. Методы доступа:

```
import javafx.scene.effect.BlendMode;
public final void setMode(BlendMode value)
public final BlendMode getMode()
public final ObjectProperty<BlendMode> modeProperty()
```

- `topInput` — верхний ввод. Если свойство имеет значение `null` (значение по умолчанию), то в качестве входных данных используется изображение узла, к которому применяется эффект. Методы доступа:

```
public final void setTopInput(Effekt value)
public final Effect getTopInput()
public final ObjectProperty<Effect> topInputProperty()
```

- `bottomInput` — нижний ввод. Если свойство имеет значение `null` (значение по умолчанию), то в качестве входных данных используется изображение узла, к которому применяется эффект. Методы доступа:

```
public final void setBottomInput(Effekt value)
public final Effect getBottomInput()
public final ObjectProperty<Effect> bottomInputProperty()
```

- `opacity` — задает степень непрозрачности верхнего ввода. В качестве значения указывается вещественное число от 0.0 (полностью прозрачный) до 1.0 (полностью непрозрачный). Значение по умолчанию: 1.0. Методы доступа:

```
public final void setOpacity(double value)
public final double getOpacity()
public final DoubleProperty opacityProperty()
```

## ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH08_02_Effect`, приведенном в каталоге `CH08_effects` сопровождающего книгу электронного архива (см. приложение).

### 8.2.17. Класс *ColorInput*

Класс `ColorInput` позволяет использовать прямоугольную область с заливкой в качестве входных данных для эффекта. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.effect.ColorInput;
```

Иерархия наследования:

`Object` — `Effect` — `ColorInput`

Конструкторы класса:

```
ColorInput()
ColorInput(double x, double y, double width, double height, Paint paint)
```

Пример:

```
ColorInput colorInput = new ColorInput(50.0, 50.0,
 70.0, 70.0, Color.WHITE);

Blend effect = new Blend();
effect.setMode(BlendMode.OVERLAY);
effect.setTopInput(colorInput);
```

Управлять эффектом позволяют следующие свойства:

- `paint` — задает характеристики заливки. В качестве значения можно указать объекты классов: `Color` (сплошной цвет), `LinearGradient` (линейный градиент), `RadialGradient` (радиальный градиент) и `ImagePattern` (фоновое изображение). Значение по умолчанию: `Color.RED`. Методы доступа:

```
public final void setPaint(Paint value)
public final Paint getPaint()
public final ObjectProperty<Paint> paintProperty()
```

- `x` — координата левого верхнего угла прямоугольной области по оси `x`. Значение по умолчанию: `0.0`. Методы доступа:

```
public final void setX(double value)
public final double getX()
public final DoubleProperty xProperty()
```

- `y` — координата левого верхнего угла прямоугольной области по оси `y`. Значение по умолчанию: `0.0`. Методы доступа:

```
public final void setY(double value)
public final double getY()
public final DoubleProperty yProperty()
```

- `width` — ширина прямоугольной области. Значение по умолчанию: `0.0`. Методы доступа:

```
public final void setWidth(double value)
public final double getWidth()
public final DoubleProperty widthProperty()
```

- ❑ **height** — высота прямоугольной области. Значение по умолчанию: 0.0. Методы доступа:

```
public final void setHeight(double value)
public final double getHeight()
public final DoubleProperty heightProperty()
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH08_02_Effect`, приведенном в каталоге `CH08_effects` сопровождающего книгу электронного архива (см. приложение).

## 8.2.18. Класс *ImageInput*

Класс `ImageInput` позволяет использовать изображение в качестве входных данных для эффекта. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.effect.ImageInput;
```

Иерархия наследования:

Object — Effect — ImageInput

Конструкторы класса:

```
ImageInput()
ImageInput(Image source)
ImageInput(Image source, double x, double y)
```

Пример:

```
WritableImage wim = new WritableImage(70, 70);
PixelWriter pw = wim.getPixelWriter();
for (int x = 0, w = (int) wim.getWidth(); x < w; x++) {
 for (int y = 0, h = (int) wim.getHeight(); y < h; y++) {
 pw.setColor(x, y, Color.WHITE);
 }
}
ImageInput imageInput = new ImageInput(wim, 50.0, 50.0);
Blend effect3 = new Blend(BlendMode.DIFFERENCE, null, imageInput);
Rectangle rect3 = new Rectangle(100.0, 100.0);
rect3.setFill(Color.BLUE);
rect3.setEffect(effect3);
```

Управлять эффектом позволяют следующие свойства:

- ❑ **source** — задает изображение. Методы доступа:

```
public final void setSource(Image value)
public final Image getSource()
public final ObjectProperty<Image> sourceProperty()
```

- **x** — координата левого верхнего угла изображения по оси **x**. Значение по умолчанию: 0.0. Методы доступа:

```
public final void setX(double value)
public final double getX()
public final DoubleProperty xProperty()
```

- **y** — координата левого верхнего угла изображения по оси **y**. Значение по умолчанию: 0.0. Методы доступа:

```
public final void setY(double value)
public final double getY()
public final DoubleProperty yProperty()
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH08_02_Effect`, приведенном в каталоге `CH08_effects` сопровождающего книгу электронного архива (см. приложение).

## 8.2.19. Класс *DisplacementMap*: искажение

Класс `DisplacementMap` позволяет выполнить искажение на основе объекта класса `FloatMap`. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.effect.DisplacementMap;
```

Иерархия наследования:

Object — Effect — DisplacementMap

Конструкторы класса:

```
DisplacementMap()
DisplacementMap(FloatMap mapData)
DisplacementMap(FloatMap mapData, double offsetX, double offsetY,
 double scaleX, double scaleY)
```

Пример:

```
FloatMap floatMap = new FloatMap(150, 100);
for (int x = 0, w = floatMap.getWidth(); x < w; x++) {
 double value = Math.sin(x / 30.0 * Math.PI) / 10.0;
 for (int y = 0, h = floatMap.getHeight(); y < h; y++) {
 floatMap.setSamples(x, y, 0.0f, (float)value);
 }
}
DisplacementMap effect = new DisplacementMap();
effect.setMapData(floatMap);
Text text = new Text("JavaFX");
text.setFill(Color.BLUE);
text.setFont(Font.font("Verdana", FontWeight.BOLD, 60.0));
text.setEffect(effect);
```

Управлять эффектом позволяют следующие свойства:

- ❑ **mapData** — задает карту смещения пикселей с помощью объекта класса `FloatMap`.  
Методы доступа:  

```
import javafx.scene.effect.FloatMap;
public final void setMapData(FloatMap value)
public final FloatMap getMapData()
public final ObjectProperty<FloatMap> mapDataProperty()
```
- ❑ **offsetX** — смещение по оси X. Значение по умолчанию: 0.0. Методы доступа:  

```
public final void setOffsetX(double value)
public final double getOffsetX()
public final DoubleProperty offsetXProperty()
```
- ❑ **offsetY** — смещение по оси Y. Значение по умолчанию: 0.0. Методы доступа:  

```
public final void setOffsetY(double value)
public final double getOffsetY()
public final DoubleProperty offsetYProperty()
```
- ❑ **scaleX** — масштаб по оси X. Значение по умолчанию: 1.0. Методы доступа:  

```
public final void setScaleX(double value)
public final double getScaleX()
public final DoubleProperty scaleXProperty()
```
- ❑ **scaleY** — масштаб по оси Y. Значение по умолчанию: 1.0. Методы доступа:  

```
public final void setScaleY(double value)
public final double getScaleY()
public final DoubleProperty scaleYProperty()
```
- ❑ **wrap** — если свойство имеет значение `true`, то смещенные пиксели будут добавляться в конец или в начало. Значение по умолчанию: `false`. Методы доступа:  

```
public final void setWrap(boolean value)
public final boolean isWrap()
public final BooleanProperty wrapProperty()
```
- ❑ **input** — задает входные данные для эффекта. Если свойство имеет значение `null` (значение по умолчанию), то в качестве входных данных используется изображение узла, к которому применяется эффект. Методы доступа:  

```
public final void setInput(Effect value)
public final Effect getInput()
public final ObjectProperty<Effect> inputProperty()
```

Карта смещения задается с помощью класса `FloatMap`. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.effect.FloatMap;
```

Конструкторы класса `FloatMap`:

```
FloatMap()
FloatMap(int width, int height)
```



Управлять объектом позволяют следующие свойства из класса `FloatMap`:

- ❑ `width` — задает ширину карты смещения в пикселах (целое число от 1 до 4096).  
Значение по умолчанию: 1. Методы доступа:

```
public final void setWidth(int value)
public final int getWidth()
public final IntegerProperty widthProperty()
```

- ❑ `height` — задает высоту карты смещения в пикселах (целое число от 1 до 4096).  
Значение по умолчанию: 1. Методы доступа:

```
public final void setHeight(int value)
public final int getHeight()
public final IntegerProperty heightProperty()
```

Добавление данных в карту смещения выполняется с помощью метода `setSamples()`.  
Форматы метода:

```
public void setSamples(int x, int y, float s0)
public void setSamples(int x, int y, float s0, float s1)
public void setSamples(int x, int y, float s0, float s1, float s2)
public void setSamples(int x, int y, float s0, float s1, float s2,
 float s3)
```

Для нужд эффекта `DisplacementMap` используется второй формат, где параметр `s0` задает коэффициент для оси `X`, а параметр `s1` — коэффициент для оси `Y`. Пример:

```
FloatMap floatMap = new FloatMap(150, 100);
for (int x = 0, w = floatMap.getWidth(); x < w; x++) {
 double value = Math.sin(x / 30.0 * Math.PI) / 10.0;
 for (int y = 0, h = floatMap.getHeight(); y < h; y++) {
 floatMap.setSamples(x, y, 0.0f, (float)value);
 }
}
```

Можно также воспользоваться методом `setSample()`. Формат метода:

```
public void setSample(int x, int y, int band, float s)
```

В этом методе параметр `band` задает номер группы (число от 0 до 3), а параметр `s` — значение для указанной группы. Перепишем предыдущую инструкцию с методом `setSamples()`, используя метод `setSample()`:

```
floatMap.setSample(x, y, 0, 0.0f);
floatMap.setSample(x, y, 1, (float)value);
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH08_02_Effect`, приведенном в каталоге `CH08_effects` сопровождающего книгу электронного архива (см. приложение).

## ГЛАВА 9



# Создание анимации

Все рассмотренные в предыдущей главе трансформации применялись сразу. Однако часто бывает нужно сделать так, чтобы трансформации применялись постепенно. Например, при наведении указателя мыши на пункт меню надо плавно увеличить значок и изменить его прозрачность, а при выведении указателя также плавно вернуть настройки в первоначальное положение. В этой главе мы как раз и научимся это делать, а также рассмотрим создание произвольной анимации.

### 9.1. Класс *Duration*: продолжительность выполнения анимации

Продолжительность выполнения анимации задается с помощью класса `Duration`. Инструкция импорта:

```
// Модуль javafx.base
import javafx.util.Duration;
```

Конструктор класса:

```
Duration(double millis)
```

Параметр `millis` задает продолжительность в миллисекундах. Пример:

```
Duration d1 = new Duration(500.0);
System.out.println(d1); // 500.0 ms
```

Создать экземпляры класса позволяют также следующие статические методы:

❑ `millis()` — задает продолжительность в миллисекундах. Формат метода:

```
public static Duration millis(double ms)
```

Пример:

```
Duration d2 = Duration.millis(500.0);
System.out.println(d2); // 500.0 ms
```

❑ `seconds()` — задает продолжительность в секундах. Формат метода:

```
public static Duration seconds(double s)
```

- ❑ `minutes()` — задает продолжительность в минутах. Формат метода:

```
public static Duration minutes(double m)
```

- ❑ `hours()` — задает продолжительность в часах. Формат метода:

```
public static Duration hours(double h)
```

- ❑ `valueOf()` — задает продолжительность на основе строки в формате число[ms|s|m|h]. Формат метода:

```
public static Duration valueOf(String time)
```

**Пример:**

```
System.out.println(Duration.valueOf("500ms")); // 500.0 ms
```

В классе `Duration` определены следующие статические константы:

- ❑ `ZERO` — нулевая продолжительность. Формат:

```
public static final Duration ZERO
```

**Пример:**

```
System.out.println(Duration.ZERO); // 0.0 ms
```

- ❑ `ONE` — продолжительность в одну миллисекунду. Формат:

```
public static final Duration ONE
```

**Пример:**

```
System.out.println(Duration.ONE); // 1.0 ms
```

- ❑ `INDEFINITE` — бесконечная продолжительность. Формат:

```
public static final Duration INDEFINITE
```

**Пример:**

```
System.out.println(Duration.INDEFINITE); // INDEFINITE
```

- ❑ `UNKNOWN` — неизвестная продолжительность. Формат:

```
public static final Duration UNKNOWN
```

**Пример:**

```
System.out.println(Duration.UNKNOWN); // UNKNOWN
```

Рассмотрим основные методы класса `Duration`:

- ❑ `toMillis()` — возвращает текущую продолжительность в миллисекундах. Формат метода:

```
public double toMillis()
```

- ❑ `toSeconds()` — возвращает текущую продолжительность в секундах. Формат метода:

```
public double toSeconds()
```

- ❑ `toMinutes()` — возвращает текущую продолжительность в минутах. Формат метода:

```
public double toMinutes()
```

- ❑ `toHours()` — возвращает текущую продолжительность в часах. Формат метода:

```
public double toHours()
```

**Пример:**

```
Duration d = new Duration(500.0);
System.out.println(d.toMillis()); // 500.0
System.out.println(d.toSeconds()); // 0.5
System.out.println(d.toMinutes()); // 0.008333333333333333
System.out.println(d.toHours()); // 1.388888888888889E-4
System.out.println(d.toString()); // 500.0 ms
```

Если используется бесконечная продолжительность, то методы: `toMillis()`, `toSeconds()`, `toMinutes()` и `toHours()` вернут значение `Infinity`:

```
d = Duration.INDEFINITE;
System.out.println(d.toMillis()); // Infinity
```

Если продолжительность неизвестна, то методы: `toMillis()`, `toSeconds()`, `toMinutes()` и `toHours()` вернут значение `NaN`:

```
d = Duration.UNKNOWN;
System.out.println(d.toMillis()); // NaN
```

- ❑ `add()` — прибавляет указанный промежуток времени. Формат метода:

```
public Duration add(Duration other)
```

- ❑ `subtract()` — вычитает указанный промежуток времени. Формат метода:

```
public Duration subtract(Duration other)
```

- ❑ `multiply()` — умножает текущую продолжительность на указанное число. Формат метода:

```
public Duration multiply(double n)
```

- ❑ `divide()` — делит текущую продолжительность на указанное число. Формат метода:

```
public Duration divide(double n)
```

**Пример:**

```
d = new Duration(500.0).divide(5.0);
System.out.println(d.toMillis()); // 100.0
```

- ❑ `negate()` — умножает текущую продолжительность на  $-1$ . Формат метода:

```
public Duration negate()
```

- ❑ `isIndefinite()` — возвращает значение `true`, если текущая продолжительность бесконечна. Формат метода:

```
public boolean isIndefinite()
```

- ❑ `isUnknown()` — возвращает значение `true`, если текущая продолжительность неизвестна. Формат метода:

```
public boolean isUnknown()
```

- ❑ `lessThan()`, `lessThanOrEqualTo()`, `greaterThan()`, `greaterThanOrEqualTo()`, `compareTo()` и `equals()` — позволяют сравнить два объекта. Форматы методов:

```
public boolean lessThan(Duration other) // <
public boolean lessThanOrEqualTo(Duration other) // <=
public boolean greaterThan(Duration other) // >
public boolean greaterThanOrEqualTo(Duration other) // >=
public int compareTo(Duration d)
public boolean equals(Object obj) // ==
```

## 9.2. Класс *Animation*

Абстрактный класс `Animation` является базовым классом для всех классов, реализующих в JavaFX различные анимации. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.animation.Animation;
```

### 9.2.1. Настройка параметров

Управлять настройками анимации позволяют следующие свойства из класса `Animation`:

- ❑ `delay` — задает задержку перед началом анимации. Значение по умолчанию: `0.0`. Методы доступа:

```
public final void setDelay(Duration value)
public final Duration getDelay()
public final ObjectProperty<Duration> delayProperty()
```

- ❑ `cycleCount` — задает количество повторений анимации. Значение по умолчанию: `1.0`. Методы доступа:

```
public final void setCycleCount(int value)
public final int getCycleCount()
public final IntegerProperty cycleCountProperty()
```

Если нужно, чтобы анимация выполнялась без ограничений по времени, то в качестве значения следует указать константу `INDEFINITE`:

```
System.out.println(Animation.INDEFINITE); // -1
```

- ❑ `autoReverse` — если свойство имеет значение `true`, то повтор анимации будет выполняться в противоположном направлении. Значение по умолчанию: `false`. Методы доступа:

```
public final void setAutoReverse(boolean value)
public final boolean isAutoReverse()
public final BooleanProperty autoReverseProperty()
```

- ❑ **rate** — задает скорость и направление анимации. Значение по умолчанию: 1.0. Отрицательное значение приведет к изменению направления на противоположное, а значение 2.0 — увеличит скорость в два раза. Если свойство имеет значение 0.0, то анимация будет остановлена. Методы доступа:

```
public final void setRate(double value)
public final double getRate()
public final DoubleProperty rateProperty()
```

- ❑ **currentRate** — содержит текущую скорость и направление анимации. Методы доступа:

```
public final double getCurrentRate()
public final ReadOnlyDoubleProperty currentRateProperty()
```

- ❑ **cycleDuration** — содержит продолжительность выполнения одного цикла анимации (при значении свойства **rate**, равном 1.0). Методы доступа:

```
public final Duration getCycleDuration()
public final ReadOnlyObjectProperty<Duration>
 cycleDurationProperty()
```

- ❑ **totalDuration** — содержит общую продолжительность анимации (**cycleDuration** \* **cycleCount**). Если анимация выполняется бесконечно, то свойство содержит значение **Duration.INDEFINITE**. Методы доступа:

```
public final Duration getTotalDuration()
public final ReadOnlyObjectProperty<Duration>
 totalDurationProperty()
```

- ❑ **currentTime** — содержит текущее время выполнения анимации (позицию на шкале времени). Методы доступа:

```
public final Duration getCurrentTime()
public final ReadOnlyObjectProperty<Duration>
 currentTimeProperty()
```

- ❑ **status** — содержит текущий статус анимации. Методы доступа:

```
public final Animation.Status getStatus()
public final ReadOnlyObjectProperty<Animation.Status>
 statusProperty()
```

Может содержать следующие константы из перечисления **Animation.Status**:

- **RUNNING** — анимация выполняется;
- **PAUSED** — выполнение анимации приостановлено;
- **STOPPED** — выполнение анимации завершено;

- ❑ `onFinished` — позволяет назначить обработчик события завершения выполнения анимации. Методы доступа:

```
public final void setOnFinished(
 EventHandler<ActionEvent> value)
public final EventHandler<ActionEvent> getOnFinished()
public final ObjectProperty<EventHandler<ActionEvent>>
 onFinishedProperty()
```

Создадим анимацию изменения прозрачности прямоугольника. Анимация начнет выполняться после задержки в одну секунду и повторится шесть раз в разных направлениях. После завершения выполнения анимации выведем сообщение в окно консоли (листинг 9.1).

#### Листинг 9.1. Анимация изменения прозрачности

```
Rectangle rect3 = new Rectangle(100.0, 100.0);
rect3.setFill(Color.RED);
rect3.relocate(350.0, 50.0);

FadeTransition ft2 = new FadeTransition(
 Duration.seconds(1.0), rect3);
ft2.setDelay(Duration.seconds(1.0)); // Задержка
ft2.setAutoReverse(true); // Изменение направления
ft2.setFromValue(1.0); // Начальное значение
ft2.setToValue(0.0); // Конечное значение
ft2.setCycleCount(6); // Количество повторов
ft2.setOnFinished(event -> {
 System.out.println("Анимация завершена");
});
ft2.play(); // Запуск анимации
```

Получить максимальную частоту смены кадров анимации позволяет метод `getTargetFramerate()` из класса `Animation`. Формат метода:

```
public final double getTargetFramerate()
```

Выведем эту частоту:

```
System.out.println(ft2.getTargetFramerate()); // 60.0
```

#### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH09_02_Animation`, приведенном в каталоге `CH09_animations` сопровождающего книгу электронного архива (см. приложение).

## 9.2.2. Использование меток

Шкала времени может содержать именованные метки, позволяющие перейти в позицию, на которую указывает метка. По умолчанию создаются две метки: `start`

(соответствует началу анимации) и `end` (соответствует концу анимации). Метод `getCuePoints()` позволяет добавить пользовательскую метку в словарь. Формат метода:

```
public final ObservableMap<String, Duration> getCuePoints()
```

Обратите внимание: метки `start` и `end` не содержатся в этом словаре. Пример добавления метки:

```
ft2.getCuePoints().put("p1", Duration.millis(100.0));
```

Перейти к указанной метке позволяет метод `jumpTo()`. Форматы метода:

```
public void jumpTo(String cuePoint)
```

```
public void jumpTo(Duration time)
```

Пример перехода к метке с названием `p1`:

```
ft2.jumpTo("p1");
```

### 9.2.3. Запуск и остановка анимации

Управлять циклом анимации позволяют следующие методы из класса `Animation`:

☐ `play()` — запускает анимацию с текущей позиции. Формат метода:

```
public void play()
```

☐ `playFromStart()` — запускает анимацию с начала. Формат метода:

```
public void playFromStart()
```

☐ `playFrom()` — запускает анимацию, начиная с указанной именованной метки или времени. Форматы метода:

```
public void playFrom(String cuePoint)
```

```
public void playFrom(Duration time)
```

☐ `pause()` — приостанавливает выполнение анимации. Формат метода:

```
public void pause()
```

☐ `stop()` — останавливает анимацию. Важно останавливать анимацию явным образом (особенно бесконечную), например, при закрытии окна, иначе она так и останется работать, пока не будут закрыты все окна приложения. Формат метода:

```
public void stop()
```

Запустим анимацию или временно приостановим выполнение анимации с помощью щелчка мыши на прямоугольнике:

```
rect3.addEventHandler(MouseEvent.MOUSE_CLICKED, event -> {
 if (ft2.getStatus() == Animation.Status.RUNNING) {
 ft2.pause();
 }
})
```



```

else {
 ft2.play();
}
});

```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH09_02_Animation`, приведенном в каталоге `CH09_animations` сопровождающего книгу электронного архива (см. приложение).

## 9.3. Классы *Transition* и *Interpolator*

Абстрактный класс `Transition` является базовым классом для всех классов, реализующих плавные трансформации. Инструкция импорта:

```

// Модуль javafx.graphics
import javafx.animation.Transition;

```

Иерархия наследования:

Object — Animation — Transition

Как видно из иерархии, класс `Transition` является наследником класса `Animation`, а следовательно, наследует и все его методы. Класс, наследующий класс `Transition`, должен переопределить и реализовать абстрактный метод `interpolate()`. Формат метода:

```
protected abstract void interpolate(double frac)
```

Метод `interpolate()` принимает вещественное число от 0.0 до 1.0, которое соответствует текущей позиции на шкале времени. Иными словами, метод `interpolate()` вызывается для каждого кадра анимации. Пример реализации метода:

```

Animation animation = new Transition() {
 {
 setCycleDuration(Duration.millis(2000));
 setInterpolator(Interpolator.EASE_OUT);
 }
 protected void interpolate(double frac) {
 // Здесь меняем свойства какого-либо объекта
 }
};
animation.play();

```

Значение параметра `frac` зависит от значения свойства `interpolator`. Это свойство содержит объект, определяющий способ прироста значения. Например, вначале анимация выполняется медленно, а под конец ускоряется, или наоборот. Значение по умолчанию: `Interpolator.EASE_BOTH`. Методы доступа:

```
public final void setInterpolator(Interpolator value)
public final Interpolator getInterpolator()
public final ObjectProperty<Interpolator> interpolatorProperty()
```

**Инструкция импорта класса Interpolator:**

```
// Модуль javafx.graphics
import javafx.animation.Interpolator;
```

**Класс Interpolator предоставляет следующие готовые объекты:**

- ❑ **DISCRETE** — значение остается начальным до окончания временного интервала, а затем становится конечным. Формат:

```
public static final Interpolator DISCRETE
```

- ❑ **LINEAR** — равномерный ход анимации. Формат:

```
public static final Interpolator LINEAR
```

- ❑ **EASE\_IN** — замедление в начале и ускорение к концу анимации. Формат:

```
public static final Interpolator EASE_IN
```

- ❑ **EASE\_OUT** — ускоренное движение в начале и замедление к концу анимации. Формат:

```
public static final Interpolator EASE_OUT
```

- ❑ **EASE\_BOTH** — замедление в начале, затем равномерное движение и замедление в конце анимации. Формат:

```
public static final Interpolator EASE_BOTH
```

**Пример анимации движения прямоугольника по горизонтали:**

```
// import javafx.animation.TranslateTransition;
Rectangle rect = new Rectangle(50.0, 50.0);
rect.setFill(Color.RED);
rect.relocate(50.0, 50.0);

TranslateTransition tt = new TranslateTransition(
 Duration.seconds(5.0), rect);

tt.setAutoReverse(true);
tt.setFromX(0.0);
tt.setToX(450.0);
tt.setCycleCount(Animation.INDEFINITE);
tt.setInterpolator(Interpolator.EASE_BOTH);
tt.play();
```

**Создать объект позволяют также статические методы SPLINE() и TANGENT(). Форматы методов:**

```
public static Interpolator SPLINE(double x1, double y1,
 double x2, double y2)
public static Interpolator TANGENT(Duration t, double v)
```

```
public static Interpolator TANGENT(Duration t1, double v1,
 Duration t2, double v2)
```

Класс, наследующий абстрактный класс `Interpolator`, должен переопределить и реализовать абстрактный метод `curve()`. Формат метода:

```
protected abstract double curve(double t)
```

Метод `curve()` принимает вещественное число от 0.0 до 1.0, которое соответствует текущей позиции на шкале времени, и должен возвращать измененное (или то же самое) значение. Пример реализации метода:

```
Interpolator interpolator = new Interpolator() {
 @Override
 protected double curve(double t) {
 return Math.pow(t, 2);
 }
};
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH09_02_Animation`, приведенном в каталоге `CH09_animations` сопровождающего книгу электронного архива (см. приложение).

## 9.4. Класс *FadeTransition*: изменение прозрачности

Класс `FadeTransition` реализует плавное изменение прозрачности. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.animation.FadeTransition;
```

Иерархия наследования:

```
Object — Animation — Transition — FadeTransition
```

Конструкторы класса:

```
FadeTransition()
FadeTransition(Duration duration)
FadeTransition(Duration duration, Node node)
```

Управлять анимацией позволяют следующие свойства:

❑ `fromValue` — начальное значение непрозрачности. Значение по умолчанию: `Double.NaN`. Методы доступа:

```
public final void setFromValue(double value)
public final double getFromValue()
public final DoubleProperty fromValueProperty()
```

❑ `toValue` — конечное значение непрозрачности. Значение по умолчанию: `Double.NaN`. Методы доступа:

```
public final void setToValue(double value)
public final double getToValue()
public final DoubleProperty toValueProperty()
```

- **byValue** — задает смещение. Это значение прибавляется к начальному значению, и сумма становится конечным значением. Если задано значение свойства `toValue`, то значение свойства `byValue` игнорируется. Методы доступа:

```
public final void setByValue(double value)
public final double getByValue()
public final DoubleProperty byValueProperty()
```

- **duration** — продолжительность анимации. Значение по умолчанию: 400 ms. Методы доступа:

```
public final void setDuration(Duration value)
public final Duration getDuration()
public final ObjectProperty<Duration> durationProperty()
```

- **node** — узел, к которому применяется анимация. Методы доступа:

```
public final void setNode(Node value)
public final Node getNode()
public final ObjectProperty<Node> nodeProperty()
```

#### Пример:

```
Rectangle rect = new Rectangle(50.0, 50.0);
rect.setFill(Color.RED);
rect.relocate(50.0, 50.0);
```

```
FadeTransition ft = new FadeTransition(
 Duration.seconds(5.0), rect);
ft.setAutoReverse(true);
ft.setFromValue(1.0);
ft.setToValue(0.0);
ft.setCycleCount(Animation.INDEFINITE);
ft.setInterpolator(Interpolator.LINEAR);
ft.play();
```

#### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH09_02_Animation`, приведенном в каталоге `CH09_animations` сопровождающего книгу электронного архива (см. приложение).

## 9.5. Класс *TranslateTransition*: изменение местоположения

Класс `TranslateTransition` реализует анимацию движения путем постепенного изменения значений свойств: `translateX`, `translateY` и `translateZ` из класса `Node`. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.animation.TranslateTransition;
```

### Иерархия наследования:

Object – Animation – Transition – TranslateTransition

### Конструкторы класса:

```
TranslateTransition()
TranslateTransition(Duration duration)
TranslateTransition(Duration duration, Node node)
```

### Управлять анимацией позволяют следующие свойства:

- ❑ **fromX** — начальная координата по оси x в системе координат узла. Методы доступа:

```
public final void setFromX(double value)
public final double getFromX()
public final DoubleProperty fromXProperty()
```

- ❑ **fromY** — начальная координата по оси y. Методы доступа:

```
public final void setFromY(double value)
public final double getFromY()
public final DoubleProperty fromYProperty()
```

- ❑ **fromZ** — начальная координата по оси z. Методы доступа:

```
public final void setFromZ(double value)
public final double getFromZ()
public final DoubleProperty fromZProperty()
```

- ❑ **toX** — конечная координата по оси x. Методы доступа:

```
public final void setToX(double value)
public final double getToX()
public final DoubleProperty toXProperty()
```

- ❑ **toY** — конечная координата по оси y. Методы доступа:

```
public final void setToY(double value)
public final double getToY()
public final DoubleProperty toYProperty()
```

- ❑ **toZ** — конечная координата по оси z. Методы доступа:

```
public final void setToZ(double value)
public final double getToZ()
public final DoubleProperty toZProperty()
```

- ❑ **byX** — задает смещение по оси x. Это значение прибавляется к значению свойства **fromX**, и сумма становится конечной координатой. Если задано значение свойства **toX**, то значение свойства **byX** игнорируется. Методы доступа:

```
public final void setByX(double value)
public final double getByX()
public final DoubleProperty byXProperty()
```

- ❑ **byY** — задает смещение по оси Y. Это значение прибавляется к значению свойства **fromY**, и сумма становится конечной координатой. Если задано значение свойства **toY**, то значение свойства **byY** игнорируется. Методы доступа:

```
public final void setByY(double value)
public final double getByY()
public final DoubleProperty byYProperty()
```

- ❑ **byZ** — задает смещение по оси Z. Это значение прибавляется к значению свойства **fromZ**, и сумма становится конечной координатой. Если задано значение свойства **toZ**, то значение свойства **byZ** игнорируется. Методы доступа:

```
public final void setByZ(double value)
public final double getByZ()
public final DoubleProperty byZProperty()
```

- ❑ **duration** — продолжительность анимации. Значение по умолчанию: 400 ms. Методы доступа:

```
public final void setDuration(Duration value)
public final Duration getDuration()
public final ObjectProperty<Duration> durationProperty()
```

- ❑ **node** — узел, к которому применяется анимация. Методы доступа:

```
public final void setNode(Node value)
public final Node getNode()
public final ObjectProperty<Node> nodeProperty()
```

### Пример анимации движения прямоугольника по диагонали:

```
Rectangle rect = new Rectangle(50.0, 50.0);
rect.setFill(Color.RED);
rect.relocate(50.0, 50.0);
```

```
TranslateTransition tr = new TranslateTransition(
 Duration.seconds(10.0), rect);

tr.setAutoReverse(true);
tr.setFromX(0.0);
tr.setFromY(0.0);
tr.setToX(400.0);
tr.setToY(300.0);
tr.setCycleCount(Animation.INDEFINITE);
tr.setInterpolator(Interpolator.LINEAR);
tr.play();
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH09_02_Animation`, приведенном в каталоге `CH09_animations` сопровождающего книгу электронного архива (см. приложение).

## 9.6. Класс *PathTransition*: движение вдоль траектории

Класс `PathTransition` реализует анимацию движения вдоль заданной траектории.

Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.animation.PathTransition;
```

Иерархия наследования:

Object — Animation — Transition — PathTransition

Конструкторы класса:

```
PathTransition()
PathTransition(Duration duration, Shape path)
PathTransition(Duration duration, Shape path, Node node)
```

Управлять анимацией позволяют следующие свойства:

- ❑ `path` — задает траекторию, вдоль которой будет перемещаться узел. В качестве значения указывается объект, класс которого является наследником класса `Shape` (см. главу 5), — например, объект класса `Path`. Методы доступа:

```
public final void setPath(Shape value)
public final Shape getPath()
public final ObjectProperty<Shape> pathProperty()
```

- ❑ `orientation` — задает ориентацию узла относительно траектории. В качестве значения можно указать константы: `NONE` (ориентация не изменяется, значение по умолчанию) или `ORTHOGONAL_TO_TANGENT` (ориентация будет соответствовать траектории) из перечисления `PathTransition.OrientationType`. Методы доступа:

```
import javafx.animation.PathTransition.OrientationType;
public final void setOrientation(
 PathTransition.OrientationType value)
public final PathTransition.OrientationType getOrientation()
public final ObjectProperty<PathTransition.OrientationType>
 orientationProperty()
```

- ❑ `duration` — продолжительность анимации. Значение по умолчанию: 400 ms. Методы доступа:

```
public final void setDuration(Duration value)
public final Duration getDuration()
public final ObjectProperty<Duration> durationProperty()
```

- ❑ `node` — узел, к которому применяется анимация. Методы доступа:

```
public final void setNode(Node value)
public final Node getNode()
public final ObjectProperty<Node> nodeProperty()
```

Пример анимации движения прямоугольника вдоль квадратичной кривой:

```
Rectangle rect3 = new Rectangle(50.0, 50.0);
rect3.setFill(Color.BLUE);
rect3.relocate(50.0, 150.0);

Path p3 = new Path(
 new MoveTo(0.0, 0.0),
 new QuadCurveTo(200.0, 200.0, 400.0, 0.0)
);

PathTransition tr3 = new PathTransition();
tr3.setDuration(Duration.seconds(5.0));
tr3.setNode(rect3);
tr3.setPath(p3);
tr3.setOrientation(OrientationType.ORTHOGONAL_TO_TANGENT);
tr3.setAutoReverse(true);
tr3.setCycleCount(Animation.INDEFINITE);
tr3.setInterpolator(Interpolator.EASE_BOTH);
tr3.play();
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH09_02_Animation`, приведенном в каталоге `CH09_animations` сопровождающего книгу электронного архива (см. приложение).

## 9.7. Класс *ScaleTransition*: изменение масштаба

Класс `ScaleTransition` реализует анимацию увеличения или уменьшения масштаба путем постепенного изменения свойств: `scaleX`, `scaleY` и `scaleZ` из класса `Node`. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.animation.ScaleTransition;
```

Иерархия наследования:

Object — Animation — Transition — ScaleTransition

Конструкторы класса:

```
ScaleTransition()
ScaleTransition(Duration duration)
ScaleTransition(Duration duration, Node node)
```

Управлять анимацией позволяют следующие свойства:

□ `fromX` — начальный масштаб по оси `x`. Методы доступа:

```
public final void setFromX(double value)
public final double getFromX()
public final DoubleProperty fromXProperty()
```



- **fromY** — начальный масштаб по оси Y. Методы доступа:

```
public final void setFromY(double value)
public final double getFromY()
public final DoubleProperty fromYProperty()
```

- **fromZ** — начальный масштаб по оси Z. Методы доступа:

```
public final void setFromZ(double value)
public final double getFromZ()
public final DoubleProperty fromZProperty()
```

- **toX** — конечный масштаб по оси X. Методы доступа:

```
public final void setToX(double value)
public final double getToX()
public final DoubleProperty toXProperty()
```

- **toY** — конечный масштаб по оси Y. Методы доступа:

```
public final void setToY(double value)
public final double getToY()
public final DoubleProperty toYProperty()
```

- **toZ** — конечный масштаб по оси Z. Методы доступа:

```
public final void setToZ(double value)
public final double getToZ()
public final DoubleProperty toZProperty()
```

- **byX** — задает смещение по оси X. Это значение прибавляется к значению свойства **fromX**, и сумма становится конечным масштабом. Если задано значение свойства **toX**, то значение свойства **byX** игнорируется. Методы доступа:

```
public final void setByX(double value)
public final double getByX()
public final DoubleProperty byXProperty()
```

- **byY** — задает смещение по оси Y. Это значение прибавляется к значению свойства **fromY**, и сумма становится конечным масштабом. Если задано значение свойства **toY**, то значение свойства **byY** игнорируется. Методы доступа:

```
public final void setByY(double value)
public final double getByY()
public final DoubleProperty byYProperty()
```

- **byZ** — задает смещение по оси Z. Это значение прибавляется к значению свойства **fromZ**, и сумма становится конечным масштабом. Если задано значение свойства **toZ**, то значение свойства **byZ** игнорируется. Методы доступа:

```
public final void setByZ(double value)
public final double getByZ()
public final DoubleProperty byZProperty()
```

- **duration** — продолжительность анимации. Значение по умолчанию: 400 ms. Методы доступа:

```
public final void setDuration(Duration value)
public final Duration getDuration()
public final ObjectProperty<Duration> durationProperty()
```

❑ **node** — узел, к которому применяется анимация. Методы доступа:

```
public final void setNode(Node value)
public final Node getNode()
public final ObjectProperty<Node> nodeProperty()
```

**Пример плавного увеличения масштаба по оси X в два раза, а по оси Y — в три раза:**

```
Rectangle rect = new Rectangle(50.0, 50.0);
rect.setFill(Color.RED);
rect.relocate(50.0, 50.0);
```

```
ScaleTransition tr = new ScaleTransition(
 Duration.seconds(5.0), rect);
tr.setAutoReverse(true);
tr.setFromX(1.0);
tr.setFromY(1.0);
tr.setToX(2.0);
tr.setToY(3.0);
tr.setCycleCount(Animation.INDEFINITE);
tr.setInterpolator(Interpolator.LINEAR);
tr.play();
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH09_02_Animation`, приведенном в каталоге `CH09_animations` сопровождающего книгу электронного архива (см. приложение).

## 9.8. Класс *RotateTransition*: вращение

Класс `RotateTransition` реализует анимацию вращения путем постепенного изменения свойства `rotate` из класса `Node`. Вращение выполняется относительно центра узла. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.animation.RotateTransition;
```

**Иерархия наследования:**

```
Object — Animation — Transition — RotateTransition
```

**Конструкторы класса:**

```
RotateTransition()
RotateTransition(Duration duration)
RotateTransition(Duration duration, Node node)
```

Управлять анимацией позволяют следующие свойства:

- ❑ **axis** — задает ось, вокруг которой выполняется вращение. В параметре `value` можно указать константы: `X_AXIS`, `Y_AXIS` или `Z_AXIS` из класса `Rotate` (см. *разд. 8.1.5*). Методы доступа:

```
public final void setAxis(Point3D value)
public final Point3D getAxis()
public final ObjectProperty<Point3D> axisProperty()
```

- ❑ **fromAngle** — начальный угол. Методы доступа:

```
public final void setFromAngle(double value)
public final double getFromAngle()
public final DoubleProperty fromAngleProperty()
```

- ❑ **toAngle** — конечный угол. Методы доступа:

```
public final void setToAngle(double value)
public final double getToAngle()
public final DoubleProperty toAngleProperty()
```

- ❑ **byAngle** — задает смещение. Это значение прибавляется к значению свойства `fromAngle`, и сумма становится конечным углом. Если задано значение свойства `toAngle`, то значение свойства `byAngle` игнорируется. Методы доступа:

```
public final void setByAngle(double value)
public final double getByAngle()
public final DoubleProperty byAngleProperty()
```

- ❑ **duration** — продолжительность анимации. Значение по умолчанию: 400 ms. Методы доступа:

```
public final void setDuration(Duration value)
public final Duration getDuration()
public final ObjectProperty<Duration> durationProperty()
```

- ❑ **node** — узел, к которому применяется анимация. Методы доступа:

```
public final void setNode(Node value)
public final Node getNode()
public final ObjectProperty<Node> nodeProperty()
```

**Пример вращения прямоугольника вокруг оси Y:**

```
Rectangle rect = new Rectangle(50.0, 50.0);
rect.setFill(Color.RED);
rect.relocate(50.0, 50.0);
```

```
RotateTransition tr = new RotateTransition(
 Duration.seconds(5.0), rect);
tr.setAutoReverse(true);
tr.setAxis(Rotate.Y_AXIS);
tr.setFromAngle(0.0);
```

```
tr.setToAngle(180.0);
tr.setCycleCount(Animation.INDEFINITE);
tr.setInterpolator(Interpolator.LINEAR);
tr.play();
```

По умолчанию вращение выполняется относительно центра узла. Если нужно вращать относительно другой точки, то можно сместить начало системы координат для узла. Пример вращения прямоугольника относительно угла:

```
Rectangle rect3 = new Rectangle(50.0, 50.0);
rect3.setFill(Color.BLUE);
rect3.relocate(350.0, 50.0);
rect3.getTransforms().add(new Translate(-25.0, -25.0));
rect3.setTranslateX(25.0);
rect3.setTranslateY(25.0);
```

```
RotateTransition tr3 = new RotateTransition();
tr3.setDuration(Duration.seconds(5.0));
tr3.setNode(rect3);
tr3.setByAngle(360.0);
tr3.setCycleCount(Animation.INDEFINITE);
tr3.setInterpolator(Interpolator.LINEAR);
tr3.play();
```

### **ЭЛЕКТРОННЫЙ АРХИВ**

Полный текст программы содержится в проекте `FX_CH09_02_Animation`, приведенном в каталоге *CH09\_animations* сопровождающего книгу электронного архива (см. приложение).

## **9.9. Класс *FillTransition*: изменение цвета заливки**

Класс `FillTransition` реализует анимацию изменения цвета заливки фигуры. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.animation.FillTransition;
```

Иерархия наследования:

```
Object — Animation — Transition — FillTransition
```

Конструкторы класса:

```
FillTransition()
FillTransition(Duration duration)
FillTransition(Duration duration, Shape shape)
FillTransition(Duration duration, Color fromValue, Color toValue)
FillTransition(Duration duration, Shape shape,
 Color fromValue, Color toValue)
```

Управлять анимацией позволяют следующие свойства:

❑ **fromValue** — начальный цвет. Методы доступа:

```
public final void setFromValue(Color value)
public final Color getFromValue()
public final ObjectProperty<Color> fromValueProperty()
```

❑ **toValue** — конечный цвет. Методы доступа:

```
public final void setToValue(Color value)
public final Color getToValue()
public final ObjectProperty<Color> toValueProperty()
```

❑ **duration** — продолжительность анимации. Значение по умолчанию: 400 ms. Методы доступа:

```
public final void setDuration(Duration value)
public final Duration getDuration()
public final ObjectProperty<Duration> durationProperty()
```

❑ **shape** — фигура, к которой применяется анимация. Методы доступа:

```
public final void setShape(Shape value)
public final Shape getShape()
public final ObjectProperty<Shape> shapeProperty()
```

**Пример:**

```
Rectangle rect = new Rectangle(50.0, 50.0);
rect.setFill(Color.RED);
rect.relocate(50.0, 50.0);

FillTransition tr = new FillTransition(
 Duration.seconds(5.0), rect);

tr.setAutoReverse(true);
tr.setFromValue(Color.RED);
tr.setToValue(Color.BLUE);
tr.setCycleCount(Animation.INDEFINITE);
tr.play();
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH09_02_Animation`, приведенном в каталоге `CH09_animations` сопровождающего книгу электронного архива (см. приложение).

## 9.10. Класс *StrokeTransition*: изменение цвета обводки

Класс `StrokeTransition` реализует анимацию изменения цвета обводки фигуры. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.animation.StrokeTransition;
```

## Иерархия наследования:

Object — Animation — Transition — StrokeTransition

## Конструкторы класса:

```
StrokeTransition()
StrokeTransition(Duration duration)
StrokeTransition(Duration duration, Shape shape)
StrokeTransition(Duration duration, Color fromValue, Color toValue)
StrokeTransition(Duration duration, Shape shape,
 Color fromValue, Color toValue)
```

## Управлять анимацией позволяют следующие свойства:

### ❑ fromValue — начальный цвет. Методы доступа:

```
public final void setFromValue(Color value)
public final Color getFromValue()
public final ObjectProperty<Color> fromValueProperty()
```

### ❑ toValue — конечный цвет. Методы доступа:

```
public final void setToValue(Color value)
public final Color getToValue()
public final ObjectProperty<Color> toValueProperty()
```

### ❑ duration — продолжительность анимации. Значение по умолчанию: 400 ms. Методы доступа:

```
public final void setDuration(Duration value)
public final Duration getDuration()
public final ObjectProperty<Duration> durationProperty()
```

### ❑ shape — фигура, к которой применяется анимация. Методы доступа:

```
public final void setShape(Shape value)
public final Shape getShape()
public final ObjectProperty<Shape> shapeProperty()
```

## Пример:

```
Rectangle rect = new Rectangle(100.0, 100.0);
rect.setFill(Color.WHITE);
rect.setStroke(Color.RED);
rect.setStrokeWidth(5.0);
rect.relocate(50.0, 50.0);

StrokeTransition tr = new StrokeTransition(
 Duration.seconds(5.0), rect);
tr.setAutoReverse(true);
tr.setFromValue(Color.RED);
tr.setToValue(Color.BLUE);
tr.setCycleCount(Animation.INDEFINITE);
tr.play();
```

**ЭЛЕКТРОННЫЙ АРХИВ**

Полный текст программы содержится в проекте `FX_CH09_02_Animation`, приведенном в каталоге `CH09_animations` сопровождающего книгу электронного архива (см. приложение).

## 9.11. Класс *ParallelTransition*: параллельное выполнение нескольких анимаций

Если нужно, чтобы несколько анимаций выполнялись одновременно, то следует воспользоваться классом `ParallelTransition`. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.animation.ParallelTransition;
```

Иерархия наследования:

```
Object – Animation – Transition – ParallelTransition
```

Конструкторы класса:

```
ParallelTransition()
ParallelTransition(Animation... children)
ParallelTransition(Node node)
ParallelTransition(Node node, Animation... children)
```

Добавить анимацию после создания объекта позволяет метод `getChildren()`, который возвращает ссылку на список с анимациями. Формат метода:

```
public final ObservableList<Animation> getChildren()
```

Указать ссылку на узел, к которому применяется анимация, можно в конструкторе или с помощью свойства `node`. Методы доступа:

```
public final void setNode(Node value)
public final Node getNode()
public final ObjectProperty<Node> nodeProperty()
```

Пример анимации движения прямоугольника с одновременным изменением прозрачности:

```
FadeTransition ft = new FadeTransition(Duration.seconds(2.5));
ft.setAutoReverse(true);
ft.setFromValue(0.3);
ft.setToValue(1.0);
ft.setCycleCount(Animation.INDEFINITE);
ft.setInterpolator(Interpolator.LINEAR);
```

```
TranslateTransition tt = new TranslateTransition(
 Duration.seconds(5.0));
tt.setAutoReverse(true);
tt.setFromX(0.0);
```

```
tt.setToX(400.0);
tt.setCycleCount(Animation.INDEFINITE);
tt.setInterpolator(Interpolator.EASE_BOTH);

Rectangle rect = new Rectangle(50.0, 50.0);
rect.setFill(Color.RED);
rect.relocate(50.0, 50.0);

ParallelTransition pt = new ParallelTransition(rect);
pt.getChildren().addAll(ft, tt);
pt.play();
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH09_02_Animation`, приведенном в каталоге `CH09_animations` сопровождающего книгу электронного архива (см. приложение).

## 9.12. Класс *SequentialTransition*: последовательное выполнение нескольких анимаций

Для последовательного выполнения нескольких анимаций (друг за другом) предназначен класс `SequentialTransition`. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.animation.SequentialTransition;
```

Иерархия наследования:

Object – Animation – Transition – `SequentialTransition`

Конструкторы класса:

```
SequentialTransition()
SequentialTransition(Animation... children)
SequentialTransition(Node node)
SequentialTransition(Node node, Animation... children)
```

Добавить анимацию после создания объекта позволяет метод `getChildren()`, который возвращает ссылку на список с анимациями. Обратите внимание: добавляемые анимации не должны быть бесконечными, иначе последующие анимации никогда не будут выполнены. Формат метода:

```
public final ObservableList<Animation> getChildren()
```

Указать ссылку на узел, к которому применяется анимация, можно в конструкторе или с помощью свойства `node`. Методы доступа:

```
public final void setNode(Node value)
public final Node getNode()
public final ObjectProperty<Node> nodeProperty()
```



Вначале изменим прозрачность прямоугольника, а затем переместим его:

```
FadeTransition ft = new FadeTransition(Duration.seconds(2.5));
ft.setAutoReverse(true);
ft.setFromValue(1.0);
ft.setToValue(0.3);
ft.setCycleCount(2);
ft.setInterpolator(Interpolator.LINEAR);

TranslateTransition tt = new TranslateTransition(
 Duration.seconds(5.0));
tt.setAutoReverse(true);
tt.setFromX(0.0);
tt.setToX(400.0);
tt.setCycleCount(2);
tt.setInterpolator(Interpolator.EASE_BOTH);

Rectangle rect = new Rectangle(50.0, 50.0);
rect.setFill(Color.RED);
rect.relocate(50.0, 50.0);

SequentialTransition sqtr = new SequentialTransition(rect);
sqtr.getChildren().addAll(ft, tt);
sqtr.setCycleCount(Animation.INDEFINITE);
sqtr.play();
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH09_02_Animation`, приведенном в каталоге `CH09_animations` сопровождающего книгу электронного архива (см. приложение).

## 9.13. Класс *PauseTransition*: пауза во время выполнения анимации

При последовательном выполнении нескольких анимаций часто бывает необходимо, чтобы следующая анимация начинала выполняться после некоторой паузы. Вставить паузу между анимациями позволяет класс `PauseTransition`. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.animation.PauseTransition;
```

Иерархия наследования:

Object – Animation – Transition – `PauseTransition`

Конструкторы класса:

```
PauseTransition()
PauseTransition(Duration duration)
```

Управлять продолжительностью паузы позволяет свойство `duration`. Методы доступа:

```
public final void setDuration(Duration value)
public final Duration getDuration()
public final ObjectProperty<Duration> durationProperty()
```

Переместим прямоугольник вправо, а после паузы переместим обратно:

```
TranslateTransition tt = new TranslateTransition(
 Duration.seconds(5.0));

tt.setFromX(0.0);
tt.setToX(400.0);

TranslateTransition tt2 = new TranslateTransition(
 Duration.seconds(5.0));

tt2.setFromX(400.0);
tt2.setToX(0.0);

Rectangle rect = new Rectangle(50.0, 50.0);
rect.setFill(Color.RED);
rect.relocate(50.0, 50.0);

PauseTransition pause = new PauseTransition();
pause.setDuration(Duration.seconds(2.0));

SequentialTransition sqtr = new SequentialTransition(rect);
sqtr.getChildren().addAll(tt, pause, tt2);
sqtr.setCycleCount(Animation.INDEFINITE);
sqtr.play();
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH09_02_Animation`, приведенном в каталоге `CH09_animations` сопровождающего книгу электронного архива (см. приложение).

## 9.14. Класс *Timeline*: шкала времени

Класс `Timeline` описывает шкалу времени, на которой размещаются ключевые кадры. Ключевые кадры задают опорные значения различных свойств для определенных точек времени, а промежуточные значения свойств между ключевыми кадрами генерируются автоматически. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.animation.Timeline;
```

Иерархия наследования:

Object – Animation – Timeline

## Конструкторы класса:

```
Timeline()
Timeline(KeyFrame... keyFrames)
Timeline(double targetFramerate)
Timeline(double targetFramerate, KeyFrame... keyFrames)
```

Параметр `keyFrames` задает ключевые кадры, а параметр `targetFramerate` — частоту кадров в секунду. По умолчанию анимация создается с частотой 60 кадров в секунду.

Добавить ключевые кадры после создания объекта позволяет метод `getKeyFrames()`. Метод возвращает список со всеми ключевыми кадрами. Формат метода:

```
public final ObservableList<KeyFrame> getKeyFrames()
```

## Пример добавления ключевых кадров:

```
tm.getKeyFrames().addAll(key1, key2);
```

Класс `Timeline` переопределяет метод `stop()` из класса `Animation`. После вызова этого метода анимация останавливается, и указатель текущей позиции перемещается в начало шкалы. Формат метода:

```
public void stop()
```

## Пример плавного изменения прозрачности прямоугольника:

```
Rectangle rect = new Rectangle(50.0, 50.0);
rect.setFill(Color.RED);
rect.relocate(50.0, 50.0);

KeyFrame key1 = new KeyFrame(Duration.ZERO,
 new KeyValue(rect.opacityProperty(), 1.0));
KeyFrame key2 = new KeyFrame(Duration.seconds(2.0),
 new KeyValue(rect.opacityProperty(), 0.3));

Timeline tm = new Timeline(key1, key2);
tm.setCycleCount(10);
tm.setAutoReverse(true);
tm.play();
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH09_14_Timeline`, приведенном в каталоге `CH09_animations` сопровождающего книгу электронного архива (см. приложение).

## 9.14.1. Класс *KeyFrame*

Ключевой кадр на шкале времени описывается с помощью класса `KeyFrame`. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.animation.KeyFrame;
```

## Конструкторы класса:

```
KeyFrame(Duration time, KeyValue... values)
KeyFrame(Duration time, String name, KeyValue... values)
KeyFrame(Duration time, EventHandler<ActionEvent> onFinish,
 KeyValue... values)
KeyFrame(Duration time, String name,
 EventHandler<ActionEvent> onFinish, KeyValue... values)
KeyFrame(Duration time, String name,
 EventHandler<ActionEvent> onFinish,
 Collection<KeyValue> values)
```

Параметр `time` задает позицию ключевого кадра на шкале времени, а параметр `values` — опорные значения различных свойств. С помощью параметра `name` можно назначить именованную метку (см. *разд. 9.2.2*). Параметр `onFinished` задает ссылку на обработчик события окончания анимации ключевого кадра.

Класс `KeyFrame` содержит следующие методы:

- ❑ `getName()` — возвращает название метки, связанной с ключевым кадром, или значение `null`. Формат метода:  

```
public String getName()
```
- ❑ `getTime()` — возвращает положение ключевого кадра на шкале времени. Формат метода:  

```
public Duration getTime()
```
- ❑ `getValues()` — возвращает множество с опорными значениями. Формат метода:  

```
public Set<KeyValue> getValues()
```
- ❑ `getOnFinished()` — возвращает ссылку на обработчик окончания анимации ключевого кадра или значение `null`, если обработчик не был назначен. Формат метода:  

```
public EventHandler<ActionEvent> getOnFinished()
```

Пример одновременного изменения прозрачности и перемещения прямоугольника:

```
Rectangle rect = new Rectangle(50.0, 50.0);
rect.setFill(Color.RED);
rect.relocate(50.0, 50.0);

KeyFrame key1 = new KeyFrame(Duration.ZERO, "Первый кадр",
 new KeyValue(rect.opacityProperty(), 1.0),
 new KeyValue(rect.translateXProperty(), 0.0));
KeyFrame key2 = new KeyFrame(Duration.seconds(4.0),
 event -> {
 System.out.println("key2 OnFinished");
 },
 new KeyValue(rect.opacityProperty(), 0.3),
 new KeyValue(rect.translateXProperty(), 300.0));
```

```
Timeline tm = new Timeline(key1, key2);
tm.setCycleCount(10);
tm.setAutoReverse(true);
tm.play();
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH09_14_Timeline`, приведенном в каталоге `CH09_animations` сопровождающего книгу электронного архива (см. приложение).

## 9.14.2. Класс *KeyValue*

Класс `KeyValue` описывает опорные значения, связанные с ключевым кадром. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.animation.KeyValue;
```

Конструкторы класса:

```
KeyValue(WritableValue<T> target, T endValue)
KeyValue(WritableValue<T> target, T endValue,
 Interpolator interpolator)
```

Параметр `target` задает ссылку на свойство, значение которого будет изменяться во время анимации, а параметр `endValue` — конечное значение, связанное с ключевым кадром. Промежуточные значения свойства между ключевыми кадрами будут генерироваться автоматически. Параметр `interpolator` определяет способ прироста значения. Например, вначале анимация выполняется медленно, а под конец ускоряется, или наоборот. Значение по умолчанию: `Interpolator.LINEAR`. Напомню, что класс `Interpolator` мы уже рассматривали в *разд. 9.3*. Пример:

```
Rectangle rect = new Rectangle(50.0, 50.0);
KeyValue kv1 = new KeyValue(rect.translateXProperty(), 0.0);
KeyValue kv2 = new KeyValue(rect.translateXProperty(), 300.0,
 Interpolator.EASE_OUT);
KeyFrame key1 = new KeyFrame(Duration.ZERO, kv1);
KeyFrame key2 = new KeyFrame(Duration.seconds(3.0), kv2);
```

Класс `KeyValue` содержит следующие методы:

❑ `getTarget()` — возвращает ссылку на свойство. Формат метода:

```
public WritableValue<?> getTarget()
```

❑ `getEndValue()` — возвращает конечное значение. Формат метода:

```
public Object getEndValue()
```

❑ `getInterpolator()` — возвращает ссылку на объект, определяющий способ прироста значения. Формат метода:

```
public Interpolator getInterpolator()
```

## 9.15. Класс *AnimationTimer*

Абстрактный класс `AnimationTimer` реализует таймер, который будет срабатывать для каждого кадра. Максимальная частота срабатывания составляет 60 кадров в секунду. Такой таймер можно использовать для создания как анимации, так и игр — в сочетании с компонентом `Canvas` (см. главу 7). Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.animation.AnimationTimer;
```

Для создания таймера необходимо наследовать класс `AnimationTimer` и реализовать абстрактный метод `handle()`. Именно этот метод и будет вызываться для каждого кадра. Формат метода:

```
public abstract void handle(long now)
```

Через параметр `now` доступна метка времени в наносекундах.

Для запуска таймера предназначен метод `start()`. Формат метода:

```
public void start()
```

Остановить таймер позволяет метод `stop()`. Важно останавливать таймер явным образом, например, при закрытии окна, иначе он так и продолжит работать, пока не будут закрыты все окна приложения. Формат метода:

```
public void stop()
```

Пример анимации перемещения прямоугольника с помощью `AnimationTimer`:

```
Rectangle rect = new Rectangle(50.0, 50.0);
rect.setFill(Color.RED);
rect.relocate(50.0, 50.0);

AnimationTimer timer = new AnimationTimer() {
 @Override
 public void handle(long now) {
 if (rect.getTranslateX() > 300.0) {
 rect.setTranslateX(0.0);
 }
 else {
 rect.setTranslateX(rect.getTranslateX() + 1.0);
 }
 }
};
timer.start();
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH09_15_AnimationTimer`, приведенном в каталоге `CH09_animations` сопровождающего книгу электронного архива (см. приложение).



# ГЛАВА 10



## 3D-графика

Помимо двумерной графики, JavaFX поддерживает работу и с трехмерной графикой. Проверить возможность работы с трехмерной графикой на конкретном компьютере позволяет следующий код:

```
if (Platform.isSupported(ConditionalFeature.SCENE3D)) {
 System.out.println("SCENE3D поддерживается");
}
else System.out.println("SCENE3D НЕ поддерживается");
```

### 10.1. Класс *Point3D*

Класс `Point3D` описывает координаты точки в трехмерном пространстве и позволяет выполнять различные действия с векторами. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.geometry.Point3D;
```

Формат конструктора класса:

```
Point3D(double x, double y, double z)
```

Пример создания объекта:

```
Point3D p = new Point3D(100.0, 50.0, 20.0);
System.out.println(p); // Point3D [x = 100.0, y = 50.0, z = 20.0]
```

В классе `Point3D` определена статическая константа `ZERO`, которая возвращает объект с нулевыми координатами. Формат:

```
public static final Point3D ZERO
```

Пример вывода значения:

```
System.out.println(Point3D.ZERO);
// Point3D [x = 0.0, y = 0.0, z = 0.0]
```

Приведем основные методы класса `Point3D`:

- `getX()`, `getY()` и `getZ()` — возвращают координаты по осям *x*, *y* и *z* соответственно. Форматы методов:



```
public final double getX()
public final double getY()
public final double getZ()
```

### Пример:

```
Point3D p = new Point3D(100.0, 50.0, 20.0);
System.out.println(p.getX() + " " + p.getY() + " " + p.getZ());
// 100.0 50.0 20.0
```

- ❑ **equals()** — сравнивает две точки и возвращает значение `true`, если координаты равны. Формат метода:

```
public boolean equals(Object obj)
```

### Пример:

```
Point3D p = new Point3D(10.0, 20.0, 30.0);
Point3D p2 = new Point3D(40.0, 80.0, 20.0);
Point3D p3 = new Point3D(10.0, 20.0, 30.0);
System.out.println(p.equals(p2)); // false
System.out.println(p.equals(p3)); // true
```

- ❑ **add()** — сложение векторов. Форматы метода:

```
public Point3D add(double x, double y, double z)
public Point3D add(Point3D point)
```

### Пример:

```
Point3D p = new Point3D(10.0, 20.0, 30.0);
Point3D p2 = new Point3D(40.0, 80.0, 100.0);
System.out.println(p.add(5.0, 3.0, 2.0));
// Point3D [x = 15.0, y = 23.0, z = 32.0]
System.out.println(p.add(p2));
// Point3D [x = 50.0, y = 100.0, z = 130.0]
```

- ❑ **subtract()** — вычитание векторов. Форматы метода:

```
public Point3D subtract(double x, double y, double z)
public Point3D subtract(Point3D point)
```

### Пример:

```
Point3D p = new Point3D(10.0, 20.0, 30.0);
Point3D p2 = new Point3D(4.0, 8.0, 10.0);
System.out.println(p.subtract(5.0, 3.0, 2.0));
// Point3D [x = 5.0, y = 17.0, z = 28.0]
System.out.println(p.subtract(p2));
// Point3D [x = 6.0, y = 12.0, z = 20.0]
```

- ❑ **multiply()** — умножение вектора на скаляр. Формат метода:

```
public Point3D multiply(double factor)
```

**Пример:**

```
Point3D p = new Point3D(10.0, 20.0, 30.0);
System.out.println(p.multiply(5.0));
// Point3D [x = 50.0, y = 100.0, z = 150.0]
```

- **distance()** — возвращает расстояние между двумя точками. Форматы метода:

```
public double distance(double x1, double y1, double z1)
public double distance(Point3D point)
```

**Пример:**

```
Point3D p = new Point3D(10.0, 20.0, 30.0);
System.out.println(p.distance(40.0, 20.0, 10.0));
// 36.05551275463989
System.out.println(
 p.distance(new Point3D(10.0, 20.0, 30.0))); // 0.0
```

- **normalize()** — производит нормализацию вектора. *Нормализованным* называется вектор с длиной, равной единице. Формат метода:

```
public Point3D normalize()
```

**Пример:**

```
Point3D p = new Point3D(3.0, 4.0, 5.0);
System.out.println(p.normalize());
// Point3D [x = 0.4242640687119285, y = 0.565685424949238,
// z = 0.7071067811865475]
```

**Процесс вычисления значений выглядит следующим образом:**

```
double len = Math.sqrt(3.0 * 3.0 + 4.0 * 4.0 + 5.0 * 5.0);
System.out.println(len); // 7.0710678118654755
System.out.println(3.0 / len); // 0.4242640687119285
System.out.println(4.0 / len); // 0.565685424949238
System.out.println(5.0 / len); // 0.7071067811865475
```

**Действительно ли длина вектора равна единице? Давайте проверим:**

```
System.out.println(Math.sqrt(
 Math.pow(0.4242640687119285, 2.0) +
 Math.pow(0.565685424949238, 2.0) +
 Math.pow(0.7071067811865475, 2.0)));
// 0.9999999999999999
```

- **magnitude()** — возвращает длину вектора. Формат метода:

```
public double magnitude()
```

**Пример:**

```
Point3D p = new Point3D(3.0, 4.0, 5.0);
p = p.normalize();
System.out.println(p.magnitude()); // 0.9999999999999999
```

- ❑ `dotProduct()` — скалярное произведение векторов. Форматы метода:

```
public double dotProduct(double x, double y, double z)
public double dotProduct(Point3D vector)
```

**Пример:**

```
Point3D p = new Point3D(3.0, 2.0, 4.0);
System.out.println(p.dotProduct(new Point3D(1.0, 4.0, 2.0)));
// 19.0
System.out.println(p.dotProduct(1.0, 4.0, 2.0)); // 19.0
System.out.println(3.0 * 1.0 + 2.0 * 4.0 + 4.0 * 2.0);
// 19.0
```

- ❑ `crossProduct()` — векторное произведение векторов. Форматы метода:

```
public Point3D crossProduct(double x, double y, double z)
public Point3D crossProduct(Point3D vector)
```

**Пример:**

```
Point3D p = new Point3D(1.0, 2.0, 3.0);
Point3D p2 = new Point3D(3.0, 4.0, 5.0);
System.out.println(p.crossProduct(p2));
// Point3D [x = -2.0, y = 4.0, z = -2.0]
```

Процесс вычисления выглядит следующим образом:

```
double[] a = {1.0, 2.0, 3.0};
double[] b = {3.0, 4.0, 5.0};
System.out.println((a[1] * b[2] - a[2] * b[1]) + ", " +
 (a[2] * b[0] - a[0] * b[2]) + ", " +
 (a[0] * b[1] - a[1] * b[0]));
// -2.0, 4.0, -2.0
```

- ❑ `midpoint()` — возвращает координаты точки, расположенной посередине между двумя указанными точками. Форматы метода:

```
public Point3D midpoint(double x, double y, double z)
public Point3D midpoint(Point3D point)
```

**Пример:**

```
Point3D p = new Point3D(10.0, 20.0, 30.0);
System.out.println(
 p.midpoint(new Point3D(10.0, 80.0, 150.0)));
// Point3D [x = 10.0, y = 50.0, z = 90.0]
```

- ❑ `angle()` — возвращает угол (в градусах) между векторами или тремя точками. Если указан нулевой вектор, или если три точки не отличаются друг от друга, метод вернет значение NaN. Форматы метода:

```
public double angle(double x, double y, double z)
public double angle(Point3D point)
public double angle(Point3D p1, Point3D p2)
```

### Пример:

```
Point3D p = new Point3D(1.0, 0.0, 0.0);
Point3D p2 = new Point3D(0.0, 1.0, 0.0);
Point3D p3 = new Point3D(0.0, 0.0, 0.0);
System.out.println(p.angle(p2)); // 90.0
System.out.println(p.angle(-1.0, 0.0, 0.0)); // 180.0
System.out.println(p3.angle(p, p2)); // 90.0
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH10_01_Point3D`, приведенном в каталоге `CH10_graphics3D` сопровождающего книгу электронного архива (см. *приложение*).

## 10.2. Класс *Shape3D*

Абстрактный класс `Shape3D` является базовым классом для всех классов, реализующих трехмерные фигуры. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.shape.Shape3D;
```

### Иерархия наследования:

`Object` — `Node` — `Shape3D`

Класс `Shape3D` содержит следующие свойства:

- `material` — определяет материал поверхности фигуры. Значение по умолчанию: `null`. Если свойство имеет значение `null`, то используется объект класса `PhongMaterial` с цветом `Color.LIGHTGRAY`. Методы доступа:

```
import javafx.scene.paint.PhongMaterial;
public final void setMaterial(Material value)
public final Material getMaterial()
public final ObjectProperty<Material> materialProperty()
```

### Пример указания синего цвета:

```
Shape3D box = new Box(5.0, 5.0, 5.0);
box.setMaterial(new PhongMaterial(Color.BLUE));
```

- `drawMode` — задает режим отрисовки фигуры. В качестве значения указываются константы `FILL` (полная отрисовка фигуры с заполнением внутренней части) или `LINE` (рисуются только линии между вершинами фигуры) из перечисления `DrawMode`. Значение по умолчанию: `DrawMode.FILL`. Методы доступа:

```
import javafx.scene.shape.DrawMode;
public final void setDrawMode(DrawMode value)
public final DrawMode getDrawMode()
public final ObjectProperty<DrawMode> drawModeProperty()
```

### Пример:

```
box.setDrawMode(DrawMode.FILL);
```

- ❑ `cullFace` — определяет грани, которые не нужно отрисовывать. В качестве значения указываются константы: `BACK` (только грани заднего плана), `FRONT` (только грани переднего плана) или `NONE` (отрисовываются все грани). Значение по умолчанию: `CullFace.BACK`. Методы доступа:

```
import javafx.scene.shape.CullFace;
public final void setCullFace(CullFace value)
public final CullFace getCullFace()
public final ObjectProperty<CullFace> cullFaceProperty()
```

### Пример:

```
box.setCullFace(CullFace.BACK);
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH10_02_Shape3D`, приведенном в каталоге `CH10_graphics3D` сопровождающего книгу электронного архива (см. приложение).

## 10.3. Класс *Box*: куб

Класс `Box` описывает куб с заданными шириной, высотой и глубиной (рис. 10.1). Центр куба расположен в начале системы координат. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.shape.Box;
```

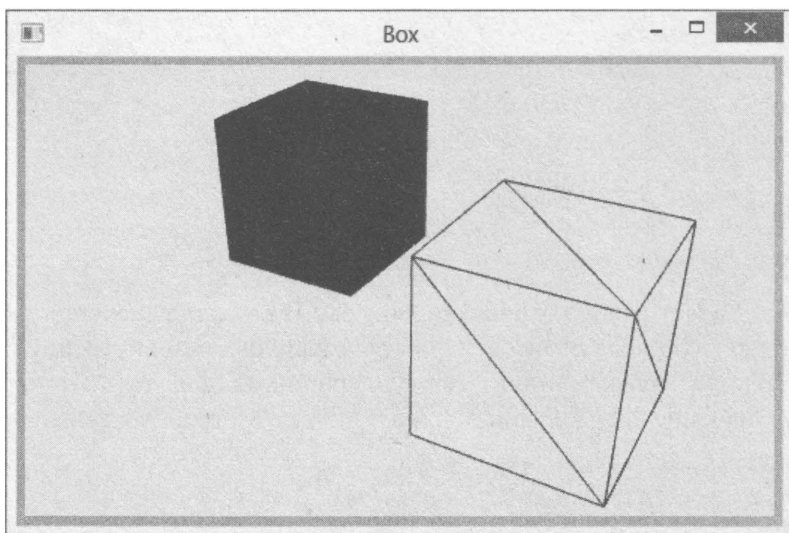


Рис. 10.1. Класс `Box`: куб

### Иерархия наследования:

Object — Node — Shape3D — Box

### Конструкторы класса:

```
Box()
Box(double width, double height, double depth)
```

В классе Box определена статическая константа DEFAULT\_SIZE, задающая размер по умолчанию. Формат:

```
public static final double DEFAULT_SIZE
```

### Выведем значение константы:

```
System.out.println(Box.DEFAULT_SIZE); // 2.0
```

### Управлять фигурой позволяют следующие свойства:

❑ width — ширина куба (размер по оси x). Значение по умолчанию: 2.0. Методы доступа:

```
public final void setWidth(double value)
public final double getWidth()
public final DoubleProperty widthProperty()
```

❑ height — высота куба (размер по оси y). Значение по умолчанию: 2.0. Методы доступа:

```
public final void setHeight(double value)
public final double getHeight()
public final DoubleProperty heightProperty()
```

❑ depth — глубина куба (размер по оси z). Значение по умолчанию: 2.0. Методы доступа:

```
public final void setDepth(double value)
public final double getDepth()
public final DoubleProperty depthProperty()
```

### Пример:

```
Box box = new Box(5.0, 5.0, 5.0);
box.setMaterial(new PhongMaterial(Color.BLUE));
box.setDrawMode(DrawMode.FILL);
box.setCullFace(CullFace.BACK);
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте FX\_CH10\_02\_Shape3D, приведенном в каталоге CH10\_graphics3D сопровождающего книгу электронного архива (см. приложение).

## 10.4. Класс *Cylinder*: цилиндр

Класс *Cylinder* описывает цилиндр (рис. 10.2). Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.shape.Cylinder;
```

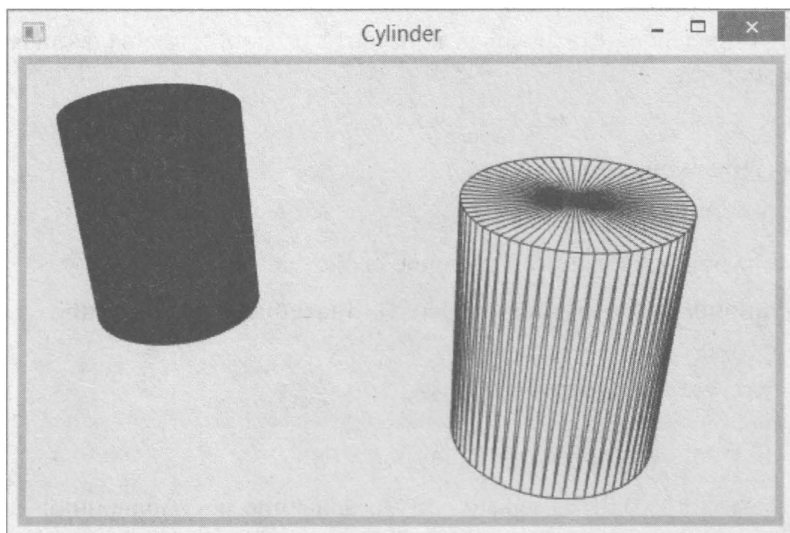


Рис. 10.2. Класс *Cylinder*: цилиндр

Иерархия наследования:

Object — Node — Shape3D — Cylinder

Конструкторы класса:

```
Cylinder()
Cylinder(double radius, double height)
Cylinder(double radius, double height, int divisions)
```

Фигура не обязательно должна быть круглой. Если в параметре *divisions* указать число 3 (это минимальное значение), то фигура будет иметь три грани. Пример:

```
Cylinder cylinder3 = new Cylinder(2.0, 5.0, 3);
cylinder3.setMaterial(new PhongMaterial(Color.CHOCOLATE));
cylinder3.setDrawMode(DrawMode.LINE);
cylinder3.setCullFace(CullFace.BACK);
```

Класс *Cylinder* содержит метод *getDivisions()*, который возвращает текущее количество граней. Формат метода:

```
public int getDivisions()
```

Управлять фигурой позволяют следующие свойства:

❑ *radius* — радиус (размер по оси *z*). Значение по умолчанию: 1.0. Методы доступа:

```
public final void setRadius(double value)
public final double getRadius()
public final DoubleProperty radiusProperty()
```

□ **height** — высота фигуры (размер по оси  $Y$ ). Значение по умолчанию: 2.0. Методы доступа:

```
public final void setHeight(double value)
public final double getHeight()
public final DoubleProperty heightProperty()
```

**Пример:**

```
Cylinder cylinder = new Cylinder(2.0, 5.0);
cylinder.setMaterial(new PhongMaterial(Color.BLUE));
cylinder.setDrawMode(DrawMode.FILL);
cylinder.setCullFace(CullFace.BACK);
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH10_02_Shape3D`, приведенном в каталоге `CH10_graphics3D` сопровождающего книгу электронного архива (см. приложение).

## 10.5. Класс *Sphere*: сфера

Класс *Sphere* описывает сферу (рис. 10.3). Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.shape.Sphere;
```

**Иерархия наследования:**

Object — Node — Shape3D — Sphere

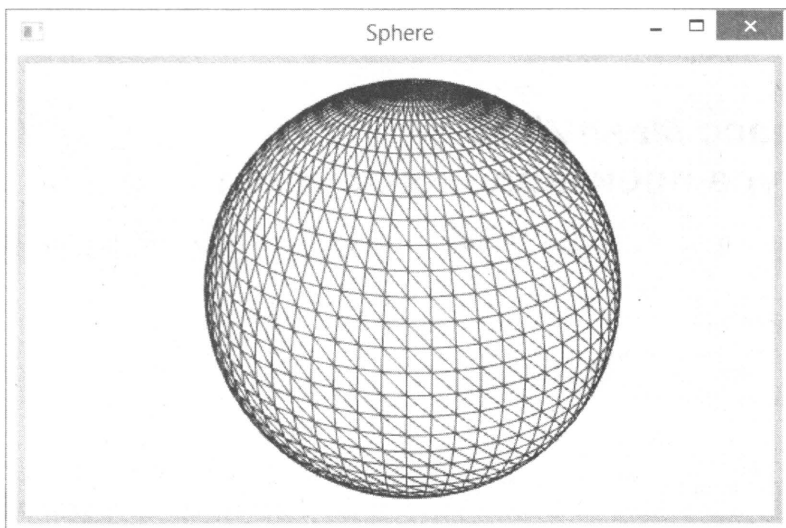


Рис. 10.3. Класс *Sphere*: сфера



**Конструкторы класса:**

```
Sphere()
Sphere(double radius)
Sphere(double radius, int divisions)
```

Фигура не обязательно должна быть круглой. Если в параметре `divisions` указать какое-либо число, то фигура будет иметь грани. Пример:

```
Sphere sphere3 = new Sphere(2.0, 12);
sphere3.setMaterial(new PhongMaterial(Color.CHOCOLATE));
sphere3.setDrawMode(DrawMode.LINE);
sphere3.setCullFace(CullFace.BACK);
```

Класс `Sphere` содержит метод `getDivisions()`, который возвращает текущее количество граней. Формат метода:

```
public int getDivisions()
```

Управлять радиусом сферы позволяет свойство `radius`. Значение по умолчанию:

**1.0. Методы доступа:**

```
public final void setRadius(double value)
public final double getRadius()
public final DoubleProperty radiusProperty()
```

**Пример:**

```
Sphere sphere = new Sphere(2.0);
sphere.setMaterial(new PhongMaterial(Color.BLUE));
sphere.setDrawMode(DrawMode.FILL);
sphere.setCullFace(CullFace.BACK);
```

**ЭЛЕКТРОННЫЙ АРХИВ**

Полный текст программы содержится в проекте `FX_CH10_02_Shape3D`, приведенном в каталоге `CH10_graphics3D` сопровождающего книгу электронного архива (см. приложение).

## 10.6. Класс *MeshView*: 3D-фигура произвольной формы

Класс `MeshView` позволяет отобразить 3D-фигуру произвольной формы. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.shape.MeshView;
```

**Иерархия наследования:**

```
Object — Node — Shape3D — MeshView
```

**Конструкторы класса:**

```
MeshView()
MeshView(Mesh mesh)
```

Класс `MeshView` содержит свойство `mesh`, которое задает отображаемую 3D-фигуру. Методы доступа:

```
public final void setMesh(Mesh value)
public final Mesh getMesh()
public final ObjectProperty<Mesh> meshProperty()
```

В качестве параметра `value` указывается объект, класс которого наследует абстрактный класс `Mesh`. На данный момент существует только один наследник — класс `TriangleMesh`, который позволяет создать 3D-фигуру, состоящую из множества треугольников. Инструкция импорта класса `TriangleMesh`:

```
// Модуль javafx.graphics
import javafx.scene.shape.TriangleMesh;
```

Конструкторы класса:

```
import javafx.scene.shape.VertexFormat;
TriangleMesh()
TriangleMesh(VertexFormat vertexFormat)
```

В параметре `vertexFormat` можно указать статические константы `POINT_TEXCOORD` или `POINT_NORMAL_TEXCOORD` из класса `VertexFormat`. Значение по умолчанию: `POINT_TEXCOORD`. Управлять значением параметра после создания объекта позволяет свойство `vertexFormat`. Методы доступа:

```
public final void setVertexFormat(VertexFormat value)
public final VertexFormat getVertexFormat()
public final ObjectProperty<VertexFormat> vertexFormatProperty()
```

После создания объекта нужно заполнить массив координатами вершин. Для этого вызываем метод `getPoints()` и добавляем координаты, например, с помощью метода `addAll()`. Формат метода `getPoints()`:

```
public final ObservableFloatArray getPoints()
```

Основные форматы метода `addAll()`:

```
public void addAll(float... elements)
public void addAll(float[] src, int srcIndex, int length)
```

Координаты точки должны быть представлены тремя числами. Первое число задает координату по оси *x*, второе — по оси *y*, а третье — по оси *z*.

В качестве примера создадим прямоугольную поверхность, которую опишем в виде двух треугольников. На поверхность наложим текстуру. Прямоугольник содержит четыре вершины, которые мы обозначим буквами *A*, *B*, *C* и *D*. Координаты этих вершин добавляем в массив:

```
TriangleMesh mesh = new TriangleMesh(VertexFormat.POINT_TEXCOORD);
mesh.getPoints().addAll(
 0.0f, 10.0f, 0.0f, // A 0
 0.0f, 0.0f, 0.0f, // B 1
```

```

 10.0f, 0.0f, 0.0f, // C 2
 10.0f, 10.0f, 0.0f // D 3
);

```

Далее нужно заполнить массив с координатами точек на текстуре. Координаты указываются в двумерном пространстве (т. е. только координаты по осям  $x$  и  $y$ ). Причем это не абсолютные координаты, а относительные в виде вещественных чисел типа `float` от `0.0f` (для 0 %) до `1.0f` (для 100 %). Координата `(0.0f, 0.0f)` соответствует левому верхнему углу изображения с текстурой, а координата `(1.0f, 1.0f)` — правому нижнему углу. Для добавления координат вызываем метод `getTexCoords()` и добавляем координаты, например, с помощью метода `addAll()`. Формат метода `getTexCoords()`:

```
public final ObservableFloatArray getTexCoords()
```

Пример добавления координат всех углов изображения с текстурой:

```

mesh.getTexCoords().addAll(
 0.0f, 0.0f, // 0
 1.0f, 0.0f, // 1
 0.0f, 1.0f, // 2
 1.0f, 1.0f // 3
);

```

Далее описываем все треугольники, вызывая метод `getFaces()`. Каждый треугольник имеет лицевую сторону и обратную. Обратная сторона не отрисовывается. Чтобы описать именно лицевую сторону треугольника, достаточно задать координаты его вершин против часовой стрелки. Формат метода `getFaces()`:

```
public final ObservableFaceArray getFaces()
```

Порядок добавления значений в этот массив зависит от формата. Для формата `VertexFormat.POINT_TEXCOORD` каждый треугольник описывается шестью целыми числами, а для формата `VertexFormat.POINT_NORMAL_TEXCOORD` — девятью целыми числами (нужно дополнительно заполнить массив нормалей с помощью метода `getNormals()`). Мы рассмотрим порядок только для формата `POINT_TEXCOORD`. Выглядит он следующим образом:

```
p1, t1, p2, t2, p3, t3
```

где `p1`, `p2` и `p3` — это индексы вершин треугольника из массива, возвращаемого методом `getPoints()`. Обратите внимание: это не индексы элементов массива, а индексы вершин. Каждая вершина описывается тремя числами. Вершины указываем в порядке против часовой стрелки.

Значения `t1`, `t2` и `t3` — это индексы точек на текстуре из массива, возвращаемого методом `getTexCoords()`.

Давайте опишем наши треугольники:

```

mesh.getFaces().addAll(
 0, 0, 1, 2, 2, 3, // ABC
 0, 0, 2, 3, 3, 1 // ACD
);

```

Теперь создадим текстуру поверхности. Для этого загружаем изображение текстуры в объект класса `Image`, а затем передаем его в метод `setDiffuseMap()` класса `PhongMaterial`:

```
PhongMaterial material = new PhongMaterial();
material.setDiffuseMap(loadImage());
```

Осталось создать объект класса `MeshView` и передать ему объекты классов `TriangleMesh` и `PhongMaterial` с помощью методов `setMesh()` и `setMaterial()` соответственно:

```
MeshView meshView = new MeshView();
meshView.setMesh(mesh);
meshView.setMaterial(material);
meshView.setDrawMode(DrawMode.FILL);
meshView.setCullFace(CullFace.NONE);
```

Точно таким же образом можно создавать произвольные 3D-фигуры, просто треугольников будет гораздо больше. Например, если мы хотим создать куб, то он содержит шесть прямоугольных граней, каждая из которых может быть описана двумя треугольниками. Таким образом, чтобы создать куб, нужно описать двенадцать треугольников и соответствующие им точки на текстуре.

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH10_02_Shape3D`, приведенном в каталоге `CH10_graphics3D` сопровождающего книгу электронного архива (см. приложение).

## 10.7. Класс *SubScene*: субсцена

Для управления свойствами трехмерного пространства класс `Scene` содержит два конструктора:

```
Scene(Parent root, double width, double height, boolean depthBuffer)
Scene(Parent root, double width, double height, boolean depthBuffer,
 SceneAntialiasing antiAliasing)
```

Если в параметре `depthBuffer` указано значение `true`, то объект сцены будет иметь буфер глубины. Проверить наличие буфера позволяет метод `isDepthBuffer()`. Формат метода:

```
public final boolean isDepthBuffer()
```

Параметр `antiAliasing` задает уровень сглаживания. Значение по умолчанию: `SceneAntialiasing.DISABLED`, поэтому грани 3D-фигуры будут отображаться лесенкой. Чтобы включить сглаживание, нужно при создании сцены указать значение `SceneAntialiasing.BALANCED`. Получить значение уровня сглаживания позволяет метод `getAntiAliasing()`. Формат метода:

```
import javafx.scene.SceneAntialiasing;
public final SceneAntialiasing getAntiAliasing()
```

Вместо использования объекта сцены для отображения трехмерного пространства можно использовать объект класса `SubScene`. Такой объект мы будем называть *субсценой*. Субсцена особенно полезна при необходимости отобразить на сцене одновременно и двумерные объекты (например, компоненты), и трехмерные (например, 3D-фигуры). Инструкция импорта класса `SubScene`:

```
// Модуль javafx.graphics
import javafx.scene.SubScene;
```

**Иерархия наследования:**

Object – Node – SubScene

**Конструкторы класса:**

```
SubScene(Parent root, double width, double height)
SubScene(Parent root, double width, double height,
 boolean depthBuffer, SceneAntialiasing antiAliasing)
```

В параметре `root` указывается ссылка на корневой контейнер, а в параметрах `width` и `height` — ширина и высота субсцены соответственно. Если в параметре `depthBuffer` указано значение `true`, то субсцена будет иметь буфер глубины. Проверить наличие буфера позволяет метод `isDepthBuffer()`. Формат метода:

```
public final boolean isDepthBuffer()
```

Параметр `antiAliasing` задает уровень сглаживания. Значение по умолчанию: `SceneAntialiasing.DISABLED`, поэтому грани 3D-фигуры будут отображаться лесенкой. Чтобы включить сглаживание, нужно при создании субсцены указать значение `SceneAntialiasing.BALANCED`. Получить значение уровня сглаживания позволяет метод `getAntiAliasing()`. Формат метода:

```
import javafx.scene.SceneAntialiasing;
public final SceneAntialiasing getAntiAliasing()
```

Управлять объектом субсцены позволяют следующие свойства:

❑ `root` — содержит ссылку на корневой контейнер. Методы доступа:

```
public final void setRoot(Parent value)
public final Parent getRoot()
public final ObjectProperty<Parent> rootProperty()
```

❑ `width` — ширина субсцены. Методы доступа:

```
public final void setWidth(double value)
public final double getWidth()
public final DoubleProperty widthProperty()
```

❑ `height` — высота субсцены. Методы доступа:

```
public final void setHeight(double value)
public final double getHeight()
public final DoubleProperty heightProperty()
```

- ❑ **fill** — задает характеристики фона. Значение по умолчанию: `null`. Методы доступа:

```
public final void setFill(Paint value)
public final Paint getFill()
public final ObjectProperty<Paint> fillProperty()
```

- ❑ **camera** — позволяет указать тип камеры. Методы доступа:

```
public final void setCamera(Camera value)
public final Camera getCamera()
public final ObjectProperty<Camera> cameraProperty()
```

**Пример добавления куба и камеры на субсцену:**

```
Box box = new Box(5.0, 5.0, 5.0);
PerspectiveCamera camera = new PerspectiveCamera(true);

Group group = new Group();
group.getChildren().addAll(box);

SubScene subScene = new SubScene(group, 500, 300, true,
 SceneAntialiasing.BALANCED);
subScene.setFill(Color.ANTIQUEWHITE);
subScene.setCamera(camera);
```

### **ЭЛЕКТРОННЫЙ АРХИВ**

Полный текст программы содержится в проекте `FX_CH10_07_SubScene`, приведенном в каталоге `CH10_graphics3D` сопровождающего книгу электронного архива (см. *приложение*).

## **10.8. Управление камерой**

Для визуализации объектов на сцене используется камера. Куда смотрит камера, та часть сцены и отображается. По умолчанию камера смотрит в направлении положительных значений оси *z*. Описывается объект камеры с помощью классов, наследующих абстрактный класс `Camera`. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.Camera;
```

**Иерархия наследования:**

Object — Node — Camera

**Класс `Camera` содержит два свойства:**

- ❑ **nearClip** — объекты, расположенные от камеры ближе значения этого свойства, не отображаются. Значение по умолчанию: `0.1`. Методы доступа:

```
public final void setNearClip(double value)
public final double getNearClip()
public final DoubleProperty nearClipProperty()
```

❑ `farClip` — объекты, расположенные от камеры дальше значения этого свойства, не отображаются. Значение по умолчанию: 100.0. Методы доступа:

```
public final void setFarClip(double value)
public final double getFarClip()
public final DoubleProperty farClipProperty()
```

Прикрепить камеру к сцене позволяет свойство `camera` из класса `Scene` (аналогичное свойство есть и в классе `SubScene`). Методы доступа:

```
public final void setCamera(Camera value)
public final Camera getCamera()
public final ObjectProperty<Camera> cameraProperty()
```

Реализациями абстрактного класса `Camera` являются классы: `ParallelCamera` (без перспективы) и `PerspectiveCamera` (с перспективой). Инструкция импорта класса `ParallelCamera`:

```
// Модуль javafx.graphics
import javafx.scene.ParallelCamera;
```

Конструктор класса `ParallelCamera`:

```
ParallelCamera()
```

Класс `ParallelCamera` наследует все методы из класса `Camera` и не добавляет никаких дополнительных методов. Камера имеет параллельные лучи, поэтому никаких перспективных искажений нет.

Инструкция импорта класса `PerspectiveCamera`:

```
// Модуль javafx.graphics
import javafx.scene.PerspectiveCamera;
```

Конструкторы класса `PerspectiveCamera`:

```
PerspectiveCamera()
PerspectiveCamera(boolean fixedEyeAtCameraZero)
```

Если параметр `fixedEyeAtCameraZero` имеет значение `true`, то положение глаз фиксируется в точке с координатами (0, 0, 0) локальной системы координат камеры. Это значение стоит использовать, если планируется перемещать камеру внутри сцены. По умолчанию параметр `fixedEyeAtCameraZero` имеет значение `false`. Проверить текущее значение позволяет метод `isFixedEyeAtCameraZero()`. Формат метода:

```
public final boolean isFixedEyeAtCameraZero()
```

Пример создания камеры, изменения ее положения и добавления камеры на суб-сцену:

```
PerspectiveCamera camera = new PerspectiveCamera(true);
double trX = 20.0, trY = 11.0, trZ = 15.0;
double aX = -30.0, aY = 230.0, aZ = 50.0;
Translate tr = new Translate(trX, trY, trZ);
```

```

Rotate rx = new Rotate(aX, Rotate.X_AXIS);
Rotate ry = new Rotate(aY, Rotate.Y_AXIS);
Rotate rz = new Rotate(aZ, Rotate.Z_AXIS);
camera.getTransforms().addAll(tr, rx, ry, rz);
SubScene subScene = new SubScene(group, 500, 300, true,
 SceneAntialiasing.BALANCED);
subScene.setFill(Color.ANTIQUEWHITE);
subScene.setCamera(camera);

```

Управлять камерой позволяют следующие свойства из класса `PerspectiveCamera`:

- ❑ `fieldOfView` — задает угол обзора в градусах. Значение по умолчанию: 30.0.

Методы доступа:

```

public final void setFieldOfView(double value)
public final double getFieldOfView()
public final DoubleProperty fieldOfViewProperty()

```

- ❑ `verticalFieldOfView` — если свойство имеет значение `true` (по умолчанию), то значение свойства `fieldOfView` применяется к вертикальной плоскости проекции, а если `false` — то к горизонтальной. Методы доступа:

```

public final void setVerticalFieldOfView(boolean value)
public final boolean isVerticalFieldOfView()
public final BooleanProperty verticalFieldOfViewProperty()

```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH10_08_Camera`, приведенном в каталоге `CH10_graphics3D` сопровождающего книгу электронного архива (см. *приложение*).

## 10.9. Управление источником света

Абстрактный класс `LightBase` описывает источник света, с помощью которого мы можем освещать объекты на сцене. Инструкция импорта:

```

// Модуль javafx.graphics
import javafx.scene.LightBase;

```

Иерархия наследования:

`Object` — `Node` — `LightBase`

Класс `LightBase` содержит два свойства:

- ❑ `color` — задает цвет источника света. Методы доступа:

```

public final void setColor(Color value)
public final Color getColor()
public final ObjectProperty<Color> colorProperty()

```

- ❑ `lightOn` — если свойство имеет значение `true`, то источник света включен, а если `false` — то выключен. Методы доступа:



```
public final void setLightOn(boolean value)
public final boolean isLightOn()
public final BooleanProperty lightOnProperty()
```

Метод `getScope()` из класса `LightBase` возвращает список узлов, освещаемых источником света. В этот список можно добавить объекты, которые мы хотим освещать на сцене. Если список пустой, то на сцене освещаются все объекты. Формат метода:

```
public ObservableList<Node> getScope()
```

Абстрактный класс `LightBase` реализуют два класса: `AmbientLight` и `PointLight`. Класс `AmbientLight` описывает источник окружающего света. Объект равномерно освещается со всех сторон. Инструкция импорта класса `AmbientLight`:

```
// Модуль javafx.graphics
import javafx.scene.AmbientLight;
```

Иерархия наследования:

```
Object — Node — LightBase — AmbientLight
```

Конструкторы класса `AmbientLight`:

```
AmbientLight()
AmbientLight(Color color)
```

Пример освещения только куба:

```
Box box = new Box(5.0, 5.0, 5.0);
AmbientLight light = new AmbientLight();
light.setColor(Color.WHITE);
light.setLightOn(true);
light.getScope().addAll(box);
```

```
Group group = new Group();
group.getChildren().addAll(box, light);
```

```
SubScene subScene = new SubScene(group, 500, 300, true,
 SceneAntialiasing.BALANCED);
```

Класс `PointLight` описывает точечный источник света. Этот источник имеет позицию в пространстве и равномерно светит во всех направлениях из этой позиции. Инструкция импорта класса `PointLight`:

```
// Модуль javafx.graphics
import javafx.scene.PointLight;
```

Иерархия наследования:

```
Object — Node — LightBase — PointLight
```

Конструкторы класса `PointLight`:

```
PointLight()
PointLight(Color color)
```

Пример освещения куба желтым светом с определенной точки:

```
PointLight light = new PointLight(Color.YELLOW);
light.setTranslateX(10.0);
light.setTranslateY(10.0);
light.setTranslateZ(-50.0);
light.getScope().addAll(box);
```

### **ЭЛЕКТРОННЫЙ АРХИВ**

Полный текст программы содержится в проекте `FX_CH10_09_Light`, приведенном в каталоге *CH10\_graphics3D* сопровождающего книгу электронного архива (см. *приложение*).



# ГЛАВА 11



## Основные компоненты

Начиная с этой главы, мы переходим к рассмотрению компонентов графического интерфейса, таких как кнопки, надписи, текстовые поля и т. д. Некоторые компоненты мы уже изучили в предыдущих главах — например, класс `ImageView` (см. *разд. 6.2*), позволяющий отобразить изображение в окне, и класс `Canvas` (см. *главу 7*), позволяющий рисовать на холсте.

Практически все компоненты графического интерфейса наследуют классы `Node` и `Parent`. Многие компоненты наследуют также класс `Region`. Следовательно, методы этих классов, которые мы рассматривали в предыдущих главах, доступны всем компонентам. Особенно хочу обратить ваше внимание на разделы от 3.2 до 3.9, в которых мы разобрались, как задать размеры компонентов и их местоположение, как изменить фон и отобразить рамку, как управлять видимостью и доступностью компонентов. Прочитайте эти разделы еще раз, прежде чем продолжать изучение материала, и не забывайте обращать внимание на иерархию наследования, которую мы будем показывать для каждого компонента.

### ПРИМЕЧАНИЕ

При использовании программы `Scene Builder` все компоненты можно найти на вкладке **Controls**.

### 11.1. Поиск узла

Как вы уже знаете, окно приложения реализуется с помощью класса `Stage`. К объекту окна мы прикрепляем объект сцены, реализуемый классом `Scene`. К объекту сцены мы прикрепляем корневой контейнер, а уже в этот контейнер добавляем узлы. Узел может быть фигурой, компонентом или контейнером, который, в свою очередь, может содержать другие узлы. В результате образуется древовидная иерархия, которая называется *графом сцены*. Давайте научимся искать узлы в этом графе.

Итак, у нас есть какой-либо узел (например, кнопка), и нам нужно получить ссылку на него. Самый простой способ получить ссылку — предварительно сохранить ссылку в переменной (например, в поле класса) при создании объекта. Все будет

хорошо, пока таких объектов не много. Если же объектов много, но все они одно-типные, то можно добавить все ссылки в массив, или в список, или в другую коллекцию. Как потом найти эту ссылку в коллекции и как понять, что найденная ссылка та, что нам нужна? Сравнивать надписи на кнопках? А если надписи одинаковые, но кнопки в разных контейнерах выполняют разные функции и переименовать их нельзя? Вот и первая проблема, с которой мы столкнулись: необходимо как-то идентифицировать узел, причем идентификатор должен быть уникальным. Добавить идентификатор к узлу позволяет свойство `id` из класса `Node`. Методы доступа:

```
public final void setId(String value)
public final String getId()
public final StringProperty idProperty()
```

**Пример добавления и получения идентификатора:**

```
Button button1 = new Button("Кнопка 1");
button1.setId("button1");
System.out.println(button1.getId()); // button1
```

Используя этот идентификатор, мы не только можем идентифицировать узел, но и назначить ему различные стили оформления в CSS-файле (о стилевом оформлении мы поговорим в *главе 19*).

С помощью метода `setUserData()` из класса `Node` мы можем сохранить пользовательские данные в узле. Формат метода:

```
public void setUserData(Object value)
```

Получить эти данные позволяет метод `getUserData()` из класса `Node`. Если данные не были сохранены ранее, то метод вернет значение `null`. Формат метода:

```
public Object getUserData()
```

**Пример:**

```
Button button1 = new Button("Кнопка 1");
button1.setId("button1");
button1.setUserData(1);
System.out.println((int)button1.getUserData()); // 1
```

### **ПРИМЕЧАНИЕ**

Методы `setUserData()` и `getUserData()` есть и в классах `Scene` и `Window`, поэтому сохранить пользовательские данные можно также в объектах сцены и окна.

Теперь идентификация узла при поиске в массиве или коллекции не составит особого труда, да и поиск прямым перебором элементов организовать не так уж и сложно. Вопрос лишь в эффективности алгоритма поиска.

Предположим, что мы нигде не сохраняли ссылку на кнопку. Просто добавили ее в какой-то контейнер и забыли в какой именно, но не забыли при этом прикрепить к ней идентификатор. Как найти эту кнопку в графе сцены?

Обычно необходимость поиска узла возникает внутри обработчика какого-либо события. Например, мы назначили обработчик события `ActionEvent.ACTION` для объекта сцены. В результате мы сможем обработать нажатие любой кнопки на сцене. Для этого достаточно получить ссылку на источник события с помощью метода `getTarget()` объекта события и выполнить приведение к типу кнопки. Пример обработки нажатия любой кнопки на сцене:

```
scene.addEventHandler(ActionEvent.ACTION, event -> {
 if (event.getTarget() instanceof Button) {
 Button b = (Button) event.getTarget();
 System.out.println(b.getText());
 System.out.println(b.getId());
 System.out.println(b.getUserData());
 }
});
```

Так как мы знаем идентификатор кнопки, то выполнить нужное действие не составит особого труда.

Имея ссылку на какой-либо узел, мы можем получить ссылку на объект сцены. Для этого предназначено свойство `scene` из класса `Node`. Методы доступа:

```
public final Scene getScene()
public final ReadOnlyObjectProperty<Scene> sceneProperty()
```

Пример получения цвета фона сцены при нажатии кнопки с идентификатором `button1`:

```
if (b.getId().equals("button1")) {
 Scene s = b.getScene();
 System.out.println("Цвет сцены " + s.getFill());
}
```

Имея ссылку на сцену, мы можем получить ссылку на объект окна. Для этого предназначено свойство `window` из класса `Scene`. Методы доступа:

```
public final Window getWindow()
public final ReadOnlyObjectProperty<Window> windowProperty()
```

Пример получения текста из заголовка окна:

```
Stage st = (Stage) s.getWindow();
System.out.println(st.getTitle());
```

Получить ссылку на сцену, имея ссылку на объект окна, позволяет метод `getScene()` из класса `Window`. Формат метода:

```
public final Scene getScene()
```

Выведем ширину сцены:

```
Scene s2 = st.getScene();
System.out.println("Ширина сцены " + s2.getWidth());
```

Имея ссылку на объект сцены, мы получаем доступ ко всем узлам внутри сцены. Для перебора всех узлов вначале нужно получить ссылку на корневой контейнер.

Управлять ссылкой на корневой контейнер позволяет свойство `root` из класса `Scene`.  
Методы доступа:

```
public final void setRoot(Parent value)
public final Parent getRoot()
public final ObjectProperty<Parent> rootProperty()
```

Получить список всех узлов внутри контейнера позволяет метод `getChildrenUnmodifiable()` из класса `Parent`. Формат метода:

```
public ObservableList<Node> getChildrenUnmodifiable()
```

Список доступен только для чтения и содержит исключительно дочерние узлы, а не весь граф сцены. Чтобы получить редактируемый список, нужно воспользоваться методом `getChildren()` из класса `Pane`. Формат метода:

```
public ObservableList<Node> getChildren()
```

Получим содержимое корневого контейнера:

```
Parent r = s2.getRoot();
for (Node n: r.getChildrenUnmodifiable()) {
 System.out.println(" - " + n);
}
```

Пример вывода:

```
- Button[id=button1, styleClass=button]'Кнопка 1'
- Button[id=button2, styleClass=button]'Кнопка 2'
- HBox@9958372
```

Как видно из результата, корневой контейнер содержит две кнопки и контейнер `HBox`. Тот факт, что мы не получили список узлов контейнера `HBox` не говорит, что контейнер пустой. Ведь мы получили только узлы первого уровня вложенности. Чтобы получить все узлы графа сцены, обычно используется рекурсивный вызов какого-либо метода. Давайте реализуем такой метод:

```
public static void printTreeNodes(Parent parent, int level) {
 String s = String.join("",
 Collections.nCopies((level - 1) * 3, " "));
 System.out.println(s + "+ " + parent.getClass().getName());
 for (Node node: parent.getChildrenUnmodifiable()) {
 if (node instanceof Parent) {
 printTreeNodes((Parent)node, level + 1);
 }
 else {
 System.out.println(s + " - " + node.getClass().getName());
 }
 }
}
```

Теперь вызовем этот метод:

```
Parent r = s2.getRoot();
printTreeNodes(r, 1);
```

Результат будет выглядеть следующим образом:

```
+ javafx.scene.layout.VBox
+ javafx.scene.control.Button
 - com.sun.javafx.scene.control.skin.LabeledText
+ javafx.scene.control.Button
 - com.sun.javafx.scene.control.skin.LabeledText
+ javafx.scene.layout.HBox
+ javafx.scene.control.Button
 - com.sun.javafx.scene.control.skin.LabeledText
+ javafx.scene.control.Button
 - com.sun.javafx.scene.control.skin.LabeledText
+ javafx.scene.layout.Pane
+ javafx.scene.control.Button
 - com.sun.javafx.scene.control.skin.LabeledText
```

Теперь мы получили содержимое контейнера `HBox`, а также всех вложенных контейнеров. Как видно из результата, кнопка также является контейнером для текстовой надписи.

Получить ссылку на родительский контейнер позволяет свойство `parent` из класса `Node`. Если узел не содержит родителя, то свойство имеет значение `null`. Методы доступа:

```
public final Parent getParent()
public final ReadOnlyObjectProperty<Parent> parentProperty()
```

**Пример:**

```
Parent p = b.getParent();
System.out.println(p); // VBox@178669ac[styleClass=root]
System.out.println(p.getParent()); // null
```

Если нам известен идентификатор узла, то для поиска узла можно воспользоваться методом `lookup()` из класса `Scene`. Формат метода:

```
public Node lookup(String selector)
```

В качестве параметра `selector` метод принимает CSS-селектор. Чтобы найти узел по идентификатору, используется следующий формат строки: `#id`. Чтобы найти узел со стилевым классом, нужно указать следующую строку: `.styleClass`. Метод возвращает ссылку на первый найденный узел или значение `null`, если узел не был найден. Пример поиска по идентификатору и стилевому классу:

```
Button b = (Button) event.getTarget();
Scene s = b.getScene();
Node nd = s.lookup("#button2");
System.out.println(nd);
// Button[id=button2, styleClass=button] 'Кнопка 2'
nd = s.lookup(".button");
System.out.println(nd);
// Button[id=button1, styleClass=button] 'Кнопка 1'
```



Стилевому классу `.button` соответствуют все кнопки. Так как кнопок у нас несколько, метод вернул ссылку на первую найденную кнопку.

Классы `Node` и `Parent` также содержат метод `lookup()`, но он просматривает не все дерево, а лишь ветвь по направлению к «листьям», начиная с текущего узла (включая этот узел). Пример:

```
Parent parent = s.lookup("#button3").getParent();
System.out.println(parent.lookup("#button4"));
// Button[id=button4, styleClass=button]'Кнопка 4'
System.out.println(parent.lookup("#button1"));
// null
```

Обратите внимание: кнопка с идентификатором `button1` не была найдена, т. к. она расположена ближе к корню дерева, а поиск производится в противоположном направлении.

Класс `Node` содержит также метод `lookupAll()`, позволяющий получить ссылки на все узлы, соответствующие указанному CSS-селектору. Метод возвращает множество с найденными узлами или пустое множество, если ничего не найдено. Просматривается не все дерево, а лишь ветвь по направлению к «листьям», начиная с текущего узла (включая этот узел). Формат метода:

```
public Set<Node> lookupAll(String selector)
```

Получим ссылки на все кнопки:

```
Node rootBox = s.getRoot();
Set<Node> set = rootBox.lookupAll(".button");
for (Node n: set) {
 System.out.println(n);
}
```

Результат выполнения:

```
Button[id=button1, styleClass=button]'Кнопка 1'
Button[id=button2, styleClass=button]'Кнопка 2'
Button[id=button3, styleClass=button]'Кнопка 3'
Button[id=button4, styleClass=button]'Кнопка 4'
Button[id=button5, styleClass=button]'Кнопка 5'
```

Мы получили все кнопки, т. к. поиск начинался с корневого контейнера.

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH11_01_searchNode`, приведенном в каталоге `CH11_components` сопровождающего книгу электронного архива (см. приложение).

## 11.2. Использование JavaFX-свойств

В предыдущих главах книги мы не раз упоминали, что управлять характеристиками различных объектов позволяют какие-либо свойства. Например, в предыдущем разделе мы говорили, что управлять ссылкой на корневой контейнер позволяет свойство `root` из класса `Scene`, и указывали методы доступа:

```
public final void setRoot(Parent value)
public final Parent getRoot()
public final ObjectProperty<Parent> rootProperty()
```

О каких свойствах мы говорили? О свойствах класса, т. е. о полях класса? И да, и нет. Мы говорили о *JavaFX-свойствах*, которые могут быть как полями класса, так и обычными локальными переменными. Если JavaFX-свойство является полем класса, то оно имеет традиционные названия методов доступа. Для указания нового значения метод имеет в названии приставку *set* — например: `setRoot()`. Если метод возвращает значение поля, то он имеет в названии приставку *get* — например: `getRoot()`, или, для логических полей, приставку *is* — например: `isSmooth()`. Для получения доступа к самому JavaFX-свойству метод в названии содержит суффикс *Property* — например: `rootProperty()`.

Учитывая, что поле класса является JavaFX-свойством, мы можем игнорировать традиционные способы доступа к полю и работать с JavaFX-свойством напрямую. Например, вместо этого кода:

```
scene.setRoot(pane);
```

мы можем написать так:

```
scene.rootProperty().setValue(pane);
```

А вместо этого:

```
System.out.println(scene.getRoot());
```

так:

```
System.out.println(scene.rootProperty().getValue());
```

Результат будет одинаковым. Просто традиционные методы, во-первых, более привычны для Java-программистов, а во-вторых, более удобны, т. к. не нужно писать много кода. Далее JavaFX-свойства мы будем называть просто «свойства», а свойства класса — полями класса.

## 11.2.1. Создание JavaFX-свойства

Библиотека JavaFX содержит большое количество свойств, описанных с помощью абстрактных классов. Почти для каждого элементарного типа данных есть соответствующий класс, описывающий JavaFX-свойство этого типа: `BooleanProperty`, `IntegerProperty`, `LongProperty`, `FloatProperty`, `DoubleProperty` и `StringProperty`. Объектные типы описываются с помощью класса `ObjectProperty<T>`. Все эти классы расположены в пакете `javafx.beans.property`. Например, инструкция импорта класса `DoubleProperty` будет выглядеть так:

```
// Модуль javafx.base
import javafx.beans.property.DoubleProperty;
```

Готовыми реализациями этих классов являются классы: `SimpleBooleanProperty`, `SimpleIntegerProperty`, `SimpleLongProperty`, `SimpleFloatProperty`, `SimpleDoubleProperty`, `SimpleStringProperty` и `SimpleObjectProperty<T>`, которые также расположены в па-

кете `javafx.beans.property`. Например, инструкция импорта класса `SimpleDoubleProperty` будет выглядеть так:

```
// Модуль javafx.base
import javafx.beans.property.SimpleDoubleProperty;
```

Давайте создадим свойство, зададим ему значение, а затем выведем это значение на консоль:

```
DoubleProperty property = new SimpleDoubleProperty();
property.setValue(20.0);
System.out.println(property.getValue()); // 20.0
property.set(40.0);
System.out.println(property.get()); // 40.0
```

**Форматы методов** `setValue()`, `getValue()`, `set()` и `get()`:

```
public void setValue(Number v)
public Double getValue()
public void set(double value)
public double get()
```

## 11.2.2. JavaFX-свойства только для чтения

Свойства могут быть доступны только для чтения. Такие свойства реализуются с помощью абстрактных классов: `ReadOnlyBooleanProperty`, `ReadOnlyIntegerProperty`, `ReadOnlyLongProperty`, `ReadOnlyFloatProperty`, `ReadOnlyDoubleProperty`, `ReadOnlyStringProperty` и `ReadOnlyObjectProperty<T>`, которые также расположены в пакете `javafx.beans.property`. Пример импорта класса `ReadOnlyDoubleProperty`:

```
// Модуль javafx.base
import javafx.beans.property.ReadOnlyDoubleProperty;
```

Чтобы создать свойство только для чтения, нужно воспользоваться классами: `ReadOnlyBooleanWrapper`, `ReadOnlyIntegerWrapper`, `ReadOnlyLongWrapper`, `ReadOnlyFloatWrapper`, `ReadOnlyDoubleWrapper`, `ReadOnlyStringWrapper` и `ReadOnlyObjectWrapper<T>`, которые также расположены в пакете `javafx.beans.property`. Все эти классы имеют метод `getReadOnlyProperty()`, который как раз и возвращает свойство только для чтения:

```
ReadOnlyDoubleWrapper wrapper = new ReadOnlyDoubleWrapper(9.0);
ReadOnlyDoubleProperty propertyReadOnly = wrapper.getReadOnlyProperty();
// propertyReadOnly.setValue(50.0); // Ошибка
System.out.println(propertyReadOnly.getValue()); // 9.0
```

Мы можем менять значение через переменную `wrapper`, а вот через переменную `propertyReadOnly` можем только читать значение. Любое изменение значения через переменную `wrapper` будет доступно и через переменную `propertyReadOnly`:

```
wrapper.setValue(50.0);
System.out.println(propertyReadOnly.getValue()); // 50.0
```

### 11.2.3. Назначение и удаление обработчиков

JavaFX-свойству можно назначить обработчик, который будет вызываться при изменении значения свойства. Такие обработчики должны реализовывать либо интерфейс `InvalidationListener`, либо интерфейс `ChangeListener<T>`.

#### Интерфейсы *InvalidationListener* и *Observable*

Функциональный интерфейс `InvalidationListener` описывает событие недостоверности значения свойства. Он содержит метод `invalidated()`, имеющий следующий формат:

```
// Модуль javafx.base
import javafx.beans.InvalidationListener;
import javafx.beans.Observable;
public void invalidated(Observable observable)
```

Методы для назначения и удаления обработчика события недостоверности значения свойства описываются в интерфейсе `Observable`. Интерфейс содержит следующие методы:

❑ `addListener()` — назначает обработчик события. Формат метода:

```
public void addListener(InvalidationListener listener)
```

❑ `removeListener()` — удаляет обработчик события. Формат метода:

```
public void removeListener(InvalidationListener listener)
```

Назначим обработчик недостоверности значения для свойства `width` объекта окна:

```
InvalidationListener invListener = new InvalidationListener() {
 @Override
 public void invalidated(Observable observable) {
 System.out.println("window invalidated()");
 }
};
window.widthProperty().addListener(invListener);
```

Так как интерфейс `InvalidationListener` является функциональным, мы можем использовать лямбда-выражения:

```
window.widthProperty().addListener(observable -> {
 System.out.println("window.widthProperty()");
 //System.out.println(window.getWidth());
});
```

Теперь при изменении ширины окна мы получим сообщение в окне консоли. Однако следует учитывать, что значение свойства не сразу пересчитывается после изменения, т. к. свойства поддерживают «ленивые» вычисления. Значение будет пересчитано только при попытке получения значения свойства. Это означает, что обработчик из нашего примера будет вызван только при первом изменении ширины окна, а потом он вызываться не будет, пока мы не захотим получить значение свойства. Только тогда значение будет пересчитано, и обработчик будет вызван при

следующем изменении значения. Попробуйте убрать комментарий перед третьей строкой, и обработчик будет вызываться постоянно.

Для удаления обработчика нужно иметь ссылку на обработчик, который использовался при назначении. Пример удаления обработчика при нажатии кнопки:

```
button.setOnAction(event -> {
 window.widthProperty().removeListener(invListener);
});
```

## Интерфейсы *ChangeListener<T>* и *ObservableValue<T>*

Функциональный интерфейс *ChangeListener<T>* описывает событие изменения значения свойства. Он содержит метод *changed()*, имеющий следующий формат:

```
// Модуль javafx.base
import javafx.beans.value.ChangeListener;
import javafx.beans.value.ObservableValue;
public void changed(ObservableValue<? extends T> observable,
 T oldValue, T newValue)
```

Через параметр *observable* доступны методы как из интерфейса *ObservableValue<T>*, так и из интерфейса *Observable*, поскольку интерфейс *ObservableValue<T>* является наследником *Observable*. Параметр *oldValue* содержит старое значение свойства, а параметр *newValue* — новое значение. Метод вызывается при любом изменении значения свойства и не зависит от «ленивых» вычислений.

Методы для назначения и удаления обработчика события изменения значения свойства описываются в интерфейсе *ObservableValue<T>*. Интерфейс содержит следующие методы:

- ❑ *addListener()* — назначает обработчик события. Формат метода:
 

```
public void addListener(ChangeListener<? super T> listener)
```
- ❑ *removeListener()* — удаляет обработчик события. Формат метода:
 

```
public void removeListener(ChangeListener<? super T> listener)
```
- ❑ *getValue()* — возвращает новое значение свойства. Формат метода:
 

```
T getValue()
```

При изменении значения ползунок (компонент *Slider*) отобразим его значение в текстовом поле (компонент *TextField*):

```
ChangeListener<? super Number>
changeLlistener = new ChangeListener<Number>() {
 @Override
 public void changed(ObservableValue<? extends Number> observable,
 Number oldValue, Number newValue) {
 textField.setText(newValue.toString());
 }
};
slider.valueProperty().addListener(changeLlistener);
```

Так как интерфейс `ChangeListener<T>` является функциональным, мы можем использовать лямбда-выражения. Изменим положение прямоугольника (фигура `Rectangle`) по оси `x` при изменении положения ползунка (компонент `Slider`):

```
slider.valueProperty().addListener((obj, oldValue, newValue) -> {
 rect.setTranslateX(newValue.doubleValue());
});
```

Для удаления обработчика нужно иметь ссылку на обработчик, который использовался при назначении. Пример удаления обработчика при нажатии кнопки:

```
button.setOnAction(event -> {
 slider.valueProperty().removeListener(changeListener);
});
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH11_02_property`, приведенном в каталоге `CH11_components` сопровождающего книгу электронного архива (см. приложение).

## 11.2.4. Двухнаправленное связывание

В предыдущем разделе мы изменяли позицию прямоугольника по оси `x` внутри обработчика события изменения значения ползунка с помощью следующего кода:

```
slider.valueProperty().addListener((obj, oldValue, newValue) -> {
 rect.setTranslateX(newValue.doubleValue());
});
```

В большинстве технологий это действие выполняется только таким способом. Но не в JavaFX! Помимо назначения обработчиков, JavaFX-свойства поддерживают *связывание*. Мы можем связать значение одного свойства со значением другого свойства. Если изменить значение первого свойства, то автоматически изменится значение и второго. Изменим предыдущую инструкцию и используем связывание:

```
rect.translateXProperty().bindBidirectional(slider.valueProperty());
```

Эта инструкция реализует *двухнаправленное связывание*. Любая попытка изменить значение одного свойства приведет к автоматическому изменению значения второго свойства. Например, если мы напишем следующую инструкцию:

```
rect.setTranslateX(20.0);
```

то изменится и значение ползунка:

```
System.out.println(slider.getValue()); // 20.0
```

Если изменим значение ползунка, то изменится и положение прямоугольника:

```
slider.setValue(50.0);
System.out.println(rect.getTranslateX()); // 50.0
```

Свойства связаны друг с другом, причем эта связь действует в обе стороны. Поэтому такое связывание и называется *двухнаправленным*.

Метод `bindBidirectional()` объявлен в интерфейсе `Property<T>`, который реализуют все JavaFX-свойства, позволяющие изменять значение. Формат метода:

```
public void bindBidirectional(Property<T> other)
```

Что делать, если мы хотим выводить значение ползунка в текстовом поле и изменять его значение из текстового поля? Ведь типы данных значений этих свойств не совпадают: свойство ползунка имеет тип `DoubleProperty`, а свойство текстового поля — тип `StringProperty`. Для решения этой проблемы класс `StringProperty` содержит два дополнительных формата метода `bindBidirectional()`:

```
public <T> void bindBidirectional(Property<T> other,
 StringConverter<T> converter)
public void bindBidirectional(Property<?> other, Format format)
```

В первом формате во втором параметре указывается объект, класс которого реализует абстрактный класс `StringConverter<T>`. Готовые реализации находятся в пакете `javafx.util.converter`. Их очень много. Например: `BooleanStringConverter`, `NumberStringConverter`, `IntegerStringConverter`, `LongStringConverter`, `FloatStringConverter`, `DoubleStringConverter`, `DateTimeStringConverter`, `LocalDateTimeStringConverter`, `LocalTimeStringConverter` и др. Полный список смотрите в документации. Пример:

```
// Модуль javafx.base
// import javafx.util.converter.NumberStringConverter;
textField.textProperty().bindBidirectional(slider.valueProperty(),
 new NumberStringConverter());
```

Во втором формате во втором параметре указывается объект, класс которого реализует абстрактный класс `Format` из пакета `java.text`. Готовыми реализациями являются классы: `NumberFormat`, `DateFormat`, `MessageFormat` и их наследники. Полный список смотрите в документации. Пример:

```
// Модуль java.base
// import java.text.NumberFormat;
textField2.textProperty().bindBidirectional(slider.valueProperty(),
 NumberFormat.getInstance());
```

Отменить двунаправленное связывание позволяет метод `unbindBidirectional()` из интерфейса `Property<T>`. Формат метода:

```
public void unbindBidirectional(Property<T> other)
```

Поскольку связывание является двунаправленным, то при одинаковых типах данных мы можем выполнить отмену через любое из этих двух свойств. Например, так:

```
rect.translateXProperty().unbindBidirectional(slider.valueProperty());
```

или так:

```
slider.valueProperty().unbindBidirectional(rect.translateXProperty());
```

В классе `StringProperty` метод `unbindBidirectional()` имеет также следующий формат:

```
public void unbindBidirectional(Object other)
```

Пример отмены связывания при нажатии кнопки:

```
button.setOnAction(event -> {
 textField.textProperty().unbindBidirectional(
 slider.valueProperty());
 button.setDisable(true);
});
```

В этом случае мы можем отменить связывание только через свойство текстового поля.

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH11_02_property`, приведенном в каталоге `CH11_components` сопровождающего книгу электронного архива (см. приложение).

## 11.2.5. Однонаправленное связывание

Помимо двунаправленного связывания свойства поддерживают *однонаправленное связывание*. Управлять однонаправленным связыванием позволяют следующие методы из интерфейса `Property<T>`:

❑ `bind()` — создает однонаправленное связывание. Формат метода:

```
public void bind(ObservableValue<? extends T> observable)
```

❑ `unbind()` — отменяет однонаправленное связывание. Формат метода:

```
public void unbind()
```

❑ `isBound()` — возвращает значение `true`, если свойство связано с другим свойством, и `false` — в противном случае. Формат метода:

```
public boolean isBound()
```

В качестве примера создадим однонаправленное связывание двух ползунков:

```
System.out.println(slider2.valueProperty().isBound()); // false
slider2.valueProperty().bind(slider.valueProperty());
System.out.println(slider2.valueProperty().isBound()); // true
```

Теперь при изменении значения свойства ползунка `slider` будет автоматически изменяться значение свойства ползунка `slider2`. При этом изменить значение свойства ползунка `slider2` явным образом будет нельзя. Например, следующая инструкция не изменит значение:

```
slider2.setValue(10.0); // Эффекта не будет!
System.out.println(slider2.getValue()); // 0.0
```

Пользователь также не сможет передвинуть ползунок `slider2`. А вот в обратную сторону все будет работать:



```
slider.setValue(50.0); // OK
System.out.println(slider2.getValue()); // 50.0
```

Отменим связывание при нажатии на кнопку и сделаем кнопку неактивной:

```
button.setOnAction(event -> {
 slider2.valueProperty().unbind();
 button.setDisable(true);
});
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH11_02_property`, приведенном в каталоге `CH11_components` сопровождающего книгу электронного архива (см. приложение).

## 11.2.6. Выражения

Итак, мы научились связывать свойства, но значения свойств при этом были одинаковыми. Что делать, если значение одного свойства должно отличаться от значения другого свойства на какое-либо число? Например, значение свойства ползунка `slider2` должно быть больше значения свойства ползунка `slider` на 50.0. В этом случае можно воспользоваться выражениями, классы которых расположены в пакете `javafx.beans.binding`. Например: `BooleanExpression`, `IntegerExpression`, `LongExpression`, `FloatExpression`, `DoubleExpression`, `StringExpression`, `ObjectExpression<T>` и др. Полный список смотрите в документации. Для каждого типа определены свои выражения и классы свойств наследуют их.

В качестве примера рассмотрим основные методы класса `DoubleExpression`:

❑ `add()` — сложение. Форматы метода:

```
public DoubleBinding add(int other)
public DoubleBinding add(long other)
public DoubleBinding add(float other)
public DoubleBinding add(double other)
public DoubleBinding add(ObservableNumberValue other)
```

Возвращаемый объект реализует интерфейс `ObservableValue<Number>`, поэтому мы можем использовать его для однонаправленного связывания. Пример решения нашей проблемы с ползунками:

```
slider2.valueProperty().bind(
 slider.valueProperty().add(50.0));
```

Пример вывода результата выражения:

```
DoubleProperty property = new SimpleDoubleProperty(20.0);
System.out.println(property.add(10.0).get()); // 30.0
```

❑ `subtract()` — вычитание. Форматы метода:

```
public DoubleBinding subtract(int other)
public DoubleBinding subtract(long other)
```

```
public DoubleBinding subtract(float other)
public DoubleBinding subtract(double other)
public DoubleBinding subtract(ObservableNumberValue other)
```

**Пример:**

```
System.out.println(property.subtract(10).get()); // 10.0
```

❑ **multiply() — умножение. Форматы метода:**

```
public DoubleBinding multiply(int other)
public DoubleBinding multiply(long other)
public DoubleBinding multiply(float other)
public DoubleBinding multiply(double other)
public DoubleBinding multiply(ObservableNumberValue other)
```

**Пример:**

```
System.out.println(property.multiply(10).get()); // 200.0
```

❑ **divide() — деление. Форматы метода:**

```
public DoubleBinding divide(int other)
public DoubleBinding divide(long other)
public DoubleBinding divide(float other)
public DoubleBinding divide(double other)
public DoubleBinding divide(ObservableNumberValue other)
```

**Пример:**

```
DoubleProperty property2 = new SimpleDoubleProperty(5.0);
System.out.println(property.divide(property2).get()); // 4.0
```

❑ **negate() — изменяет знак на противоположный. Формат метода:**

```
public DoubleBinding negate()
```

**Пример:**

```
System.out.println(property.negate().get()); // -20.0
```

❑ **asObject() — создает связанное объектное свойство. Пример:**

```
DoubleProperty doubleProperty = new SimpleDoubleProperty(15.0);
ObjectProperty<Double> objectProperty =
 new SimpleObjectProperty<>(5.0);
objectProperty.bind(doubleProperty.asObject());
```

```
System.out.println(doubleProperty.getValue()); // 15.0
System.out.println(objectProperty.getValue()); // 15.0
```

❑ **intValue(), longValue(), floatValue() и doubleValue() — выполняют преобразования типов. Форматы методов:**

```
public int intValue()
public long longValue()
public float floatValue()
public double doubleValue()
```

**Пример:**

```
System.out.println(property.intValue()); // 20
```

Кроме того, класс `DoubleExpression` наследует все методы из класса `NumberExpressionBase`. В этом классе определены методы для выполнения операций сравнения: `greaterThan()`, `greaterThanOrEqualTo()`, `isEqualTo()`, `isNotEqualTo()`, `lessThan()` и `lessThanOrEqualTo()`. Можно сравнивать как с числовым значением, так и со свойством. Пример:

```
DoubleProperty property = new SimpleDoubleProperty(20.0);
System.out.println(property.greaterThan(10.0).get()); // true
System.out.println(property.greaterThan(30).get()); // false
System.out.println(
 property.greaterThanOrEqualTo(20.0).get()); // true
System.out.println(
 property.isEqualTo(19.99999, 0.0001).get()); // true
System.out.println(
 property.isNotEqualTo(19.99999, 0.0001).get()); // false
System.out.println(property.lessThan(10.0).get()); // false
System.out.println(property.lessThan(30).get()); // true
System.out.println(
 property.lessThanOrEqualTo(20.0).get()); // true
```

Класс `NumberExpressionBase` содержит также метод `asString()`. Форматы метода:

```
public StringBinding asString()
public StringBinding asString(String format)
public StringBinding asString(Locale locale, String format)
```

**Пример:**

```
DoubleProperty property = new SimpleDoubleProperty(5.023545664);
System.out.println(property.asString().get()); // 5.023545664
System.out.println(property.asString("%.3f").get()); // 5,024
System.out.println(property.asString(
 new Locale("en", "US"), "%.2f").get()); // 5.02
System.out.println(property.toString());
// DoubleProperty [value: 5.023545664]
```

Помимо свойств, содержащих вещественные значения, при рассмотрении компонентов мы будем часто сталкиваться со свойствами, которые содержат строки. Такие свойства описываются абстрактным классом `StringProperty`. Создать свойство позволяет класс `SimpleStringProperty`. Пример создания свойства и вывода его значения:

```
// import javafx.beans.property.StringProperty;
// import javafx.beans.property.SimpleStringProperty;
StringProperty property = new SimpleStringProperty("строка");
System.out.println(property.getValue()); // строка
System.out.println(property.get()); // строка
```

Класс `StringProperty` наследует следующие основные методы из класса `StringExpression`:

- ❑ `getValueSafe()` — возвращает строку. В отличие от методов `getValue()` и `get()`, если свойство не содержит значение, то метод вернет пустую строку, а не значение `null`. Формат метода:

```
public final String getValueSafe()
```

Пример:

```
StringProperty property2 = new SimpleStringProperty();
System.out.println(property2.getValue()); // null
System.out.println(property2.get()); // null
System.out.println(
 "\"" + property2.getValueSafe() + "\""); // ''
System.out.println(
 "\"" + property.getValueSafe() + "\""); // 'строка'
```

- ❑ `isEmpty()` и `isNotEmpty()` — проверка на пустоту строки. Формат метода:

```
public BooleanBinding isEmpty()
public BooleanBinding isNotEmpty()
```

Пример:

```
System.out.println(property.isEmpty().get()); // false
System.out.println(property2.isEmpty().get()); // true
System.out.println(property.isNotEmpty().get()); // true
System.out.println(property2.isNotEmpty().get()); // false
```

Пример связывания свойства `disable` объекта кнопки со значением пустоты текстового поля:

```
TextField textField = new TextField();
Button btn = new Button("Отправить");
btn.disableProperty().bind(textField.textProperty().isEmpty());
```

Если поле не содержит текст, то кнопка будет неактивна. При вводе текста кнопка станет активной и ее можно будет нажать;

- ❑ `isNull()` и `isNotNull()` — проверка существования значения. Форматы методов:

```
public BooleanBinding isNull()
public BooleanBinding isNotNull()
```

Пример:

```
System.out.println(property.isNull().get()); // false
System.out.println(property2.isNull().get()); // true
System.out.println(property.isNotNull().get()); // true
System.out.println(property2.isNotNull().get()); // false
```

- ❑ `length()` — длина строки. Формат метода:

```
public IntegerBinding length()
```

**Пример:**

```
System.out.println(property.length().get()); // 6
System.out.println(property2.length().get()); // 0
```

**□ concat() — конкатенация строк. Формат метода:**

```
public StringExpression concat(Object other)
```

**Пример:**

```
System.out.println(property.concat("2").get()); // строка2
```

**□ greaterThan(), greaterThanOrEqualTo(), isEqualTo(), isEqualToIgnoreCase(), isNotEqualTo(), isNotEqualToIgnoreCase(), lessThan() и lessThanOrEqualTo() — позволяют выполнить сравнение. Пример:**

```
System.out.println(
 property.isEqualTo("строка").get()); // true
System.out.println(
 property.isEqualTo("СтРоКа").get()); // false
System.out.println(
 property.isEqualToIgnoreCase("строка").get()); // true
System.out.println(
 property.isEqualToIgnoreCase("СтРоКа").get()); // true
```

**ЭЛЕКТРОННЫЙ АРХИВ**

Полный текст программы содержится в проекте `FX_CH11_02_property`, приведенном в каталоге `CH11_components` сопровождающего книгу электронного архива (см. *приложение*).

JavaFX-свойства — это очень мощный и гибкий инструмент, позволяющий не только хранить значения, но и организовывать связи между этими значениями. Мы рассмотрели лишь маленькую часть этой вселенной. Достаточно лишь сказать, что, например, класс `SimpleDoubleProperty` наследует пять классов (не считая класс `Object`) и реализует десять интерфейсов! Вот такие вот «простые» (слово «simple» переводится как «простой») классы... Мы еще не раз будем рассматривать примеры использования JavaFX-свойств при изучении компонентов.

## 11.3. Классы *Labeled* и *Label*

Абстрактный класс `Labeled` является базовым классом для всех классов, компоненты которых содержат текстовую надпись. Помните, в *разд. 11.1* при выводе всех дочерних узлов мы получили вот такой результат?

```
+ javafx.scene.control.Button
- com.sun.javafx.scene.control.skin.LabeledText
```

Здесь кнопка внутри содержит текстовую надпись, которая и отображается на ней. Класс `Button` наследует класс `Labeled`, а следовательно, и все методы этого класса,

которые позволяют управлять характеристиками надписи. Пример создания кнопки и вывода на ней надписи **Нажми меня**:

```
Button button = new Button("Нажми меня");
```

**Инструкция импорта класса Labeled:**

```
// Модуль javafx.controls
import javafx.scene.control.Labeled;
```

**Иерархия наследования:**

Object — Node — Parent — Region — Control — Labeled

Реализацией абстрактного класса Labeled является класс Label. Именно этот класс мы будем использовать в наших примерах. Инструкция импорта класса Label:

```
// Модуль javafx.controls
import javafx.scene.control.Label;
```

**Иерархия наследования:**

Object — Node — Parent — Region — Control — Labeled — Label

### **ПРИМЕЧАНИЕ**

Для вывода надписи в окне можно также воспользоваться классом Text (см. разд. 5.18). Разница между классами Text и Label состоит в том, что класс Text описывает фигуру, а класс Label — компонент.

**Конструкторы класса Label:**

```
Label()
Label(String text)
Label(String text, Node graphic)
```

**Первый конструктор создает объект надписи, содержащий пустую строку:**

```
Label label = new Label();
System.out.println(label.getText().length()); // 0
label.setText("Текст надписи");
```

**Второй конструктор позволяет сразу указать текст надписи. Отобразим текст на двух строках:**

```
Label label2 = new Label("Текст\n на двух строках");
```

**Третий конструктор позволяет помимо текста отобразить изображение. По умолчанию изображение отображается слева от текста. Пример вывода текста под изображением:**

```
ImageView imageView = new ImageView();
imageView.setImage(createImage(80, 80, Color.BLUE));
Label label3 = new Label("Текст под изображением", imageView);
label3.setContentDisplay(ContentDisplay.TOP);
```

Во многих дальнейших примерах для создания изображения мы будем пользоваться статическим методом createImage(). Метод содержит следующий код:

```

public static Image createImage(int width, int height, Color color) {
 WritableImage wim = new WritableImage(width, height);
 PixelWriter pw = wim.getPixelWriter();
 for (int x = 0; x < width; x++) {
 for (int y = 0; y < height; y++) {
 pw.setColor(x, y, color);
 }
 }
 return wim;
}

```

**Класс Labeled содержит следующие свойства:**

- ❑ **text** — содержит текст надписи. Если надо перенести часть текста на другую строку, то нужно внутри текста указать символ перевода строки `\n`. Методы доступа:

```

public final void setText(String value)
public final String getText()
public final StringProperty textProperty()

```

- ❑ **font** — задает характеристики шрифта с помощью класса `Font` (см. *разд. 5.18.3*). Методы доступа:

```

public final void setFont(Font value)
public final Font getFont()
public final ObjectProperty fontProperty()

```

**Пример:**

```

Label label4 = new Label("Verdana BOLD ITALIC 12.0");
label4.setFont(Font.font("Verdana", FontWeight.BOLD,
 FontPosture.ITALIC, 12.0));

```

- ❑ **underline** — если свойство имеет значение `true`, то текст будет подчеркнут. Значение по умолчанию: `false`. Методы доступа:

```

public final void setUnderline(boolean value)
public final boolean isUnderline()
public final BooleanProperty underlineProperty()

```

- ❑ **textFill** — задает цвет текста. Методы доступа:

```

public final void setTextFill(Paint value)
public final Paint getTextFill()
public final ObjectProperty<Paint> textFillProperty()

```

Отообразим белый подчеркнутый текст на черном фоне. Напомню, что характеристики фона задаются с помощью класса `Background` (см. *разд. 3.5*):

```

Label label5 = new Label("Подчеркнутый текст");
label5.setUnderline(true);
label5.setFont(new Font(24.0));

```

```
label5.setBackground(new Background(
 new BackgroundFill(Color.BLACK, null, null)));
label5.setTextFill(Color.WHITE);
```

- ❑ **ellipsisString** — задает строку, которая будет отображаться при обрезке текста. Текст обрезается, если он не помещается на одной строке или внутри области. Значение по умолчанию: .... Методы доступа:

```
public final void setEllipsisString(String value)
public final String getEllipsisString()
public final StringProperty ellipsisStringProperty()
```

- ❑ **textOverflow** — задает режим обрезки текста, если он не помещается внутри области. Методы доступа:

```
public final void setTextOverflow(OverflowStyle value)
public final OverflowStyle getTextOverflow()
public final ObjectProperty<OverflowStyle> textOverflowProperty()
```

В параметре `value` указываются следующие константы из перечисления `OverflowStyle`:

```
// Модуль javafx.controls
import javafx.scene.control.OverflowStyle;
```

- **ELLIPSIS** — текст обрезается в конце и добавляется значение свойства `ellipsisString` (значение по умолчанию);
- **WORD\_ELLIPSIS** — аналогично **ELLIPSIS**, но обрезка выполняется между словами, а не в любом месте слова;
- **CENTER\_ELLIPSIS** — текст обрезается посередине и добавляется значение свойства `ellipsisString`;
- **CENTER\_WORD\_ELLIPSIS** — аналогично **CENTER\_ELLIPSIS**, но обрезка выполняется между словами, а не в любом месте слова;
- **LEADING\_ELLIPSIS** — текст обрезается в начале и добавляется значение свойства `ellipsisString`;
- **LEADING\_WORD\_ELLIPSIS** — аналогично **LEADING\_ELLIPSIS**, но обрезка выполняется между словами, а не в любом месте слова;
- **CLIP** — текст просто обрезается в конце.

Пример обрезки посередине:

```
Label label2 = new Label(createLongText(20));
label2.setTextOverflow(OverflowStyle.CENTER_WORD_ELLIPSIS);
```

Результат:

Длинная строка слово3 слово...o17 слово18 слово19 слово20

Если использовать константу `CENTER_WORD_ELLIPSIS`, то результат будет таким:

Длинная строка слово3...слово18 слово19 слово20



- ❑ `wrapText` — если свойство имеет значение `true`, и текст не помещается на одной строке, он будет переноситься на новую строку. По умолчанию свойство имеет значение `false`, при котором длинный текст обрезается. Методы доступа:

```
public final void setWrapText(boolean value)
public final boolean isWrapText()
public final BooleanProperty wrapTextProperty()
```

- ❑ `lineSpacing` — расстояние между строками при использовании многострочного режима. Значение по умолчанию: `0.0`. Методы доступа:

```
public final void setLineSpacing(double value)
public final double getLineSpacing()
public final DoubleProperty lineSpacingProperty()
```

- ❑ `textAlignment` — задает выравнивание текста при использовании многострочного режима. Значение по умолчанию: `TextAlignment.LEFT`. Методы доступа:

```
public final void setTextAlignment(TextAlignment value)
public final TextAlignment getTextAlignment()
public final ObjectProperty<TextAlignment>
 textAlignmentProperty()
```

В качестве значения параметра `value` можно указать следующие константы из перечисления `TextAlignment`:

```
// Модуль javafx.graphics
import javafx.scene.text.TextAlignment;
```

- `LEFT` — по левому краю области (значение по умолчанию);
- `RIGHT` — по правому краю области;
- `CENTER` — по центру области;
- `JUSTIFY` — по ширине области.

**Пример выравнивания длинного текста по ширине:**

```
Label label = new Label(createLongText(20));
label.setWrapText(true);
label.setTextAlignment(TextAlignment.JUSTIFY);
```

- ❑ `graphic` — задает изображение, которое будет отображаться вместе с текстом или вместо текста. Значение по умолчанию: `null`. Методы доступа:

```
public final void setGraphic(Node value)
public final Node getGraphic()
public final ObjectProperty<Node> graphicProperty()
```

**Пример:**

```
ImageView imageView = new ImageView();
imageView.setImage(createImage(80, 80, Color.BLUE));
Label label = new Label("Текст");
label.setGraphic(imageView);
```

- `contentDisplay` — задает положение изображения относительно текста. Значение по умолчанию: `ContentDisplay.LEFT` (изображение выводится слева от текста). Методы доступа:

```
public final void setContentDisplay(ContentDisplay value)
public final ContentDisplay getContentDisplay()
public final ObjectProperty<ContentDisplay>
 contentDisplayProperty()
```

В качестве значения параметра `value` можно указать следующие константы из перечисления `ContentDisplay`:

```
// Модуль javafx.controls
import javafx.scene.control.ContentDisplay;
```

- `LEFT` — изображение выводится слева от текста (значение по умолчанию);
- `RIGHT` — изображение выводится справа от текста;
- `CENTER` — изображение выводится по центру текста;
- `TOP` — изображение выводится над текстом;
- `BOTTOM` — изображение выводится под текстом;
- `TEXT_ONLY` — выводится только текст;
- `GRAPHIC_ONLY` — выводится только изображение.

Пример вывода текста под изображением:

```
Label label4 = new Label("ContentDisplay.TOP");
label4.setGraphic(
 new ImageView(createImage(80, 80, Color.RED)));
label4.setContentDisplay(ContentDisplay.TOP);
```

- `graphicTextGap` — задает расстояние между изображением и текстом. Методы доступа:

```
public final void setGraphicTextGap(double value)
public final double getGraphicTextGap()
public final DoubleProperty graphicTextGapProperty()
```

- `alignment` — задает выравнивание текста и изображения внутри свободной области компонента. Если этой свободной области нет, то и эффекта не будет видно. В качестве параметра указываются следующие константы из перечисления `javafx.geometry.Pos`: `BASELINE_CENTER`, `BASELINE_LEFT`, `BASELINE_RIGHT`, `BOTTOM_CENTER`, `BOTTOM_LEFT`, `BOTTOM_RIGHT`, `CENTER`, `CENTER_LEFT`, `CENTER_RIGHT`, `TOP_CENTER`, `TOP_LEFT` и `TOP_RIGHT`. Значение по умолчанию: `Pos.CENTER_LEFT`. Методы доступа:

```
public final void setAlignment(Pos value)
public final Pos getAlignment()
public final ObjectProperty<Pos> alignmentProperty()
```

- `labelPadding` — возвращает отступ от границы компонента до текста и изображения. Это значение не учитывает значение свойства `padding` из класса `Region`.

Задать значение свойству `labelPadding` можно только через стиль: `-fx-label-padding`. Методы доступа:

```
public final Insets getLabelPadding()
public final ReadOnlyObjectProperty<Insets>
 labelPaddingProperty()
```

**Пример:**

```
Label label = new Label("Текст");
label.setStyle("-fx-label-padding: 20.0px;");
```

- `mnemonicParsing` — разрешает (значение `true`) или запрещает (значение `false`) разбор текста надписи для назначения клавиши быстрого доступа. Чтобы задать клавишу быстрого доступа, следует в тексте надписи указать символ `_` (подчеркивание) перед буквой. В этом случае буква, перед которой указан символ `_`, будет подчеркнута, что является подсказкой пользователю. При одновременном нажатии клавиши `<Alt>` и подчеркнутой буквы компонент окажется в фокусе ввода. Напомню, что назначение клавиш быстрого доступа мы уже рассматривали в *разд. 4.8.5. Методы доступа*:

```
public final void setMnemonicParsing(boolean value)
public final boolean isMnemonicParsing()
public final BooleanProperty mnemonicParsingProperty()
```

**Пример назначения клавиши быстрого доступа для кнопки:**

```
Button button = new Button("_Button");
button.setMnemonicParsing(true);
```

Некоторые компоненты (например, текстовое поле) не имеют надписи. Чтобы задать клавиши быстрого доступа для таких компонентов, необходимо отдельно создать надпись и связать ее с компонентом. Управлять связыванием позволяет свойство `labelFor` из класса `Label`. Методы доступа:

```
public final void setLabelFor(Node value)
public final Node getLabelFor()
public ObjectProperty<Node> labelForProperty()
```

**Пример связывания надписи с однострочным текстовым полем:**

```
Label labelForText = new Label("Tex_t:");
TextField text = new TextField();
labelForText.setMnemonicParsing(true);
labelForText.setLabelFor(text);
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH11_03_Label`, приведенном в каталоге `CH11_components` сопровождающего книгу электронного архива (см. *приложение*).

## 11.4. Класс *ButtonBase*

Абстрактный класс `ButtonBase` является базовым классом для кнопок, переключателей, флажков и гиперссылок. Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.ButtonBase;
```

Иерархия наследования:

`Object` — `Node` — `Parent` — `Region` — `Control` — `Labeled` — `ButtonBase`

Класс `ButtonBase` содержит следующие свойства:

□ `onAction` — позволяет назначить обработчик нажатия кнопки. Методы доступа:

```
public final void setOnAction(
 EventHandler<ActionEvent> value)
public final EventHandler<ActionEvent> getOnAction()
public final ObjectProperty<EventHandler<ActionEvent>>
 onActionProperty()
```

Пример назначения обработчика с помощью лямбда-выражения:

```
ButtonBase button = new Button("На_жми меня");
button.setOnAction(event -> {
 System.out.println("Нажата кнопка");
});
```

Символ подчеркивания внутри текста задает клавишу быстрого доступа.

Пример назначения обработчика с помощью анонимного класса:

```
button.setOnAction(new EventHandler<ActionEvent>() {
 @Override
 public void handle(ActionEvent event) {
 System.out.println("Нажата кнопка");
 }
});
```

Пример назначения обработчика с помощью метода `addEventHandler()`:

```
button.addEventHandler(ActionEvent.ACTION,
 new EventHandler<ActionEvent>() {
 @Override
 public void handle(ActionEvent event) {
 System.out.println("Нажата кнопка");
 }
});
```

□ `armed` — содержит значение `true`, если кнопка была нажата с помощью мыши, и `false` — если из программы, например, путем вызова метода `fire()` или в результате нажатия клавиш быстрого доступа. Методы доступа:

```
public final boolean isArmed()
public final ReadOnlyBooleanProperty armedProperty()
```

С помощью метода `fire()` можно сгенерировать событие нажатия кнопки из программы. Формат метода:

```
public abstract void fire()
```

Пример:

```
button.fire();
```

В этом случае свойство `armed` будет иметь значение `false`. Имитировать нажатие кнопки пользователем позволяет следующий код:

```
button.arm();
button.requestFocus();
PauseTransition tr = new PauseTransition(Duration.seconds(1.0));
tr.setOnFinished(e -> {
 button.fire();
 button.disarm();
});
tr.play();
```

В этом случае свойство `armed` будет иметь значение `true`. Метод `arm()` имитирует нажатие кнопки мыши над командной кнопкой (кнопка становится темно-серой), а метод `disarm()` имитирует выход указателя мыши из области командной кнопки (кнопка примет первоначальный вид). Форматы методов:

```
public void arm()
public void disarm()
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH11_04_ButtonBase`, приведенном в каталоге `CH11_components` сопровождающего книгу электронного архива (см. приложение).

## 11.5. Класс *Hyperlink*: гиперссылка

Класс `Hyperlink` позволяет отобразить гиперссылку в окне. Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.Hyperlink;
```

Иерархия наследования:

```
Object — Node — Parent — Region — Control — Labeled — ButtonBase —
Hyperlink
```

Как видно из иерархии, гиперссылка — это и текстовая надпись (наследует класс `Labeled`), и кнопка (наследует класс `ButtonBase`). Не забывайте обращать внимание на иерархию наследования!

Конструкторы класса `Hyperlink`:

```
Hyperlink()
Hyperlink(String text)
Hyperlink(String text, Node graphic)
```

В параметре `text` указывается текст ссылки. Пример обработки перехода по гиперссылке:

```
Hyperlink hyperlink = new Hyperlink("http://www.google.com");
hyperlink.setOnAction(event -> {
 System.out.println("Переход по гиперссылке");
});
```

Чтобы запустить Web-браузер, используемый в системе по умолчанию, и передать ему адрес Web-страницы, можно воспользоваться следующим кодом:

```
// import java.awt.Desktop;
if (Desktop.isDesktopSupported()) {
 try {
 Desktop.getDesktop().browse(URI.create("http://www.google.com"));
 } catch (Exception e) {
 System.out.println("Не удалось запустить Web-браузер");
 }
}
```

Параметр `graphic` позволяет отобразить изображение рядом с текстом ссылки или вместо него. Пример:

```
Hyperlink hyperlink3 = new Hyperlink("Текст ссылки с изображением",
 new ImageView(createImage(16, 16, Color.BLUE)));
hyperlink3.setOnAction(event -> {
 System.out.println("Переход по гиперссылке3");
});
```

Класс `Hyperlink` содержит свойство `visited`. Если это свойство имеет значение `true`, то гиперссылка помечается как посещенная. Гиперссылка автоматически помечается посещенной после первого же перехода по ней. Методы доступа:

```
public final void setVisited(boolean value)
public final boolean isVisited()
public final BooleanProperty visitedProperty()
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH11_04_ButtonBase`, приведенном в каталоге `CH11_components` сопровождающего книгу электронного архива (см. приложение).

## 11.6. Класс *Button*: командная кнопка

Командная кнопка (рис. 11.1) является наиболее часто используемым компонентом. При нажатии кнопки внутри обработчика обычно выполняется какая-либо операция. Кнопка реализуется с помощью класса `Button`. Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.Button;
```

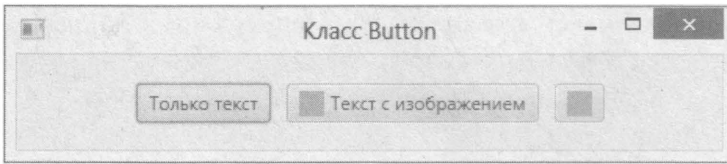


Рис. 11.1. Класс Button: командная кнопка

### Иерархия наследования:

Object — Node — Parent — Region — Control — Labeled — ButtonBase — Button

### Конструкторы класса:

```
Button()
Button(String text)
Button(String text, Node graphic)
```

В параметре `text` указывается текст, который отобразится на кнопке. Обратите внимание: для кнопки свойство `mnemonicParsing` из класса `Labeled` установлено в положение `true`, поэтому разбор текста для назначения клавиши быстрого доступа выполняется по умолчанию. Пример обработки нажатия кнопки:

```
Button button = new Button("На_жми меня");
button.setOnAction(event -> {
 System.out.println("Нажата кнопка");
});
```

Параметр `graphic` позволяет отобразить изображение рядом с текстом или вместо него. Пример вывода только изображения:

```
Button button3 = new Button("Кнопка 3",
 new ImageView(createImage(32, 32, Color.RED)));
button3.setContentDisplay(ContentDisplay.GRAPHIC_ONLY);
button3.setOnAction(new EventHandler<ActionEvent>() {
 @Override
 public void handle(ActionEvent event) {
 System.out.println("Нажата кнопка3");
 }
});
```

Класс `Button` содержит следующие свойства:

- `defaultButton` — если свойство содержит значение `true`, то кнопка будет обозначена как кнопка по умолчанию. Нажать кнопку по умолчанию можно с помощью клавиши `<Enter>`. Чтобы нажать обычную кнопку, находящуюся в фокусе ввода, нужно нажать клавишу `<Пробел>`. Методы доступа:

```
public final void setDefaultButton(boolean value)
public final boolean isDefaultButton()
public final BooleanProperty defaultButtonProperty()
```

□ `cancelButton` — если свойство содержит значение `true`, то кнопка будет обозначена как кнопка отмены. Нажать кнопку отмены можно с помощью клавиши `<Esc>`. Методы доступа:

```
public final void setCancelButton(boolean value)
public final boolean isCancelButton()
public final BooleanProperty cancelButtonProperty()
```

Часто после нажатия кнопки приходится выполнять какую-либо длительную операцию. В этом случае, чтобы пользователь не нажимал кнопку повторно несколько раз, думая, что программа «зависла», нужно сделать кнопку недоступной (см. *разд. 3.9*). Если узел недоступен, то он не принимает событий, а, следовательно, при попытке нажать недоступную кнопку обработчик вызван не будет. Пример:

```
Button button6 = new Button("Кнопка 6");
button6.setOnAction(event -> {
 System.out.println("Нажата кнопка 6");
 button6.setDisable(true);
 PauseTransition tr = new PauseTransition(Duration.seconds(2.0));
 tr.setOnFinished(e -> {
 System.out.println("Операция завершена");
 button6.setDisable(false);
 });
 tr.play();
});
```

В этом примере мы воспользовались классом `PauseTransition` для имитации выполнения длительного процесса. После нажатия кнопки мы делаем ее недоступной, затем выполняем операцию, а после завершения выполнения операции выводим результат и делаем кнопку доступной.

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH11_04_ButtonBase`, приведенном в каталоге `CH11_components` сопровождающего книгу электронного архива (см. *приложение*).

## 11.7. Класс *ButtonBar*

Порядок отображения стандартных кнопок в разных операционных системах будет различным. Так как библиотека `JavaFX` является кроссплатформенной, мы должны уважать привычки пользователей и выводить кнопки в порядке, принятом в операционной системе пользователя. Класс `ButtonBar` позволяет помочь нам в этом. Он описывает горизонтальный контейнер, в который мы можем добавить кнопки, указав их предназначение. Согласно предназначению, кнопки будут автоматически размещены внутри контейнера в том порядке, который принят в текущей операционной системе. Инструкция импорта класса `ButtonBar`:

```
// Модуль javafx.controls
import javafx.scene.control.ButtonBar;
```



## Иерархия наследования:

Object — Node — Parent — Region — Control — ButtonBar

## Конструкторы класса:

```
ButtonBar()
ButtonBar(String buttonOrder)
```

Первый конструктор задает порядок следования кнопок по умолчанию (зависит от текущей операционной системы, на которой выполняется приложение), а второй конструктор позволяет указать нужный порядок в виде строки специального формата. Сформировать такие строки для основных операционных систем позволяют статические константы: `BUTTON_ORDER_NONE` (порядок следования кнопок будет соответствовать порядку внутри списка, который возвращает метод `getButtons()`, а кнопки будут выровнены по правому краю контейнера), `BUTTON_ORDER_WINDOWS`, `BUTTON_ORDER_LINUX` и `BUTTON_ORDER_MAC_OS` из класса `ButtonBar`. Выведем их значения:

```
System.out.println(ButtonBar.BUTTON_ORDER_NONE.length());
// 0
System.out.println(ButtonBar.BUTTON_ORDER_WINDOWS);
// L_E+U+FBXI_YNOCAH_R
System.out.println(ButtonBar.BUTTON_ORDER_LINUX);
// L_HE+UNYACBXIO_R
System.out.println(ButtonBar.BUTTON_ORDER_MAC_OS);
// L_HE+U+FBIX_NCYOA_R
```

Указать предназначение кнопки позволяет статический метод `setButtonData()` из класса `ButtonBar`. Для получения текущего предназначения кнопки можно воспользоваться методом `getButtonData()`. Форматы методов:

```
public static void setButtonData(Node button,
 ButtonBar.ButtonData buttonData)
public static ButtonBar.ButtonData getButtonData(Node button)
```

В параметре `button` указывается ссылка на кнопку, а в параметре `buttonData` — предназначение кнопки. В качестве значения параметра `buttonData` можно указать следующие константы из перечисления `ButtonBar.ButtonData`: `YES`, `NO`, `OK_DONE`, `CANCEL_CLOSE`, `FINISH`, `APPLY`, `BACK_PREVIOUS`, `NEXT_FORWARD`, `LEFT`, `RIGHT`, `SMALL_GAP`, `BIG_GAP`, `HELP`, `HELP_2` или `OTHER`.

Получить букву, соответствующую предназначению, позволяет метод `getTypeCode()` из перечисления `ButtonBar.ButtonData`. Эту букву можно указать в строке формата во втором конструкторе класса `ButtonBar`. Выведем букву, соответствующую константе `YES`:

```
System.out.println(ButtonData.YES.getTypeCode()); // Y
```

Для добавления кнопки в контейнер предназначен метод `getButtons()`, который возвращает ссылку на список дочерних узлов. Формат метода:

```
public final ObservableList<Node> getButtons()
```

Пример создания кнопок, указания предназначения и добавления в контейнер:

```
Button btnYes = new Button("Yes");
ButtonBar.setButtonData(btnYes, ButtonData.YES);

Button btnNo = new Button("No");
ButtonBar.setButtonData(btnNo, ButtonData.NO);

Button btnCancel = new Button("Cancel");
ButtonBar.setButtonData(btnCancel, ButtonData.CANCEL_CLOSE);

ButtonBar buttonBar = new ButtonBar();
buttonBar.getButtons().addAll(btnYes, btnNo, btnCancel);
```

Класс `ButtonBar` содержит следующие свойства:

- ❑ `buttonOrder` — задает порядок следования кнопок в виде строки специального формата (как во втором конструкторе класса `ButtonBar`). Методы доступа:

```
public final void setButtonOrder(String buttonOrder)
public final String getButtonOrder()
public final StringProperty buttonOrderProperty()
```

- ❑ `buttonMinWidth` — задает минимальную ширину для всех кнопок внутри контейнера. Методы доступа:

```
public final void setButtonMinWidth(double value)
public final double getButtonMinWidth()
public final DoubleProperty buttonMinWidthProperty()
```

По умолчанию ширина всех кнопок будет одинаковой. Если необходимо, чтобы какая-либо кнопка не зависела от размера других кнопок, то следует вызвать статический метод `setButtonUniformSize()` из класса `ButtonBar` и передать во втором параметре значение `false`. Формат метода:

```
public static void setButtonUniformSize(Node button,
 boolean uniformSize)
```

Получить текущее значение позволяет статический метод `isButtonUniformSize()`. Формат метода:

```
public static boolean isButtonUniformSize(Node button)
```

Пример:

```
Button btnYes = new Button("Yes");
ButtonBar.setButtonData(btnYes, ButtonData.YES);
btnYes.setPrefWidth(150.0);
ButtonBar.setButtonUniformSize(btnYes, false);
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH11_04_ButtonBase`, приведенном в каталоге `CH11_components` сопровождающего книгу электронного архива (см. приложение).

## 11.8. Переключатели

Переключатели могут находиться в двух основных состояниях: включен и выключен. В JavaFX переключатели реализуются двумя классами: `ToggleButton` и `RadioButton`.

### 11.8.1. Класс *ToggleButton* и интерфейс *Toggle*

Класс `ToggleButton` реализует переключатель в виде обычной кнопки, имеющей дополнительно нажатое (включенное) состояние (рис. 11.2). Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.ToggleButton;
```

Иерархия наследования:

```
Object — Node — Parent — Region — Control — Labeled — ButtonBase —
ToggleButton
```

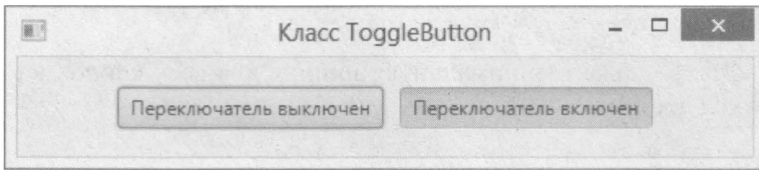


Рис. 11.2. Класс `ToggleButton`: переключатель в виде обычной кнопки

Конструкторы класса:

```
ToggleButton()
ToggleButton(String text)
ToggleButton(String text, Node graphic)
```

Пример создания переключателя и назначения обработчиков изменения статуса:

```
ToggleButton button = new ToggleButton("Текст на кнопке");
button.selectedProperty().addListener((obj, oldValue, newValue) -> {
 System.out.println("button selected = " + newValue);
});
button.setOnAction(event -> {
 if (button.isSelected()) {
 System.out.println("Переключатель включен");
 }
 else {
 System.out.println("Переключатель выключен");
 }
});
```

Управлять переключателем позволяют следующие свойства из класса `ToggleButton` (эти свойства и их методы доступа описаны в интерфейсе `Toggle`):

- `selected` — содержит значение `true`, если переключатель включен, и значение `false` — если выключен. Методы доступа:

```
public final void setSelected(boolean value)
public final boolean isSelected()
public final BooleanProperty selectedProperty()
```

- `toggleGroup` — содержит ссылку на группу, к которой относится переключатель, или значение `null`, если переключатель не был добавлен в группу. Методы доступа:

```
public final void setToggleGroup(ToggleGroup value)
public final ToggleGroup getToggleGroup()
public final ObjectProperty<ToggleGroup> toggleGroupProperty()
```

Интерфейс `Toggle` содержит также следующие методы (эти методы класс `ToggleButton` наследует из базовых классов):

```
public ObservableMap<Object, Object> getProperties()
public void setUserData(Object value)
public Object getUserData()
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH11_04_ButtonBase`, приведенном в каталоге `CH11_components` сопровождающего книгу электронного архива (см. приложение).

## 11.8.2. Класс *RadioButton*

Класс `RadioButton` реализует переключатель в виде кружка с точкой внутри и текстом рядом (рис. 11.3). Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.RadioButton;
```

Инструкция импорта:

```
Object — Node — Parent — Region — Control — Labeled — ButtonBase —
ToggleButton — RadioButton
```

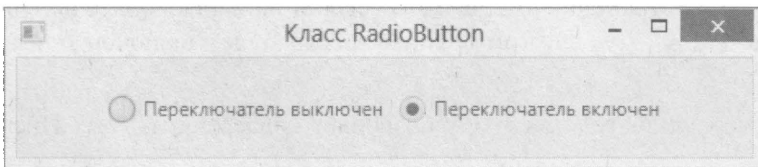


Рис. 11.3. Класс `RadioButton`: переключатель в виде кружка с точкой внутри

Конструкторы класса:

```
RadioButton()
RadioButton(String text)
```

Пример создания переключателя и назначения обработчиков изменения статуса:

```
RadioButton button = new RadioButton("Текст");
button.selectedProperty().addListener((obj, oldValue, newValue) -> {
 System.out.println("button selected = " + newValue);
});
button.setOnAction(event -> {
 if (button.isSelected()) {
 System.out.println("Переключатель включен");
 }
 else {
 System.out.println("Переключатель выключен");
 }
});
```

Класс `RadioButton` наследует все методы из базовых классов и не добавляет никаких общедоступных методов. Так как класс `RadioButton` наследует класс `ToggleButton`, то он также реализует интерфейс `Toggle`.

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH11_04_ButtonBase`, приведенном в каталоге `CH11_components` сопровождающего книгу электронного архива (см. *приложение*).

## 11.8.3. Класс *ToggleGroup*: объединение переключателей в группу

Обычно переключатели организуются в группы. Внутри группы может быть включен только один переключатель. При попытке включить другой переключатель, ранее включенный переключатель автоматически отключается. Поэтому переключатели часто называют *радиокнопками* (принцип их работы такой же, как и у кнопок переключения диапазонов в старых радиоприемниках).

Обратите внимание: при использовании переключателей из класса `ToggleButton` можно выключить все переключатели внутри группы, а, значит, внутри обработчика можно получить значение `null` вместо ссылки на переключатель. При использовании класса `RadioButton` внутри группы всегда будет включен один какой-либо переключатель.

Объединить переключатели в группу позволяет класс `ToggleGroup`. Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.ToggleGroup;
```

Иерархия наследования:

Object — `ToggleGroup`

## Конструктор класса:

```
ToggleGroup()
```

Для добавления переключателя в группу можно воспользоваться методом `setToggleGroup()` из класса `ToggleButton`. Можно также добавить переключатели в список, возвращаемый методом `getToggles()` из класса `ToggleGroup`. Пример создания переключателей, добавления к ним данных, объединения их в группу и обработки события изменения состояния переключателей:

```
RadioButton btn = new RadioButton("Переключатель 1");
btn.setUserData("btn");
RadioButton btn2 = new RadioButton("Переключатель 2");
btn2.setUserData("btn2");
RadioButton btn3 = new RadioButton("Переключатель 3");
btn3.setUserData("btn3");
```

```
ToggleGroup group = new ToggleGroup();
btn.setToggleGroup(group);
btn2.setToggleGroup(group);
btn3.setToggleGroup(group);
group.selectToggle(btn);
```

```
group.selectedToggleProperty().addListener(
 (obj, oldValue, newValue) -> {
 if (newValue instanceof Toggle) {
 System.out.println("group selected = " +
 newValue.getUserData());
 }
 });
```

Класс `ToggleGroup` содержит свойство `selectedToggle`, которое позволяет получить ссылку на включенный переключатель внутри группы, а также обработать событие изменения состояния переключателей. Обратите внимание: свойство может содержать значение `null`, если ни один переключатель внутри группы не включен. Методы доступа:

```
public final Toggle getSelectedToggle()
public final ReadOnlyObjectProperty<Toggle> selectedToggleProperty()
```

Класс `ToggleGroup` содержит следующие методы:

❑ `selectToggle()` — позволяет включить указанный переключатель. Остальные переключатели внутри группы будут выключены. Формат метода:

```
public final void selectToggle(Toggle value)
```

❑ `getToggles()` — возвращает ссылку на список, содержащий ссылки на переключатели внутри группы. Формат метода:

```
public final ObservableList<Toggle> getToggles()
```

### Пример добавления переключателей в группу:

```
ToggleGroup group = new ToggleGroup();
group.getToggles().addAll(btn, btn2, btn3);
```

- `setUserData()`, `getUserData()`, `getProperties()` и `hasProperties()` — позволяют работать с пользовательскими данными, сохраненными в объекте группы. Форматы методов:

```
public void setUserData(Object value)
public Object getUserData()
public final ObservableMap<Object, Object> getProperties()
public boolean hasProperties()
```

### Пример:

```
System.out.println(group.hasProperties()); // false
group.setUserData(1);
System.out.println(group.getUserData()); // 1
System.out.println(group.hasProperties()); // true
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH11_04_ButtonBase`, приведенном в каталоге `CH11_components` сопровождающего книгу электронного архива (см. приложение).

## 11.9. Класс *CheckBox*: флажок

Флажок предназначен для включения или выключения каких-либо опций настроек программы пользователем и может иметь несколько состояний: установлен, сброшен и `indeterminate` (рис. 11.4). Флажок реализуется с помощью класса `CheckBox`. Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.CheckBox;
```

### Иерархия наследования:

`Object` — `Node` — `Parent` — `Region` — `Control` — `Labeled` — `ButtonBase` — `CheckBox`

### Конструкторы класса:

```
CheckBox()
CheckBox(String text)
```

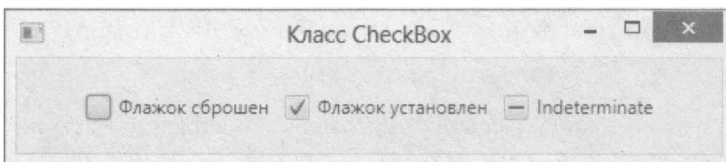


Рис. 11.4. Класс `CheckBox`: флажок

Управлять компонентом позволяют следующие свойства:

- ❑ **selected** — содержит значение `true`, если флажок установлен. Методы доступа:  

```
public final void setSelected(boolean value)
public final boolean isSelected()
public final BooleanProperty selectedProperty()
```
- ❑ **indeterminate** — содержит значение `true`, если флажок находится в неопределенном состоянии, или значение `false` — в противном случае. В неопределенном состоянии внутри рамки отображается символ `–`. Методы доступа:  

```
public final void setIndeterminate(boolean value)
public final boolean isIndeterminate()
public final BooleanProperty indeterminateProperty()
```
- ❑ **allowIndeterminate** — если свойство имеет значение `true`, то пользователь может выбрать одно из трех состояний флажка (установлен, сброшен или не определен), а если `false` (значение по умолчанию) — то только одно из двух (установлен или сброшен). Методы доступа:  

```
public final void setAllowIndeterminate(boolean value)
public final boolean isAllowIndeterminate()
public final BooleanProperty allowIndeterminateProperty()
```

Пример использования флажка с тремя состояниями:

```
CheckBox checkBox = new CheckBox("Текст рядом с флажком");
checkBox.setAllowIndeterminate(true);
checkBox.setIndeterminate(true);
checkBox.selectedProperty().addListener((obj, oldValue, newValue) -> {
 System.out.println("checkBox selected = " + newValue);
});
checkBox.indeterminateProperty().addListener(
 (obj, oldValue, newValue) -> {
 System.out.println("checkBox indeterminate = " + newValue);
 });
```

Выведем состояние флажка после нажатия кнопки:

```
Button button = new Button("Проверить статус флажка");
button.setOnAction(event -> {
 if (checkBox.isIndeterminate()) {
 System.out.println("Флажок в неопределенном состоянии");
 }
 else {
 if (checkBox.isSelected()) {
 System.out.println("Флажок установлен");
 }
 else {
 System.out.println("Флажок сброшен");
 }
 }
});
```



## ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH11_04_ButtonBase`, приведенном в каталоге `CH11_components` сопровождающего книгу электронного архива (см. приложение).

## 11.10. Класс *Slider*: шкала с ползунком

Класс `Slider` реализует шкалу с ползунком (рис. 11.5), который можно перемещать с помощью указателя мыши или клавиатуры. Шкала может иметь горизонтальную ориентацию (по умолчанию) или вертикальную. Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.Slider;
```

Иерархия наследования:

`Object` — `Node` — `Parent` — `Region` — `Control` — `Slider`

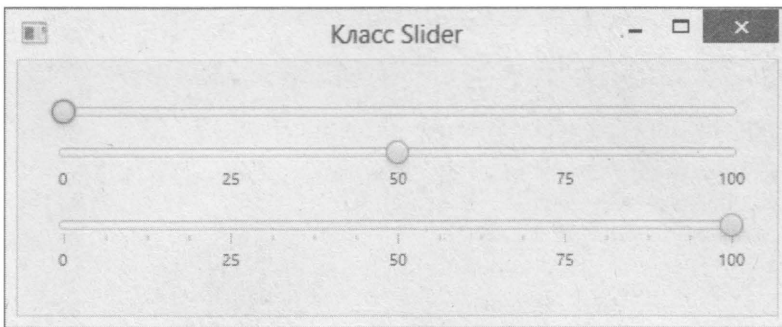


Рис. 11.5. Класс `Slider`: шкала с ползунком

Конструкторы класса:

```
Slider()
Slider(double min, double max, double value)
```

Во втором конструкторе первый параметр задает минимальное значение, второй — максимальное значение, а третий — текущее значение. Пример создания шкалы и обработки изменения значения:

```
Slider slider = new Slider(0.0, 100.0, 50.0);
slider.setShowTickMarks(true);
slider.setShowTickLabels(true);
slider.setMajorTickUnit(25.0);
slider.setMinorTickCount(4);
slider.setBlockIncrement(5.0);
slider.valueProperty().addListener((obj, oldValue, newValue) -> {
 System.out.println("slider value = " + newValue);
});
```

Пример связывания значения ползунка с текстовым полем. Если поле доступно для изменения пользователем, то нужно дополнительно указать способ преобразования значения и фильтр символов с помощью метода `setTextFormatter()` из класса `TextInputControl` (см. *разд. 12.2.7*), иначе возможны сообщения об ошибках преобразования в окне консоли:

```
TextField text = new TextField();
text.setPrefWidth(100.0);
// text.setTextFormatter(...);
text.textProperty().bindBidirectional(slider.valueProperty(),
 new NumberStringConverter());
```

Пример получения значения ползунка внутри обработчика нажатия кнопки:

```
Button button = new Button("Получить текущее значение");
button.setOnAction(event -> {
 System.out.println("value = " + slider.getValue());
});
```

Управлять компонентом позволяют следующие свойства:

- ❑ **value** — содержит текущее значение в диапазоне от `min` до `max` (включительно).

Методы доступа:

```
public final void setValue(double value)
public final double getValue()
public final DoubleProperty valueProperty()
```

- ❑ **valueChanging** — свойство содержит значение `true`, если пользователь в данный момент перемещает ползунок, и `false`, если перемещение ползунка закончилось.

Методы доступа:

```
public final void setValueChanging(boolean value)
public final boolean isValueChanging()
public final BooleanProperty valueChangingProperty()
```

- ❑ **min** — минимальное значение диапазона. Методы доступа:

```
public final void setMin(double value)
public final double getMin()
public final DoubleProperty minProperty()
```

- ❑ **max** — максимальное значение диапазона. Методы доступа:

```
public final void setMax(double value)
public final double getMax()
public final DoubleProperty maxProperty()
```

- ❑ **showTickLabels** — если свойство имеет значение `true`, то будут отображаться числовые значения диапазона около главных рисок. Методы доступа:

```
public final void setShowTickLabels(boolean value)
public final boolean isShowTickLabels()
public final BooleanProperty showTickLabelsProperty()
```

- `showTickMarks` — если свойство имеет значение `true`, то будут отображаться риски. Риски бывают главными (длинными) и вспомогательными (короткими). Методы доступа:

```
public final void setShowTickMarks(boolean value)
public final boolean isShowTickMarks()
public final BooleanProperty showTickMarksProperty()
```

- `majorTickUnit` — задает интервал между главными рисками. Например, если у нас есть диапазон от 0 до 100 и мы укажем значение 25, то на шкале отобразятся риски с числовыми значениями: 0, 25, 50, 75 и 100. Методы доступа:

```
public final void setMajorTickUnit(double value)
public final double getMajorTickUnit()
public final DoubleProperty majorTickUnitProperty()
```

- `minorTickCount` — задает количество вспомогательных рисков между главными рисками. Методы доступа:

```
public final void setMinorTickCount(int value)
public final int getMinorTickCount()
public final IntegerProperty minorTickCountProperty()
```

- `snapToTicks` — если свойство имеет значение `true`, то значение ползунка будет выравниваться по рискам. Значение по умолчанию: `false`. Методы доступа:

```
public final void setSnapToTicks(boolean value)
public final boolean isSnapToTicks()
public final BooleanProperty snapToTicksProperty()
```

- `blockIncrement` — задает интервал, на который изменится значение ползунка при нажатии на клавиатуре клавиш со стрелками влево и вправо. Если нажать клавишу `<Home>`, то ползунок переместится в начало шкалы, а если `<End>` — то в конец. Методы доступа:

```
public final void setBlockIncrement(double value)
public final double getBlockIncrement()
public final DoubleProperty blockIncrementProperty()
```

- `labelFormatter` — задает способ преобразования числа в строку для вывода числовых значений рядом с главными рисками. Методы доступа:

```
public final void setLabelFormatter(
 StringConverter<Double> value)
public final StringConverter<Double> getLabelFormatter()
public final ObjectProperty<StringConverter<Double>>
 labelFormatterProperty()
```

- `orientation` — задает ориентацию компонента. В качестве значения параметра `value` можно указать константы: `HORIZONTAL` (горизонтальная ориентация — риски располагаются под шкалой, значение по умолчанию) или `VERTICAL` (вертикальная ориентация — риски располагаются справа от шкалы) из перечисления `javafx.geometry.Orientation`. Методы доступа:

```
public final void setOrientation(Orientation value)
public final Orientation getOrientation()
public final ObjectProperty<Orientation> orientationProperty()
```

Класс `Slider` содержит следующие методы:

- ❑ `adjustValue()` — задает новое значение ползунка. В отличие от метода `setValue()`, метод `adjustValue()` учитывает значение свойства `snapToTicks`. Формат метода:

```
public void adjustValue(double newValue)
```

- ❑ `increment()` — увеличивает значение ползунка на величину свойства `blockIncrement`. Формат метода:

```
public void increment()
```

- ❑ `decrement()` — уменьшает значение ползунка на величину свойства `blockIncrement`. Формат метода:

```
public void decrement()
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH11_10_Slider`, приведенном в каталоге `CH11_components` сопровождающего книгу электронного архива (см. приложение).

## 11.11. Индикаторы хода процесса

С помощью индикатора хода процесса можно информировать пользователя о текущем состоянии выполнения длительной операции. В JavaFX индикаторы реализуются с помощью классов: `ProgressIndicator` и `ProgressBar` (рис. 11.6).

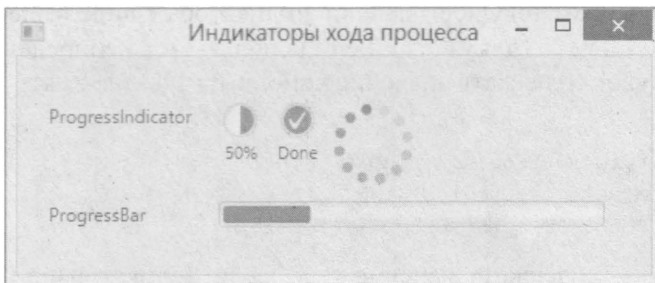


Рис. 11.6. Индикаторы хода процесса

### 11.11.1. Класс *ProgressIndicator*

Класс `ProgressIndicator` реализует индикатор хода процесса в форме круга, внутри которого текущий статус обозначается в виде сегментов. Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.ProgressIndicator;
```

## Иерархия наследования:

Object — Node — Parent — Region — Control — ProgressIndicator

## Конструкторы класса:

```
ProgressIndicator()
ProgressIndicator(double progress)
```

Первый конструктор создает индикатор с неопределенным значением. Внутри индикатора будут постоянно по кругу перемещаться сегменты, показывая ход выполнения процесса с неопределенным количеством шагов. Этот режим можно использовать, когда заранее невозможно разделить задачу на части, и неизвестны сроки ее выполнения. Пример перевода индикатора из неопределенного значения в значение 0 %:

```
ProgressIndicator progress = new ProgressIndicator();
progress.setProgress(0.0);
```

Второй конструктор позволяет указать статус в виде вещественного числа от 0.0 (для 0 %) до 1.0 (для 100 %). Пример установки значения 50 %:

```
ProgressIndicator progress2 = new ProgressIndicator(0.5);
```

Класс `ProgressIndicator` содержит константу `INDETERMINATE_PROGRESS`, с помощью которой можно задать неопределенное значение индикатора. Формат:

```
public static final double INDETERMINATE_PROGRESS
```

## Выведем значение константы:

```
System.out.println(ProgressIndicator.INDETERMINATE_PROGRESS);
// -1.0
```

## Управлять компонентом позволяют следующие свойства:

- `progress` — содержит текущее значение индикатора в виде вещественного числа от 0.0 (для 0 %) до 1.0 (для 100 %). Если используется неопределенное значение, то свойство будет содержать значение константы `INDETERMINATE_PROGRESS`. Методы доступа:

```
public final void setProgress(double value)
public final double getProgress()
public final DoubleProperty progressProperty()
```

- `indeterminate` — содержит значение `true`, если индикатор не имеет диапазона (используется неопределенное значение). Методы доступа:

```
public final boolean isIndeterminate()
public final ReadOnlyBooleanProperty indeterminateProperty()
```

## ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH11_11_ProgressBar`, приведенном в каталоге `CH11_components` сопровождающего книгу электронного архива (см. приложение).

## 11.11.2. Класс *ProgressBar*

Класс `ProgressBar` реализует индикатор хода процесса в виде горизонтальной полосы. Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.ProgressBar;
```

Иерархия наследования:

Object — Node — Parent — Region — Control — ProgressIndicator — ProgressBar

Конструкторы класса:

```
ProgressBar()
ProgressBar(double progress)
```

В параметре `progress` указывается статус в виде вещественного числа от 0.0 (для 0 %) до 1.0 (для 100 %). Пример установки значения 25 % и неопределенного значения:

```
ProgressBar p = new ProgressBar();
p.setProgress(0.25);
ProgressBar p2 = new ProgressBar(
 ProgressIndicator.INDETERMINATE_PROGRESS);
```

Класс `ProgressBar` наследует все методы из класса `ProgressIndicator` и других базовых классов и не добавляет никаких общедоступных методов.

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH11_11_ProgressBar`, приведенном в каталоге `CH11_components` сопровождающего книгу электронного архива (см. приложение).

## 11.11.3. Класс *Task<V>*:

### задача для выполнения в отдельном потоке

Как вы уже знаете, длительные операции нельзя выполнять в потоке диспетчера задач (в потоке с названием `JavaFX Application Thread`). В этом случае окно перестанет реагировать на действия пользователя, и пользователь подумает, что приложение «зависло» и захочет его закрыть. Таким образом, длительные операции мы должны выносить в отдельные потоки.

Помните, что из отдельных потоков нельзя изменять свойства компонентов напрямую. Если это необходимо, то следует сформировать задачу (интерфейс `Runnable`) и передать ее статическому методу `runLater()` из класса `Platform`. Формат метода:

```
public static void runLater(Runnable runnable)
```

Пример:

```
Platform.runLater(() -> {
 label.setText("Новый текст надписи");
});
```

Библиотека JavaFX содержит абстрактный класс `Task<V>`, позволяющий описать однократную задачу, которую мы можем выполнить в отдельном потоке. Класс реализует интерфейсы: `Future<V>` и `RunnableFuture<V>`, а также наследует класс `FutureTask<V>`, следовательно, мы можем получить результат выполнения этой задачи. Давайте рассмотрим этот класс, а заодно и пример использования индикатора хода процесса.

**Инструкция импорта класса `Task<V>`:**

```
// Модуль javafx.graphics
import javafx.concurrent.Task;
```

**Иерархия наследования:**

`Object` — `FutureTask<V>` — `Task<V>`

**Класс `Task<V>` реализует следующие интерфейсы:**

`Runnable`, `Future<V>`, `RunnableFuture<V>`, `Worker<V>`, `EventTarget`

**Мы должны наследовать абстрактный класс `Task<V>` и реализовать метод `call()`.  
Формат метода:**

```
protected abstract V call() throws Exception
```

Метод `call()` будет выполнен в отдельном потоке. Внутри этого метода произведем решение задачи и возвращаем результат. При успешном выполнении задачи будет вызван метод `succeeded()`. Формат метода:

```
protected void succeeded()
```

**Пример реализации через анонимный класс и запуска задачи на выполнение в отдельном потоке-демо:**

```
Task<Integer> task = new Task<Integer>() {
 @Override
 protected Integer call() throws Exception {
 System.out.println(Thread.currentThread().getName());
 int result = 50;
 return result;
 }
 @Override
 protected void succeeded() {
 super.succeeded();
 System.out.println("result = " + getValue());
 }
};

Thread t = new Thread(task);
t.setDaemon(true);
t.start();
```

В этом примере в качестве типа возвращаемого значения мы указали класс `Integer`. Если наша задача просто выполняется и не возвращает никакого значения, то нужно указать тип `Void` и вернуть значение `null`. Пример:

```
Task<Void> task = new Task<Void>() {
 @Override
 protected Void call() throws Exception {
 System.out.println(Thread.currentThread().getName());
 return null;
 }
};
```

Если устанавливается флаг прерывания (например, с помощью метода `cancel()`), то вызывается метод `cancelled()`. Проверить состояние флага позволяет метод `isCancelled()`. **Форматы методов:**

```
public final boolean cancel()
public boolean cancel(boolean mayInterruptIfRunning)
public boolean isCancelled()
protected void cancelled()
```

**Пример переопределения метода `cancelled()` внутри класса:**

```
@Override
protected void cancelled() {
 super.cancelled();
 System.out.println("Установлен флаг прерывания!");
}
```

Если в процессе выполнения задачи возникла ошибка, то вызывается метод `failed()`. **Формат метода:**

```
protected void failed()
```

**Пример переопределения метода `failed()` внутри класса:**

```
@Override
protected void failed() {
 super.failed();
 System.out.println("Ошибка!");
}
```

В классе `Task<V>` существует свойство `progress`, которое содержит значение для индикатора хода выполнения процесса. Методы доступа:

```
public final double getProgress()
public final ReadOnlyDoubleProperty progressProperty()
```

Изменить значение этого свойства позволяет метод `updateProgress()`, который мы можем вызвать внутри метода `call()`. **Форматы метода:**

```
protected void updateProgress(long workDone, long max)
protected void updateProgress(double workDone, double max)
```

В параметре `workDone` указывается текущий статус процесса, а в параметре `max` — конечный статус. Эти значения будут переведены в диапазон значений индикатора хода выполнения процесса.



В качестве примера создадим три компонента: индикатор хода процесса, надпись и кнопку для запуска задачи на выполнение:

```
ProgressIndicator progress = new ProgressIndicator();
progress.setProgress(0.0);
progress.setVisible(false);
```

```
Label label = new Label("");
```

```
Button button = new Button("Запустить процесс");
```

Внутри обработчика нажатия кнопки создадим задачу и запустим ее на выполнение:

```
button.setOnAction(event -> {
 button.setDisable(true);
 Task<Integer> task = new Task<Integer>() {
 @Override
 protected Integer call() throws Exception {
 int result = 50;
 for (int i = 0; i < 101; i += 5) {
 if (isCancelled()) break;
 updateMessage("i = " + i);
 updateProgress(i, 100);
 Thread.sleep(500);
 }
 }
 Platform.runLater(() -> {
 PauseTransition pause = new PauseTransition(
 Duration.millis(200.0));
 pause.setOnFinished(e -> {
 progress.progressProperty().unbind();
 label.textProperty().unbind();
 progress.visibleProperty().unbind();
 button.setDisable(false);
 });
 pause.play();
 });
 return result;
 }
 @Override
 protected void succeeded() {
 super.succeeded();
 System.out.println("result = " + getValue());
 updateMessage("Задача успешно завершена!");
 }
});
progress.progressProperty().bind(task.progressProperty());
progress.visibleProperty().bind(task.runningProperty());
label.textProperty().bind(task.messageProperty());
```

```
Thread t = new Thread(task);
t.setDaemon(true);
t.start();
});
```

Итак, во второй строке мы делаем кнопку недоступной. Так как процесс у нас длительный, то пользователь не должен нажать кнопку еще раз только лишь потому, что ему надоело ждать, или просто случайно. Далее реализуем метод `call()` и внутри него описываем алгоритм обработки задачи. В нашем примере это всего лишь цикл, внутри которого мы «засыпаем» на 500 миллисекунд, имитируя выполнение задачи. Внутри цикла мы проверяем статус установки флага прерывания с помощью следующего кода:

```
if (isCancelled()) break;
```

Если флаг установлен, то выходим из цикла. Если нет, то обновляем текст сообщения с помощью метода `updateMessage()` и значение для индикатора хода выполнения процесса с помощью метода `updateProgress()`:

```
updateMessage("i = " + i);
updateProgress(i, 100);
```

Так как мы связали свойство `message` со свойством `text` надписи, а свойство `progress` со свойством `progress` индикатора, то изменения будут автоматически отображены в окне, и пользователь увидит, на какой стадии находится процесс выполнения задачи. Связывание выполнялось с помощью следующих строк кода:

```
progress.progressProperty().bind(task.progressProperty());
progress.visibleProperty().bind(task.runningProperty());
label.textProperty().bind(task.messageProperty());
```

Здесь мы еще связали свойство `running` со свойством `visible` индикатора. Пока выполняется задача, индикатор будет отображаться в окне. Как только выполнение закончится, индикатор станет невидимым.

После завершения выполнения процесса формируем задачу и передаем ее методу `runLater()`. Внутри этой задачи мы добавляем задержку с помощью класса `PauseTransition`, чтобы изменения вступили в силу. Когда срок паузы истечет, отменяем связывание для всех свойств. Класс `Task<V>` описывает задачу, которая может быть выполнена только один раз, поэтому это связывание больше не пригодится. Затем делаем кнопку снова доступной. При повторном нажатии кнопки будет создана новая задача, и весь процесс повторится. Вот таким образом мы можем управлять индикатором хода выполнения процесса из отдельного потока, в котором выполняется длительная операция.

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH11_11_ProgressBar`, приведенном в каталоге `CH11_components` сопровождающего книгу электронного архива (см. приложение).

## 11.11.4. Интерфейс *Worker<V>*

Класс `Task<V>` содержит следующие свойства, методы доступа к которым описаны в интерфейсе `Worker<V>`:

- ❑ `state` — содержит текущий статус задачи в виде следующих констант из перечисления `Worker.State`: `READY` (ожидает запуска), `SCHEDULED` (запланирован запуск, но задача еще не запущена на выполнение), `RUNNING` (задача выполняется), `SUCCEEDED` (задача успешно выполнена), `FAILED` (ошибка при выполнении задачи) и `CANCELLED` (задача отменена). Методы доступа:

```
import javafx.concurrent.Worker;
public final Worker.State getState()
public final ReadOnlyObjectProperty<Worker.State>
 stateProperty()
```

- ❑ `running` — содержит значение `true`, если статус `SCHEDULED` или `RUNNING`. Мы можем связать это свойство со свойством видимости индикатора хода процесса. Методы доступа:

```
public final boolean isRunning()
public final ReadOnlyBooleanProperty runningProperty()
```

- ❑ `value` — содержит результат выполнения задачи при статусе `SUCCEEDED`. Методы доступа:

```
public final V getValue()
public final ReadOnlyObjectProperty<V> valueProperty()
```

Пример:

```
if (task.getState() == Worker.State.SUCCEEDED) {
 System.out.println("value = " + task.getValue());
}
```

- ❑ `workDone` — текущее значение выполнения задачи от `-1.0` до значения свойства `totalWork`. Методы доступа:

```
public final double getWorkDone()
public final ReadOnlyDoubleProperty workDoneProperty()
```

- ❑ `totalWork` — максимальное значение выполнения задачи. Методы доступа:

```
public final double getTotalWork()
public final ReadOnlyDoubleProperty totalWorkProperty()
```

- ❑ `progress` — содержит значение для индикатора хода выполнения процесса. Методы доступа:

```
public final double getProgress()
public final ReadOnlyDoubleProperty progressProperty()
```

Изменить значение этого свойства позволяет метод `updateProgress()` из класса `Task<V>`, который мы можем вызвать внутри метода `call()`. Форматы метода:

```
protected void updateProgress(long workDone, long max)
protected void updateProgress(double workDone, double max)
```

В параметре `workDone` указывается текущий статус процесса, а в параметре `max` — конечный статус. Эти значения будут переведены в диапазон значений индикатора хода выполнения процесса;

- ❑ `message` — текстовое описание статуса выполнения задачи. Это свойство мы можем связать со свойством `text` надписи. Методы доступа:

```
public final String getMessage()
public final ReadOnlyStringProperty messageProperty()
```

Изменить значение этого свойства позволяет метод `updateMessage()` из класса `Task<V>`, который мы можем вызвать внутри метода `call()` или внутри других методов. Формат метода:

```
protected void updateMessage(String message)
```

- ❑ `title` — содержит заголовок. Методы доступа:

```
public final String getTitle()
public final ReadOnlyStringProperty titleProperty()
```

Изменить значение этого свойства позволяет метод `updateTitle()` из класса `Task<V>`, который мы можем вызвать внутри метода `call()`. Формат метода:

```
protected void updateTitle(String title)
```

- ❑ `exception` — содержит объект исключения, если при выполнении произошла ошибка. Методы доступа:

```
public final Throwable getException()
public final ReadOnlyObjectProperty<Throwable>
 exceptionProperty()
```

Интерфейс `Worker<V>` содержит также объявление метода `cancel()`, который позволяет установить флаг отмены выполнения задачи. Формат метода в классе `Task<V>`:

```
public final boolean cancel()
```

Установку этого флага можно проверить с помощью метода `isCancelled()`. Формат метода:

```
public boolean isCancelled()
```

## 11.11.5. Обработка событий выполнения задачи

События процесса выполнения задачи описываются с помощью класса `WorkerStateEvent`. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.concurrent.WorkerStateEvent;
```

Иерархия наследования:

```
Object — EventObject — Event — WorkerStateEvent
```

В классе `WorkerStateEvent` определены следующие типы событий:

- ANY — любое из приведенных далее событий. Формат:

```
public static final EventType<WorkerStateEvent> ANY
```

Назначить обработчики событий процесса выполнения задачи мы можем уже знакомыми нам методами: `addEventHandler()` и `addEventFilter()` из класса `Task<V>`, а удалить обработчик — с помощью методов `removeEventHandler()` и `removeEventFilter()` соответственно. Пример назначения обработчика:

```
task.addEventHandler(WorkerStateEvent.ANY, e -> {
 System.out.println("ANY - " + e.getEventType());
});
```

- `WORKER_STATE_READY` — событие установки статуса `READY` (задача ожидает запуска). Формат:

```
public static final EventType<WorkerStateEvent>
 WORKER_STATE_READY
```

- `WORKER_STATE_SCHEDULED` — событие установки статуса `SCHEDULED` (запланирован запуск, но задача еще не запущена на выполнение). Формат:

```
public static final EventType<WorkerStateEvent>
 WORKER_STATE_SCHEDULED
```

Управлять обработчиком события позволяет также свойство `onScheduled` из класса `Task<V>`. Методы доступа:

```
public final void setOnScheduled(
 EventHandler<WorkerStateEvent> value)
public final EventHandler<WorkerStateEvent> getOnScheduled()
public final ObjectProperty<EventHandler<WorkerStateEvent>>
 onScheduledProperty()
```

- `WORKER_STATE_RUNNING` — событие установки статуса `RUNNING` (задача выполняется). Формат:

```
public static final EventType<WorkerStateEvent>
 WORKER_STATE_RUNNING
```

Управлять обработчиком события позволяет также свойство `onRunning` из класса `Task<V>`. Методы доступа:

```
public final void setOnRunning(
 EventHandler<WorkerStateEvent> value)
public final EventHandler<WorkerStateEvent> getOnRunning()
public final ObjectProperty<EventHandler<WorkerStateEvent>>
 onRunningProperty()
```

- `WORKER_STATE_SUCCEEDED` — событие установки статуса `SUCCEEDED` (задача успешно выполнена). Формат:

```
public static final EventType<WorkerStateEvent>
 WORKER_STATE_SUCCEEDED
```

Управлять обработчиком события позволяет также свойство `onSucceeded` из класса `Task<V>`. Методы доступа:

```
public final void setOnSucceeded(
 EventHandler<WorkerStateEvent> value)
public final EventHandler<WorkerStateEvent> getOnSucceeded()
public final ObjectProperty<EventHandler<WorkerStateEvent>>
 onSuccessedProperty()
```

Пример назначения обработчика и получения результата внутри обработчика:

```
task.setOnSucceeded(e -> {
 label.textProperty().unbind();
 progress.visibleProperty().unbind();
 progress.progressProperty().unbind();
 label.setText("Задача успешно завершена! Результат: " +
 e.getSource().getValue());
 task.setOnSucceeded(null);
 button.setDisable(false);
});
```

Обработчики вызываются в потоке диспетчера задач, поэтому мы можем свободно изменять свойства компонентов. Через метод `getSource()` объекта события доступен объект-источник события, который реализует интерфейс `Worker<V>`. Формат метода:

```
public Worker getSource()
```

- ❑ `WORKER_STATE_FAILED` — событие установки статуса `FAILED` (ошибка при выполнении задачи). Формат:

```
public static final EventType<WorkerStateEvent>
 WORKER_STATE_FAILED
```

Управлять обработчиком события позволяет также свойство `onFailed` из класса `Task<V>`. Методы доступа:

```
public final void setOnFailed(
 EventHandler<WorkerStateEvent> value)
public final EventHandler<WorkerStateEvent> getOnFailed()
public final ObjectProperty<EventHandler<WorkerStateEvent>>
 onFailedProperty()
```

- ❑ `WORKER_STATE_CANCELLED` — событие установки статуса `CANCELLED` (задача отменена). Формат:

```
public static final EventType<WorkerStateEvent>
 WORKER_STATE_CANCELLED
```

Управлять обработчиком события позволяет также свойство `onCancelled` из класса `Task<V>`. Методы доступа:

```
public final void setOnCancelled(
 EventHandler<WorkerStateEvent> value)
```

```
public final EventHandler<WorkerStateEvent> getOnCancelled()
public final ObjectProperty<EventHandler<WorkerStateEvent>>
 onCancelledProperty()
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH11_11_ProgressBar`, приведенном в каталоге `CH11_components` сопровождающего книгу электронного архива (см. приложение).

## 11.11.6. Класс `Service<V>`

Как вы уже знаете, класс `Task<V>` описывает одноразовую задачу. Чтобы запустить задачу повторно, необходимо создать новый объект. Абстрактный класс `Service<V>` позволяет автоматизировать процесс создания объектов задач и запуска их на выполнение в отдельном потоке или с помощью пула потоков. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.concurrent.Service;
```

Мы должны наследовать класс `Service<V>` и реализовать абстрактный метод `createTask()`, внутри которого нужно создать и вернуть объект класса `Task<V>`. Формат метода:

```
protected abstract Task<V> createTask()
```

Давайте создадим класс `MyService` и реализуем этот метод:

```
class MyService extends Service<Integer> {
 protected Task<Integer> createTask() {
 return new Task<Integer>() {
 protected Integer call() throws Exception {
 int result = 50;
 for (int i = 0; i < 101; i += 5) {
 if (isCancelled()) break;
 updateMessage("i = " + i);
 updateProgress(i, 100);
 Thread.sleep(500);
 }
 return result;
 }
 };
 }
}
```

Класс `Service<V>` реализует все методы из интерфейса `Worker<V>` (см. разд. 11.11.4). Обработать события можно с помощью тех же самых методов, которые мы рассматривали в разд. 11.11.5. Класс `Service<V>` дополнительно содержит свойство `onReady`, с помощью которого можно назначить обработчик события `WORKER_STATE_READY` (событие установки статуса `READY`: задача ожидает запуска). Методы доступа:

```
public final void setOnReady(EventHandler<WorkerStateEvent> value)
public final EventHandler<WorkerStateEvent> getOnReady()
public final ObjectProperty<EventHandler<WorkerStateEvent>>
 onReadyProperty()
```

**Пример создания объекта сервиса, назначения обработчика для статуса SUCCEEDED (задача успешно выполнена) и получения результата выполнения задачи:**

```
MyService service = new MyService();
// Связывание свойств с индикатором и надписью
progress.progressProperty().bind(service.progressProperty());
progress.visibleProperty().bind(service.runningProperty());
label.textProperty().bind(service.messageProperty());
// Обработка статуса SUCCEEDED
service.setOnSucceeded(e -> {
 label.textProperty().unbind();
 label.setText("Задача успешно завершена! Результат: " +
 e.getSource().getValue());
 button.setDisable(false);
 service.reset(); // Создание новой задачи
});
```

Управлять сервисом позволяют следующие методы:

- ❑ **start()** — запускает задачу на выполнение. Сервис должен иметь статус **READY** (задача ожидает запуска). По умолчанию задача запускается на выполнение в потоке-демон — учитывайте это, если планируете работать с ресурсами (например, с файлами). Поток-демон может быть прерван на любой стадии выполнения после завершения всех основных потоков, поэтому работать с ресурсами в таком потоке не следует. Формат метода:

```
public void start()
```

**Пример запуска задачи на выполнение после нажатия пользователем кнопки:**

```
button.setOnAction(event -> {
 if (service.getState() == Worker.State.READY) {
 button.setDisable(true);
 service.start();
 }
 else {
 System.out.println("Статус не READY");
 }
});
```

- ❑ **reset()** — сбрасывает текущую задачу и создает новую. Сервис должен иметь статус: **SUCCEEDED**, **FAILED**, **CANCELLED** или **READY**. Формат метода:

```
public void reset()
```

- ❑ **restart()** — отменяет текущую задачу и перезапускает сервис. Формат метода:

```
public void restart()
```



Итак, по умолчанию задача запускается в потоке-демоне, что не очень хорошо, если мы работаем с ресурсами. Кроме того, запуск потока — это дорогостоящая операция. Если мы запускаем задачи часто, то лучше воспользоваться пулами потоков. При работе с пулом потоки запускаются один раз (или при необходимости) и находятся в режиме ожидания задач постоянно или какое-то время (например, 60 секунд). Указать ссылку на пул потоков позволяет свойство `executor` из класса `Service<V>`. Методы доступа:

```
public final void setExecutor(Executor value)
public final Executor getExecutor()
public final ObjectProperty<Executor> executorProperty()
```

Создадим пул потоков из одного потока и добавим ссылку на него в наш сервис:

```
ExecutorService pool = Executors.newFixedThreadPool(1);
service.setExecutor(pool);
```

Чтобы остановить потоки в пуле, не забудьте вызвать метод `shutdown()` или `shutdownNow()` (например, при закрытии окна), иначе после завершения работы приложения потоки так и будут работать, т. к. они не являются демонами. Пример остановки потоков внутри обработчика закрытия окна:

```
window.setOnCloseRequest(event -> {
 service.cancel();
 pool.shutdown();
});
```

Перезапуск задач в классе `Service<V>` выполняется вручную. Если необходимо, чтобы перезапуск выполнялся автоматически через какой-либо промежуток времени, то следует воспользоваться наследником класса `Service<V>` классом `ScheduledService<V>`. Описание этого класса смотрите в документации.

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH11_11_ProgressBar`, приведенном в каталоге `CH11_components` сопровождающего книгу электронного архива (см. *приложение*).

## 11.12. Класс *ScrollBar*: полоса прокрутки

Класс `ScrollBar` реализует горизонтальную (по умолчанию) или вертикальную полосу прокрутки. Изменить значение можно с помощью нажатия кнопок, расположенных по краям полосы, щелчка мышью на полосе, путем перемещения ползунка, а также поворотом колесика мыши. Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.ScrollBar;
```

Иерархия наследования:

```
Object — Node — Parent — Region — Control — ScrollBar
```

## Конструктор класса:

```
ScrollBar()
```

## Пример создания горизонтальной полосы прокрутки и обработки изменения значения:

```
ScrollBar scrollBar = new ScrollBar();
scrollBar.setPrefHeight(20.0);
scrollBar.setMin(0.0);
scrollBar.setMax(100.0);
scrollBar.setValue(0.0);
scrollBar.setBlockIncrement(5.0);

scrollBar.valueProperty().addListener((obj, oldValue, newValue) -> {
 System.out.println("scrollBar value = " + newValue);
});
```

## Пример получения значения внутри обработчика нажатия кнопки:

```
Button button = new Button("Получить текущее значение");
button.setOnAction(event -> {
 System.out.println("value = " + scrollBar.getValue());
});
```

### **ПРИМЕЧАНИЕ**

Полоса прокрутки редко используется отдельно. Гораздо удобнее воспользоваться областью с полосами прокрутки, которую реализует класс `ScrollPane` (см. *разд. 11.13*).

## Управлять компонентом позволяют следующие свойства:

- ❑ **value** — содержит текущее значение в диапазоне от `min` до `max` (включительно).

### Методы доступа:

```
public final void setValue(double value)
public final double getValue()
public final DoubleProperty valueProperty()
```

- ❑ **min** — минимальное значение диапазона. Значение по умолчанию: 0.0. Методы доступа:

```
public final void setMin(double value)
public final double getMin()
public final DoubleProperty minProperty()
```

- ❑ **max** — максимальное значение диапазона. Значение по умолчанию: 100.0. Методы доступа:

```
public final void setMax(double value)
public final double getMax()
public final DoubleProperty maxProperty()
```

- ❑ **blockIncrement** — задает интервал, на который изменится значение при щелчке мышью на полосе. Методы доступа:

```
public final void setBlockIncrement(double value)
public final double getBlockIncrement()
public final DoubleProperty blockIncrementProperty()
```

- ❑ **unitIncrement** — задает интервал, на который изменится значение при нажатии кнопок со стрелками, повороте колесика мыши, а также при вызове методов `increment()` и `decrement()`. Методы доступа:

```
public final void setUnitIncrement(double value)
public final double getUnitIncrement()
public final DoubleProperty unitIncrementProperty()
```

- ❑ **orientation** — задает ориентацию компонента. В качестве значения параметра `value` можно указать константы: `HORIZONTAL` (горизонтальная ориентация, значение по умолчанию) или `VERTICAL` (вертикальная ориентация) из перечисления `javafx.geometry.Orientation`. Методы доступа:

```
public final void setOrientation(Orientation value)
public final Orientation getOrientation()
public final ObjectProperty<Orientation> orientationProperty()
```

- ❑ **visibleAmount** — задает длину ползунка. Методы доступа:

```
public final void setVisibleAmount(double value)
public final double getVisibleAmount()
public final DoubleProperty visibleAmountProperty()
```

Класс `ScrollBar` содержит следующие методы:

- ❑ **adjustValue()** — изменяет значение свойства `value` на величину свойства `blockIncrement` до указанного значения `position` (вещественное число от 0.0 до 1.0, означающее процентное соотношение). Например, если диапазон от 0.0 до 100.0 и текущее значение 25.0, то значение `position`, равное 0.5, приведет к увеличению значения `value` на `blockIncrement`, но не более значения 50.0. Если текущее значение 75.0, то значение `position`, равное 0.5, приведет к уменьшению значения `value` на `blockIncrement`, но не менее значения 50.0. Формат метода:

```
public void adjustValue(double position)
```

- ❑ **increment()** — увеличивает значение на величину свойства `unitIncrement`. Формат метода:

```
public void increment()
```

- ❑ **decrement()** — уменьшает значение на величину свойства `unitIncrement`. Формат метода:

```
public void decrement()
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH11_12_ScrollBar`, приведенном в каталоге `CH11_components` сопровождающего книгу электронного архива (см. приложение).

## 11.13. Класс *ScrollPane*: область с полосами прокрутки

Класс `ScrollPane` реализует область с полосами прокрутки. Если содержимое не помещается в размеры области, то автоматически отображаются полосы прокрутки. Изменение положения ползунков с помощью мыши автоматически приводит к прокрутке содержимого области. Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.ScrollPane;
```

Иерархия наследования:

`Object` — `Node` — `Parent` — `Region` — `Control` — `ScrollPane`

Конструкторы класса:

```
ScrollPane()
ScrollPane(Node content)
```

Пример:

```
Rectangle rect = new Rectangle(600.0, 600.0);
rect.relocate(10.0, 10.0);
rect.setFill(Color.GREEN);
```

```
Pane pane = new Pane(rect);
pane.setPrefSize(620.0, 620.0);
```

```
ScrollPane scrollPane = new ScrollPane();
scrollPane.setContent(pane);
scrollPane.setPrefViewportWidth(400.0);
scrollPane.setPrefViewportHeight(300.0);
```

Управлять компонентом позволяют следующие свойства:

- ❑ `content` — задает узел, который будет отображаться внутри области. Методы доступа:

```
public final void setContent(Node value)
public final Node getContent()
public final ObjectProperty<Node> contentProperty()
```

- ❑ `hvalue` — текущее значение горизонтальной полосы прокрутки. Методы доступа:

```
public final void setHvalue(double value)
public final double getHvalue()
public final DoubleProperty hvalueProperty()
```

- ❑ `hmin` — минимальное значение горизонтальной полосы прокрутки. Значение по умолчанию: 0.0. Методы доступа:

```
public final void setHmin(double value)
public final double getHmin()
public final DoubleProperty hminProperty()
```

- ❑ **hmax** — максимальное значение горизонтальной полосы прокрутки. Значение по умолчанию: 1.0. Методы доступа:

```
public final void setHmax(double value)
public final double getHmax()
public final DoubleProperty hmaxProperty()
```

- ❑ **hbarPolicy** — задает режим отображения горизонтальной полосы прокрутки. Методы доступа:

```
public final void setHbarPolicy(
 ScrollPane.ScrollBarPolicy value)
public final ScrollPane.ScrollBarPolicy getHbarPolicy()
public final ObjectProperty<ScrollPane.ScrollBarPolicy>
 hbarPolicyProperty()
```

В параметре `value` можно указать следующие константы из перечисления `ScrollPane.ScrollBarPolicy`:

```
import javafx.scene.control.ScrollPane.ScrollBarPolicy;
```

- **AS\_NEEDED** — полоса прокрутки отображается по мере необходимости (значение по умолчанию);
- **ALWAYS** — полоса прокрутки отображается всегда;
- **NEVER** — полоса прокрутки не отображается никогда.

**Формат:**

```
public static final ScrollPane.ScrollBarPolicy AS_NEEDED
public static final ScrollPane.ScrollBarPolicy ALWAYS
public static final ScrollPane.ScrollBarPolicy NEVER
```

- ❑ **vvalue** — текущее значение вертикальной полосы прокрутки. Методы доступа:

```
public final void setVvalue(double value)
public final double getVvalue()
public final DoubleProperty vvalueProperty()
```

- ❑ **vmin** — минимальное значение вертикальной полосы прокрутки. Значение по умолчанию: 0.0. Методы доступа:

```
public final void setVmin(double value)
public final double getVmin()
public final DoubleProperty vminProperty()
```

- ❑ **vmax** — максимальное значение вертикальной полосы прокрутки. Значение по умолчанию: 1.0. Методы доступа:

```
public final void setVmax(double value)
public final double getVmax()
public final DoubleProperty vmaxProperty()
```

- ❑ **vbarPolicy** — задает режим отображения вертикальной полосы прокрутки (**AS\_NEEDED**, **ALWAYS** или **NEVER**). Значение по умолчанию: **AS\_NEEDED**. Методы доступа:

```
public final void setVbarPolicy(
 ScrollPane.ScrollBarPolicy value)
public final ScrollPane.ScrollBarPolicy getVbarPolicy()
public final ObjectProperty<ScrollPane.ScrollBarPolicy>
 vbarPolicyProperty()
```

- ❑ **minViewportWidth** — минимальная ширина области. Методы доступа:

```
public final void setMinViewportWidth(double value)
public final double getMinViewportWidth()
public final DoubleProperty minViewportWidthProperty()
```

- ❑ **minViewportHeight** — минимальная высота области. Методы доступа:

```
public final void setMinViewportHeight(double value)
public final double getMinViewportHeight()
public final DoubleProperty minViewportHeightProperty()
```

- ❑ **prefViewportWidth** — предпочтительная ширина области. Методы доступа:

```
public final void setPrefViewportWidth(double value)
public final double getPrefViewportWidth()
public final DoubleProperty prefViewportWidthProperty()
```

- ❑ **prefViewportHeight** — предпочтительная высота области. Методы доступа:

```
public final void setPrefViewportHeight(double value)
public final double getPrefViewportHeight()
public final DoubleProperty prefViewportHeightProperty()
```

- ❑ **viewportBounds** — положение и размеры области. Методы доступа:

```
public final void setViewportBounds(Bounds value)
public final Bounds getViewportBounds()
public final ObjectProperty<Bounds> viewportBoundsProperty()
```

- ❑ **pannable** — если свойство имеет значение `true`, то содержимое области можно будет перемещать с помощью мыши. Значение по умолчанию: `false`. Методы доступа:

```
public final void setPannable(boolean value)
public final boolean isPannable()
public final BooleanProperty pannableProperty()
```

- ❑ **fitToWidth** — если свойство имеет значение `true`, то узел внутри области будет менять свою ширину одновременно с изменением размера области, при условии, что узел поддерживает изменение размера. Например, контейнер будет заполнять всю доступную ширину области. Значение по умолчанию: `false`. Методы доступа:

```
public final void setFitToWidth(boolean value)
public final boolean isFitToWidth()
public final BooleanProperty fitToWidthProperty()
```

- ❑ `fitToHeight` — если свойство имеет значение `true`, то узел внутри области будет менять свою высоту одновременно с изменением размера области, при условии, что узел поддерживает изменение размера. Значение по умолчанию: `false`.  
Методы доступа:

```
public final void setFitToHeight(boolean value)
public final boolean isFitToHeight()
public final BooleanProperty fitToHeightProperty()
```

При работе с компонентом `ScrollPane` бывает нужно получить доступ к полосам прокрутки — например, чтобы изменить толщину полос прокрутки. Чтобы получить ссылку на полосу прокрутки, можно воспользоваться методом `lookupAll()` из класса `Node`. Достаточно выполнить поиск по стилевому классу `scroll-bar`. Давайте получим ссылки и изменим ширину вертикальной полосы прокрутки и высоту горизонтальной полосы:

```
// window.show();
Set<Node> set = scrollPane.lookupAll(".scroll-bar");
for (Node node: set) {
 if (node instanceof ScrollBar) {
 ScrollBar sb = (ScrollBar) node;
 if (sb.getOrientation() == Orientation.VERTICAL) {
 sb.setPrefWidth(20.0);
 }
 else {
 sb.setPrefHeight(20.0);
 }
 }
}
```

Обратите внимание: этот код должен быть расположен после инструкции отображения окна, иначе поиск полос прокрутки закончится неудачей, и ссылки мы не получим.

Аналогичную операцию можно выполнить с помощью стилей (см. *разд. 19.12*):

```
ScrollPane .scroll-bar:horizontal { -fx-pref-height: 20px; }
ScrollPane .scroll-bar:vertical { -fx-pref-width: 20px; }
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH11_13_ScrollPane`, приведенном в каталоге `CH11_components` сопровождающего книгу электронного архива (см. *приложение*).

## 11.14. Класс *ToolBar*: панель инструментов

Панели инструментов предназначены для горизонтального или вертикального выравнивания кнопок с часто используемыми командами, а также любых других узлов. Очень часто кнопки на панели инструментов дублируют пункты главного

меню приложения. Создать панель инструментов позволяет класс `ToolBar`. Если компоненты не помещаются на панели, то отобразится кнопка, с помощью которой можно выбрать скрытые компоненты (рис. 11.7). Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.ToolBar;
```

**Иерархия наследования:**

Object — Node — Parent — Region — Control — ToolBar

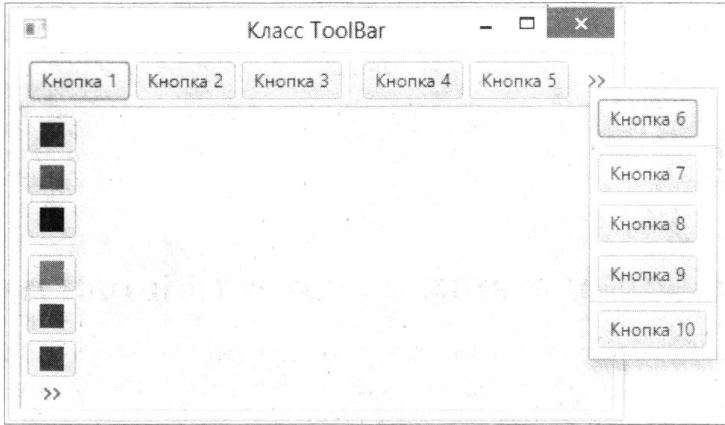


Рис. 11.7. Панели инструментов с кнопками

**Конструкторы класса:**

```
ToolBar()
ToolBar(Node... items)
```

Первый конструктор создает пустую панель инструментов, а второй — позволяет сразу добавить компоненты на панель. Чтобы добавить компоненты после создания объекта, следует воспользоваться методом `getItems()`. Формат метода:

```
public final ObservableList<Node> getItems()
```

**Пример создания панели инструментов с горизонтальной ориентацией и добавления десяти кнопок на панель:**

```
ToolBar toolBar = new ToolBar();
toolBar.setOrientation(Orientation.HORIZONTAL);
for (int i = 1; i < 11; i++) {
 Button b = new Button("Кнопка " + i);
 toolBar.getItems().add(b);
}
```

Для прикрепления панели инструментов к какой-либо боковой части окна удобно воспользоваться контейнером `BorderPane`:

```
BorderPane pane = new BorderPane();
pane.setTop(toolBar);
```



По умолчанию создается панель с горизонтальной ориентацией. Чтобы задать вертикальную ориентацию, следует присвоить константу `VERTICAL` из перечисления `Orientation` свойству `orientation`. Методы доступа:

```
import javafx.geometry.Orientation;
public final void setOrientation(Orientation value)
public final Orientation getOrientation()
public final ObjectProperty<Orientation> orientationProperty()
```

Пример:

```
toolBar.setOrientation(Orientation.VERTICAL);
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH11_14_ToolBar`, приведенном в каталоге `CH11_components` сопровождающего книгу электронного архива (см. приложение).

## 11.15. Класс *Separator*: разделительная линия

Класс `Separator` реализует горизонтальную или вертикальную разделительную линию, с помощью которой можно визуальнo отделить одну группу компонентов от другой. Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.Separator;
```

Иерархия наследования:

`Object` — `Node` — `Parent` — `Region` — `Control` — `Separator`

Конструкторы класса:

```
Separator()
Separator(Orientation orientation)
```

Первый конструктор создает горизонтальную разделительную линию, а второй — позволяет указать ориентацию линии. Пример создания вертикальной линии:

```
Separator separator = new Separator(Orientation.VERTICAL);
```

Класс `Separator` содержит следующие свойства:

- `orientation` — задает ориентацию линии. В качестве значения можно указать константы `HORIZONTAL` (горизонтальная линия, значение по умолчанию) и `VERTICAL` (вертикальная линия) из перечисления `Orientation`. Методы доступа:

```
import javafx.geometry.Orientation;
public final void setOrientation(Orientation value)
public final Orientation getOrientation()
public final ObjectProperty<Orientation> orientationProperty()
```

- ❑ `valignment` — задает вертикальное выравнивание для линии с горизонтальной ориентацией. В качестве значения можно указать константы `TOP`, `BASELINE`, `CENTER` или `BOTTOM` из перечисления `javafx.geometry.VPos`. Методы доступа:

```
public final void setValignment(VPos value)
public final VPos getValignment()
public final ObjectProperty<VPos> valignmentProperty()
```

- ❑ `halignment` — задает горизонтальное выравнивание для линии с вертикальной ориентацией. В качестве значения можно указать константы `LEFT`, `CENTER` или `RIGHT` из перечисления `javafx.geometry.HPos`. Методы доступа:

```
public final void setHalignment(HPos value)
public final HPos getHalignment()
public final ObjectProperty<HPos> halignmentProperty()
```

### **ЭЛЕКТРОННЫЙ АРХИВ**

Полный текст программы содержится в проекте `FX_CH11_15_Separator`, приведенном в каталоге `CH11_components` сопровождающего книгу электронного архива (см. приложение).



## ГЛАВА 12



# Работа с текстом и Интернетом

В этой главе мы рассмотрим различные компоненты, позволяющие работать с текстом, — например, однострочные и многострочные текстовые поля. Напомню, что мы уже рассматривали компонент `Label` (см. *разд. 11.3*), позволяющий отобразить нередактируемый текст в окне, а также фигуру `Text` (см. *разд. 5.18*), которую также можно использовать для вывода текста.

## 12.1. Класс `Spinner<T>`

Класс `Spinner<T>` описывает текстовое поле (рис. 12.1), в котором отображается числовое значение из диапазона или значение из списка. Поле содержит две маленькие кнопки со стрелками, которые позволяют перемещаться по диапазону или списку с помощью щелчков мыши. Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.Spinner;
```

Иерархия наследования:

`Object` — `Node` — `Parent` — `Region` — `Control` — `Spinner`

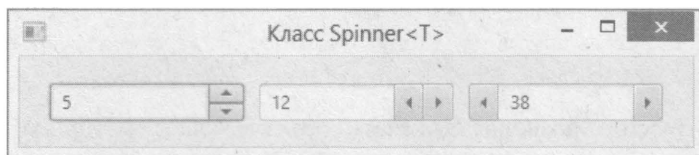


Рис. 12.1. Класс `Spinner<T>`

Конструкторы класса:

```
Spinner()
Spinner(int min, int max, int initialValue)
Spinner(int min, int max, int initialValue, int amountToStepBy)
Spinner(double min, double max, double initialValue)
```

```
Spinner(double min, double max, double initialValue,
 double amountToStepBy)
Spinner(ObservableList<T> items)
Spinner(SpinnerValueFactory<T> valueFactory)
```

Первый конструктор создает пустой объект. Пример создания целочисленного диапазона значений от 0 до 10 и значения по умолчанию 5:

```
// import javafx.scene.control.SpinnerValueFactory;
Spinner<Integer> spinner = new Spinner<Integer>();
spinner.setValueFactory(
 new SpinnerValueFactory.IntegerSpinnerValueFactory(0, 10, 5));
```

Второй и третий конструкторы создают целочисленный диапазон значений от min до max. Параметр `initialValue` задает значение по умолчанию, а параметр `amountToStepBy` — шаг, на который будет изменяться значение при нажатии кнопок со стрелками. Если параметр `amountToStepBy` не задан, то шаг будет равен значению 1. Пример:

```
Spinner<Integer> intSpinner = new Spinner<Integer>(0, 100, 0);
Spinner<Integer> intSpinner2 = new Spinner<Integer>(0, 100, 0, 5);
```

Четвертый и пятый конструкторы создают диапазон вещественных чисел от min до max. Параметр `initialValue` задает значение по умолчанию, а параметр `amountToStepBy` — шаг, на который будет изменяться значение при нажатии кнопок со стрелками. Если параметр `amountToStepBy` не задан, то шаг будет равен значению 1.0. Пример:

```
Spinner<Double> doubleSpinner = new Spinner<Double>(0.0, 10.0, 0.0);
Spinner<Double> doubleSpinner2 = new Spinner<Double>(
 0.0, 10.0, 0.0, 0.1);
```

Шестой конструктор позволяет указать список возможных значений. Нажатие кнопок со стрелками будет приводить к перемещению указателя внутри списка и отображению текущего элемента списка в текстовом поле. Пример:

```
ObservableList<String> list =
 FXCollections.<String>observableArrayList("red", "green", "blue");
Spinner<String> listSpinner = new Spinner<String>(list);
```

Седьмой конструктор позволяет указать объект, класс которого реализует абстрактный класс `SpinnerValueFactory<T>`. Пример создания целочисленного диапазона:

```
Spinner<Integer> intSpinner3 = new Spinner<Integer>(
 new SpinnerValueFactory.IntegerSpinnerValueFactory(0, 10, 1));
```

Можно наследовать класс `SpinnerValueFactory<T>` и реализовать абстрактные методы или воспользоваться следующими готовыми статическими классами:

```
import javafx.scene.control.SpinnerValueFactory;
```

- ❑ `IntegerSpinnerValueFactory` — создает диапазон целочисленных значений. Пример:

```
Spinner<Integer> spinner = new Spinner<Integer>();
spinner.setValueFactory(
 new SpinnerValueFactory.IntegerSpinnerValueFactory(
 0, 10, 5, 1));
```

- ❑ `DoubleSpinnerValueFactory` — создает диапазон вещественных значений. Пример:

```
Spinner<Double> spinner2 = new Spinner<Double>();
spinner2.setValueFactory(
 new SpinnerValueFactory.DoubleSpinnerValueFactory(
 0.0, 10.0, 5.0, 0.5));
```

- ❑ `ListSpinnerValueFactory<T>` — создает список значений. Пример:

```
Spinner<String> spinner3 = new Spinner<String>();
spinner3.setValueFactory(
 new SpinnerValueFactory.ListSpinnerValueFactory<String>(
 FXCollections.<String>observableArrayList(
 "red", "green", "blue")));
```

Стилевое оформление компонента задается с помощью следующих статических констант из класса `Spinner<T>`:

- ❑ `STYLE_CLASS_ARROWS_ON_LEFT_HORIZONTAL` — кнопки со стрелками расположены в левой части поля друг рядом с другом по горизонтали;
- ❑ `STYLE_CLASS_ARROWS_ON_LEFT_VERTICAL` — кнопки со стрелками расположены в левой части поля. Кнопка увеличения значения сверху, а кнопка уменьшения значения — снизу;
- ❑ `STYLE_CLASS_ARROWS_ON_RIGHT_HORIZONTAL` — кнопки со стрелками расположены в правой части поля друг рядом с другом по горизонтали;
- ❑ `STYLE_CLASS_SPLIT_ARROWS_HORIZONTAL` — кнопка увеличения значения расположена в правой части поля, а кнопка уменьшения значения — в левой части поля;
- ❑ `STYLE_CLASS_SPLIT_ARROWS_VERTICAL` — кнопка увеличения значения расположена над полем, а кнопка уменьшения значения — под полем.

По умолчанию кнопки со стрелками расположены в правой части поля. Кнопка увеличения значения — сверху, а кнопка уменьшения значения — снизу. Пример изменения стиля:

```
Spinner<Integer> intSpinner = new Spinner<Integer>(0, 100, 0);
intSpinner.getStyleClass().add(
 Spinner.STYLE_CLASS_ARROWS_ON_LEFT_HORIZONTAL);
```

Управлять компонентом позволяют следующие свойства:

- ❑ `value` — содержит текущее значение. Методы доступа:

```
public final T getValue()
public final ReadOnlyObjectProperty<T> valueProperty()
```

Если необходимо изменить значение из программы, то можно воспользоваться следующим кодом:

```
doubleSpinner.getValueFactory().setValue(3.1);
```

- ❑ `valueFactory` — содержит ссылку на объект, класс которого реализует абстрактный класс `SpinnerValueFactory<T>`. Методы доступа:

```
public final void setValueFactory(
 SpinnerValueFactory<T> value)
public final SpinnerValueFactory<T> getValueFactory()
public final ObjectProperty<SpinnerValueFactory<T>>
 valueFactoryProperty()
```

- ❑ `promptText` — задает текст подсказки пользователю, который будет выводиться в поле, когда оно не содержит значения и находится вне фокуса ввода. Свойство доступно, начиная с JavaFX 9. Методы доступа:

```
public final void setPromptText(String value)
public final String getPromptText()
public final StringProperty promptTextProperty()
```

- ❑ `initialDelay` — задает задержку перед первой сменой значения при удерживании нажатой кнопки со стрелкой. Значение по умолчанию: 300.0 ms. Свойство доступно, начиная с JavaFX 11. Методы доступа:

```
public final void setInitialDelay(Duration value)
public final Duration getInitialDelay()
public final ObjectProperty<Duration> initialDelayProperty()
```

- ❑ `repeatDelay` — задает задержку между последующими сменами значений при удерживании нажатой кнопки со стрелкой. Значение по умолчанию: 60.0 ms. Свойство доступно, начиная с JavaFX 11. Методы доступа:

```
public final void setRepeatDelay(Duration value)
public final Duration getRepeatDelay()
public final ObjectProperty<Duration> repeatDelayProperty()
```

- ❑ `editor` — содержит ссылку на текстовое поле. Методы доступа:

```
public final TextField getEditor()
public final ReadOnlyObjectProperty<TextField> editorProperty()
```

- ❑ `editable` — если свойство имеет значение `true`, то пользователь сможет изменить значение поля, например, путем ввода значения с клавиатуры. Если введенное значение нельзя будет преобразовать к типу компонента (например, пользователь вместо числа ввел буквы или оставил поле пустым), то будет сгенерировано исключение. Введенные данные передаются на преобразование после нажатия пользователем клавиши `<Enter>`, а также после нажатия кнопок со стрелками. Значение по умолчанию: `false`. Методы доступа:

```
public final void setEditable(boolean value)
public final boolean isEditable()
public final BooleanProperty editableProperty()
```

Давайте реализуем редактируемый пользователем компонент и обработаем ошибки, которые могут возникнуть в процессе преобразования типов:

```
Spinner<Integer> intSpinner = new Spinner<Integer>(0, 100, 5, 1);
intSpinner.setEditable(true);
intSpinner.getValueFactory().setConverter(
 new StringConverter<Integer>() {
 @Override
 public Integer fromString(String string) {
 if (string.matches("[0-9]+$")) {
 try {
 return Integer.valueOf(string);
 }
 catch (NumberFormatException e) {
 return 0;
 }
 }
 return 0;
 }
 @Override
 public String toString(Integer obj) {
 return (obj == null) ? "0" : obj.toString();
 }
 });
```

Класс `Spinner<T>` содержит также следующие методы:

- ❑ `increment()` — увеличивает значение на один шаг или на указанное количество шагов. Форматы метода:

```
public void increment()
public void increment(int steps)
```

- ❑ `decrement()` — уменьшает значение на один шаг или на указанное количество шагов. Форматы метода:

```
public void decrement()
public void decrement(int steps)
```

- ❑ `commitValue()` — применяет ввод и отправляет данные на преобразование типов (см. *разд. 12.2.7*). Эквивалентно нажатию пользователем клавиши `<Enter>`. Метод доступен, начиная с JavaFX 9. Формат метода:

```
public final void commitValue()
```

- ❑ `cancelEdit()` — отменяет ввод и вставляет в поле предыдущее значение. Эквивалентно нажатию пользователем клавиши `<Esc>`. Метод доступен, начиная с JavaFX 9. Формат метода:

```
public final void cancelEdit()
```

Если нужно, чтобы значение менялось по кругу — например, максимальное значение при следующем шаге менялось минимальным и наоборот, то следует воспользоваться свойством `wrapAround` из класса `SpinnerValueFactory<T>`. Методы доступа:



```
public final void setWrapAround(boolean value)
public final boolean isWrapAround()
public final BooleanProperty wrapAroundProperty()
```

**Пример:**

```
Spinner<Integer> intSpinner = new Spinner<Integer>(0, 100, 0);
intSpinner.getValueFactory().setWrapAround(true);
```

**Класс `SpinnerValueFactory<T>` содержит также свойство `value`, которое хранит текущее значение, и свойство `converter`, которое задает объект, выполняющий преобразование типов. Методы доступа к этим свойствам:**

```
public final void setValue(T newValue)
public final T getValue()
public final ObjectProperty<T> valueProperty()

public final void setConverter(StringConverter<T> newValue)
public final StringConverter<T> getConverter()
public final ObjectProperty<StringConverter<T>> converterProperty()
```

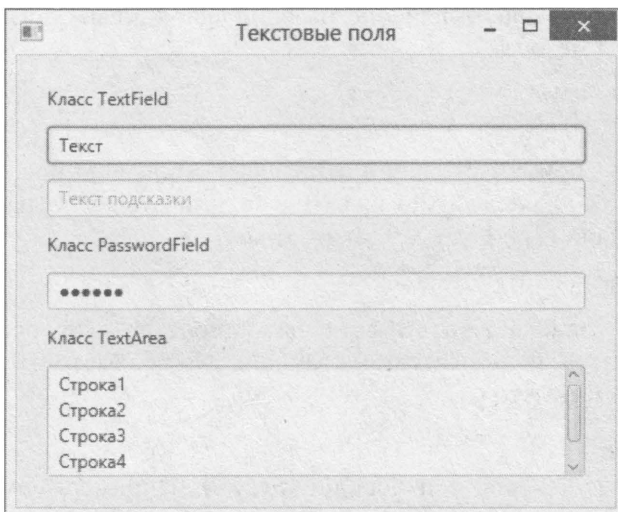
### **ЭЛЕКТРОННЫЙ АРХИВ**

Полный текст программы содержится в проекте `FX_CH12_01_Spinner`, приведенном в каталоге `CH12_text` сопровождающего книгу электронного архива (см. приложение).

## **12.2. Класс *TextInputControl***

Абстрактный класс `TextInputControl` является базовым классом для текстовых полей (рис. 12.2). Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.TextInputControl;
```



**Рис. 12.2. Текстовые поля**

## Иерархия наследования:

Object — Node — Parent — Region — Control — TextInputControl

### 12.2.1. Основные свойства и методы

Приведем основные свойства класса `TextInputControl`:

- ❑ `text` — содержит текст, отображаемый в текстовом поле. Методы доступа:

```
public final void setText(String value)
public final String getText()
public final StringProperty textProperty()
```

Пример вставки текста в поле из программы:

```
TextField text = new TextField();
text.setText("Иванов");
```

- ❑ `promptText` — задает текст подсказки пользователю, который будет выводиться в поле, когда оно не содержит значения и находится вне фокуса ввода. Методы доступа:

```
public final void setPromptText(String value)
public final String getPromptText()
public final StringProperty promptTextProperty()
```

Пример:

```
text.setPromptText("Введите фамилию");
```

- ❑ `font` — задает характеристики шрифта (см. *разд. 5.18.3*). Методы доступа:

```
public final void setFont(Font value)
public final Font getFont()
public final ObjectProperty fontProperty()
```

Пример:

```
text.setFont(Font.font("Verdana", 18.0));
```

- ❑ `length` — содержит количество символов в тексте. Методы доступа:

```
public final int getLength()
public final ReadOnlyIntegerProperty lengthProperty()
```

- ❑ `editable` — если свойство имеет значение `false`, то поле будет доступно только для чтения. Значение по умолчанию: `true` (пользователь может редактировать содержимое поля). Методы доступа:

```
public final void setEditable(boolean value)
public final boolean isEditable()
public final BooleanProperty editableProperty()
```

Приведем основные методы класса `TextInputControl`:

- ❑ `appendText()` — добавляет текст в конец существующего значения. Формат метода:

```
public void appendText(String text)
```

- ❑ `insertText()` — вставляет текст в указанную позицию. Формат метода:

```
public void insertText(int index, String text)
```

Вставим текст в начало поля:

```
text.insertText(0, "- ");
```

- ❑ `clear()` — очищает текстовое поле. Формат метода:

```
public void clear()
```

- ❑ `deletePreviousChar()` — удаляет символ, расположенный слева от текстового курсора. Формат метода:

```
public boolean deletePreviousChar()
```

- ❑ `deleteNextChar()` — удаляет символ, расположенный справа от текстового курсора. Формат метода:

```
public boolean deleteNextChar()
```

- ❑ `commitValue()` — применяет ввод и отправляет данные на преобразование типов (см. *разд. 12.2.7*). Эквивалентно нажатию пользователем клавиши `<Enter>`. Формат метода:

```
public final void commitValue()
```

- ❑ `cancelEdit()` — отменяет ввод и вставляет в поле предыдущее значение. Эквивалентно нажатию пользователем клавиши `<Esc>`. Формат метода:

```
public final void cancelEdit()
```

Текстовые поля по умолчанию реализуют функционал стандартных клавиш — например: `<Delete>` (удалить следующий символ), `<Backspace>` (удалить предыдущий символ) и др.

## 12.2.2. Управление позицией текстового курсора

Получить текущую позицию текстового курсора внутри поля можно с помощью свойства `caretPosition` из класса `TextInputControl`. Методы доступа:

```
public final int getCaretPosition()
```

```
public final ReadOnlyIntegerProperty caretPositionProperty()
```

Управлять позицией текстового курсора позволяют следующие методы из класса `TextInputControl`:

- ❑ `home()` — перемещает текстовый курсор в начало поля. Если существует выделенный фрагмент, то выделение снимается. Формат метода:

```
public void home()
```

- ❑ `positionCaret()` — перемещает текстовый курсор в указанную позицию (нумерация начинается с 0). Формат метода:  

```
public void positionCaret(int pos)
```
- ❑ `end()` — перемещает текстовый курсор в конец поля. Если существует выделенный фрагмент, то выделение снимается. Формат метода:  

```
public void end()
```
- ❑ `backward()` — сдвигает текстовый курсор на один символ к началу поля. Если существует выделенный фрагмент, то перемещает указатель в начало выделения и снимает выделение. Формат метода:  

```
public void backward()
```
- ❑ `forward()` — сдвигает текстовый курсор на один символ к концу поля. Если существует выделенный фрагмент, то перемещает указатель в конец выделения и снимает выделение. Формат метода:  

```
public void forward()
```
- ❑ `previousWord()` — сдвигает текстовый курсор к началу предыдущего (или текущего) слова. Если существует выделенный фрагмент, то выделение снимается. Формат метода:  

```
public void previousWord()
```
- ❑ `nextWord()` — сдвигает текстовый курсор к началу следующего слова. Если существует выделенный фрагмент, то выделение снимается. Формат метода:  

```
public void nextWord()
```
- ❑ `endOfNextWord()` — сдвигает текстовый курсор в конец следующего (или текущего) слова. Если существует выделенный фрагмент, то выделение снимается. Формат метода:  

```
public void endOfNextWord()
```

Текстовые поля по умолчанию реализуют функционал стандартных клавиш для перемещения текстового курсора — например: <Home> (в начало), <End> (в конец) и др. Клавиши со стрелками позволяют переместить текстовый курсор на одну позицию вперед, назад, вверх или вниз.

### 12.2.3. Класс *IndexRange*: работа с диапазоном

Для работы с диапазоном от начальной позиции до конечной предназначены следующие методы из класса `TextInputControl`:

- ❑ `getText()` — возвращает фрагмент между позициями `start` и `end`. Если индексы не существуют, то генерируется исключение. Формат метода:  

```
public String getText(int start, int end)
```
- ❑ `replaceText()` — заменяет фрагмент между позициями `start` и `end` указанным текстом. Если индексы не существуют, то генерируется исключение. Форматы метода:

```
public void replaceText(int start, int end, String text)
public void replaceText(IndexRange range, String text)
```

- ❑ `deleteText()` — удаляет фрагмент между позициями `start` и `end`. Если индексы не существуют, то генерируется исключение. Форматы метода:

```
public void deleteText(int start, int end)
public void deleteText(IndexRange range)
```

Диапазон индексов может быть представлен классом `IndexRange`. Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.IndexRange;
```

Конструкторы класса:

```
IndexRange(int start, int end)
IndexRange(IndexRange range)
```

Параметр `start` задает начальный индекс диапазона, а параметр `end` — конечный индекс. Если начальный индекс больше конечного, то генерируется исключение. Пример создания объекта и вывода значений:

```
IndexRange indexRange = new IndexRange(1, 5);
System.out.println(indexRange.getStart()); // 1
System.out.println(indexRange.getEnd()); // 5
System.out.println(indexRange.getLength()); // 4
```

Статический метод `valueOf()` позволяет создать объект на основе строки в формате `"start,end"`. Формат метода:

```
public static IndexRange valueOf(String value)
```

Пример:

```
IndexRange indexRange = IndexRange.valueOf("1,5");
System.out.println(indexRange.getStart()); // 1
System.out.println(indexRange.getEnd()); // 5
```

Статический метод `normalize()` создает объект и производит нормализацию индексов диапазона. Формат метода:

```
public static IndexRange normalize(int v1, int v2)
```

Пример:

```
IndexRange indexRange = IndexRange.normalize(5, 1);
System.out.println(indexRange.getStart()); // 1
System.out.println(indexRange.getEnd()); // 5
```

Класс `IndexRange` содержит следующие основные методы:

- ❑ `getStart()` — возвращает начальный индекс. Формат метода:

```
public int getStart()
```

- ❑ `getEnd()` — возвращает конечный индекс. Формат метода:

```
public int getEnd()
```

- ❑ `getLength()` — возвращает длину диапазона. Формат метода:

```
public int getLength()
```

## 12.2.4. Работа с выделением

Для работы с выделением внутри текстового поля предназначены следующие свойства класса `TextInputControl`:

- ❑ `anchor` — содержит начальную позицию диапазона выделения. Конечная позиция диапазона выделения содержится в свойстве `caretPosition` (см. *разд. 12.2.2*). Обратите внимание: пользователь может выделять текст в разных направлениях, а это значит, что индекс начальной позиции может быть как меньше, так и больше индекса конечной позиции диапазона выделения. Если выделенного фрагмента нет, то индексы позиций будут равны. Методы доступа:

```
public final int getAnchor()
public final ReadOnlyIntegerProperty anchorProperty()
```

- ❑ `selectedText` — содержит текст выделенного фрагмента. Методы доступа:

```
public final String getSelectedText()
public final ReadOnlyStringProperty selectedTextProperty()
```

- ❑ `selection` — содержит диапазон выделенного фрагмента. Методы доступа:

```
public final IndexRange getSelection()
public final ReadOnlyObjectProperty<IndexRange>
 selectionProperty()
```

Для работы с выделением предназначены следующие методы из класса `TextInputControl`:

- ❑ `selectAll()` — выделяет все содержимое поля. Формат метода:

```
public void selectAll()
```

- ❑ `deselect()` — снимает выделение. Формат метода:

```
public void deselect()
```

- ❑ `selectRange()` — выделяет указанный диапазон. Формат метода:

```
public void selectRange(int anchor, int caretPosition)
```

- ❑ `selectPositionCaret()` — задает новое значение свойства `caretPosition`. Если выделения не было, то оно образуется от предыдущей позиции текстового курсора до новой позиции. Если выделение уже было, то будет выделен диапазон от `anchor` до `caretPosition`. Формат метода:

```
public void selectPositionCaret(int pos)
```

- ❑ `extendSelection()` — изменяет выделение таким образом, чтобы указанная позиция стала границей выделения. Если выделения не было, то оно образуется от предыдущей позиции текстового курсора до новой позиции. Формат метода:

```
public void extendSelection(int pos)
```

Давайте рассмотрим отличие метода `extendSelection()` от метода `selectPositionCaret()`. Предположим, что у нас есть текст `Текст` внутри поля и выделено слово поля в направлении от буквы `п` до буквы `я`. В итоге свойство `anchor` имеет значение 13, а свойство `caretPosition` — 17.

После выполнения этого кода:

```
text.selectPositionCaret(5);
```

свойство `anchor` будет иметь значение 13, а свойство `caretPosition` — 5. В итоге будет выделен фрагмент " внутри ".

После выполнения этого кода:

```
text.extendSelection(5);
```

свойство `anchor` будет иметь значение 5, а свойство `caretPosition` — 17. В итоге будет выделен фрагмент " внутри поля";

- ❑ `selectBackward()` — перемещает позицию `caretPosition` на одну букву назад, изменяя при этом выделение. Формат метода:  

```
public void selectBackward()
```
- ❑ `selectForward()` — перемещает позицию `caretPosition` на одну букву вперед, изменяя при этом выделение. Формат метода:  

```
public void selectForward()
```
- ❑ `selectPreviousWord()` — перемещает позицию `caretPosition` к началу предыдущего (или текущего) слова, изменяя при этом выделение. Формат метода:  

```
public void selectPreviousWord()
```
- ❑ `selectNextWord()` — перемещает позицию `caretPosition` к началу следующего слова, изменяя при этом выделение. Формат метода:  

```
public void selectNextWord()
```
- ❑ `selectEndOfNextWord()` — перемещает позицию `caretPosition` в конец следующего (или текущего) слова, изменяя при этом выделение. Формат метода:  

```
public void selectEndOfNextWord()
```
- ❑ `selectHome()` — перемещает позицию `caretPosition` в начало поля, изменяя при этом выделение. Формат метода:  

```
public void selectHome()
```
- ❑ `selectEnd()` — перемещает позицию `caretPosition` в конец поля, изменяя при этом выделение. Формат метода:  

```
public void selectEnd()
```
- ❑ `replaceSelection()` — заменяет выделенный фрагмент указанным текстом и снимает выделение. Если выделения нет, то текст вставляется в текущую позицию текстового курсора. Формат метода:  

```
public void replaceSelection(String replacement)
```

Текстовые поля по умолчанию реализуют стандартные сочетания клавиш: **<Ctrl>+<A>** (выделить все), **<Shift>** плюс клавиши со стрелками (изменение границ выделения), **<Shift>+<End>** (выделение до конца поля) и др. Кроме того, пользователю в контекстном меню доступен пункт **Select All**.

## 12.2.5. Копирование и вставка текста

Выполнить копирование и вставку текста позволяют следующие методы из класса `TextInputControl`:

- ❑ `copy()` — копирует выделенный текст в буфер обмена. Формат метода:

```
public void copy()
```

- ❑ `cut()` — вырезает выделенный текст и вставляет его в буфер обмена. Формат метода:

```
public void cut()
```

- ❑ `paste()` — вставляет текст из буфера обмена в текущую позицию текстового курсора или вместо выделенного фрагмента. Формат метода:

```
public void paste()
```

Текстовые поля по умолчанию реализуют стандартные сочетания клавиш: **<Ctrl>+<C>** (копировать), **<Ctrl>+<X>** (вырезать) и **<Ctrl>+<V>** (вставить). Кроме того, пользователю в контекстном меню доступны пункты **Copy**, **Cut** и **Paste**.

## 12.2.6. Отмена и повтор ввода

Текстовые поля поддерживают возможность отмены и повтора ввода. Для этого предназначены следующие свойства из класса `TextInputControl`:

- ❑ `undoable` — содержит значение `true`, если доступна операция отмены последнего ввода. Методы доступа:

```
public final boolean isUndoable()
```

```
public final ReadOnlyBooleanProperty undoableProperty()
```

- ❑ `redoable` — содержит значение `true`, если доступна операция повтора ранее отмененного ввода. Методы доступа:

```
public final boolean isRedoable()
```

```
public final ReadOnlyBooleanProperty redoableProperty()
```

Управлять операциями отмены и повтора ввода позволяют следующие методы:

- ❑ `undo()` — отменяет последнюю операцию ввода. Формат метода:

```
public final void undo()
```

- ❑ `redo()` — повторяет ранее отмененную операцию ввода. Формат метода:

```
public final void redo()
```



Текстовые поля по умолчанию реализуют стандартные сочетания клавиш: <Ctrl>+<Z> (отменить) и <Ctrl>+<Y> (повторить). Кроме того, пользователю в контекстном меню доступны пункты **Undo** и **Redo**.

## 12.2.7. Класс *TextFormatter<V>*

Начиная с версии 8.40, класс `TextInputControl` содержит свойство `textFormatter`, которое позволяет указать объект класса `TextFormatter<V>`, с помощью которого можно контролировать ввод данных и реализовать преобразование типов. Методы доступа:

```
public final void setTextFormatter(TextFormatter<?> value)
public final TextFormatter<?> getTextFormatter()
public final ObjectProperty<TextFormatter<?>> textFormatterProperty()
```

Инструкция импорта класса `TextFormatter<V>`:

```
// Модуль javafx.controls
import javafx.scene.control.TextFormatter;
```

Конструкторы класса:

```
import javafx.util.StringConverter;
import java.util.function.UnaryOperator;
TextFormatter(StringConverter<V> valueConverter)
TextFormatter(StringConverter<V> valueConverter, V defaultValue)
TextFormatter(StringConverter<V> valueConverter, V defaultValue,
 UnaryOperator<TextFormatter.Change> filter)
TextFormatter(UnaryOperator<TextFormatter.Change> filter)
```

Параметр `valueConverter` задает объект, который будет выполнять преобразование типов, а параметр `defaultValue` позволяет указать значение по умолчанию. Введенное в поле значение передается объекту `valueConverter` после нажатия пользователем клавиши <Enter>, а не во время ввода. Ограничим ввод целочисленным значением типа `Integer`:

```
// import javafx.util.converter.IntegerStringConverter;
TextFormatter<Integer> tf = new TextFormatter<Integer>(
 new IntegerStringConverter(), 0);
TextField text = new TextField("0");
text.setTextFormatter(tf);
```

Получим значение после нажатия кнопки:

```
Button button = new Button("Получить значение");
button.setOnAction(event -> {
 Integer value = (Integer) text.getTextFormatter().getValue();
 System.out.println(value);
});
```

В этом примере для получения значения мы воспользовались свойством `value` из класса `TextFormatter<V>`. Методы доступа:

```
public final void setValue(V value)
public final V getValue()
public final ObjectProperty<V> valueProperty()
```

В параметре `valueConverter` можно указать статическую константу `IDENTITY_STRING_CONVERTER` из класса `TextFormatter<V>`, которая содержит объект, преобразующий строку в ту же самую строку. Это может пригодиться, если нужно указать только параметр `defaultValue`. Формат:

```
public static final StringConverter<String> IDENTITY_STRING_CONVERTER
```

Параметр `filter` позволяет указать ссылку на объект, реализующий функциональный интерфейс `UnaryOperator`. Вводимое в поле значение передается этому объекту постоянно во время ввода, а также во время выделения фрагмента. Внутри этого объекта мы можем проверить данные. Если данные соответствуют нашим ожиданиям, то возвращаем данные как есть или модифицируем их, а если не соответствуют, то возвращаем значение `null`. Если мы возвращаем `null`, то новые данные в поле не будут вставлены. Ограничим ввод английскими буквами в нижнем регистре:

```
UnaryOperator<TextFormatter.Change> filter = change -> {
 String s = change.getControlNewText();
 if (s.matches("^[a-z]+)?$")) {
 return change;
 }
 return null;
};

TextFormatter<String> tf2 = new TextFormatter<String>(
 TextFormatter.IDENTITY_STRING_CONVERTER, "abc", filter);
TextField text2 = new TextField();
text2.setTextFormatter(tf2);
```

В этом примере для создания объекта фильтра мы воспользовались лямбда-выражением, которое в качестве параметра принимает объект статического класса `TextFormatter.Change`. Приведем основные методы этого класса:

❑ `getText()` — возвращает добавленный фрагмент или пустую строку (например, при удалении символа). Формат метода:

```
public final String getText()
```

❑ `getControlNewText()` — возвращает новый текст с учетом изменений. Формат метода:

```
public final String getControlNewText()
```

❑ `getControlText()` — возвращает текст до изменения. Формат метода:

```
public final String getControlText()
```

❑ `isAdded()` — возвращает значение `true`, если был добавлен текст. Формат метода:

```
public final boolean isAdded()
```

- ❑ `isDeleted()` — **возвращает значение** `true`, если был удален текст. Формат метода:  
`public final boolean isDeleted()`
- ❑ `isReplaced()` — **возвращает значение** `true`, если был заменен текст. Формат метода:  
`public final boolean isReplaced()`
- ❑ `isContentChange()` — **возвращает значение** `true`, если текст был добавлен, удален или заменен, и `false` — в противном случае (например, пользователь просто выделил фрагмент внутри поля). Формат метода:  
`public final boolean isContentChange()`
- ❑ `getAnchor()` и `getCaretPosition()` — **возвращают позицию** `anchor` и `caretPosition` с учетом изменений. Форматы методов:  
`public final int getAnchor()`  
`public final int getCaretPosition()`
- ❑ `getControlAnchor()` и `getControlCaretPosition()` — **возвращают позицию** `anchor` и `caretPosition` до изменений. Форматы методов:  
`public final int getControlAnchor()`  
`public final int getControlCaretPosition()`
- ❑ `getSelection()` — **возвращает диапазон** выделенного фрагмента с учетом изменений. Формат метода:  
`public final IndexRange getSelection()`
- ❑ `getRangeStart()` и `getRangeEnd()` — **возвращают начальный и конечный индексы** для модификации. Форматы методов:  
`public final int getRangeStart()`  
`public final int getRangeEnd()`
- ❑ `getControl()` — **возвращает ссылку** на текстовое поле. Формат метода:  
`public final Control getControl()`
- ❑ `setText()` — **позволяет указать новый фрагмент** вместо фрагмента, возвращаемого методом `getText()`. Формат метода:  
`public final void setText(String value)`
- ❑ `setRange()` — **позволяет изменить начальный и конечный индексы** для модификации. Формат метода:  
`public final void setRange(int start, int end)`  
  
Сделаем так, чтобы добавленный фрагмент всегда вставлялся в начало поля, вне зависимости от положения текстового курсора в момент ввода:  
`if (change.isAdded()) change.setRange(0, 0);`
- ❑ `setAnchor()`, `setCaretPosition()` и `selectRange()` — **задают позицию** `anchor` и (или) `caretPosition` после изменений. Форматы методов:

```
public final void setAnchor(int newAnchor)
public final void setCaretPosition(int newCaretPosition)
public final void selectRange(int newAnchor,
 int newCaretPosition)
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH12_03_TextField`, приведенном в каталоге `CH12_text` сопровождающего книгу электронного архива (см. приложение).

## 12.3. Класс *TextField*: однострочное текстовое поле

Класс `TextField` реализует однострочное текстовое поле, предназначенное для ввода и редактирования текста небольшого объема (см. рис. 12.2). Поле по умолчанию поддерживает стандартные комбинации клавиш быстрого доступа, работу с буфером обмена и многое другое. Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.TextField;
```

Иерархия наследования:

Object — Node — Parent — Region — Control — `TextInputControl` — `TextField`

Конструкторы класса:

```
TextField()
TextField(String text)
```

Первый конструктор создает пустое поле, а второй конструктор позволяет указать текст, который будет отображаться в поле. Создадим текстовое поле, назначим текст подсказки и клавишу быстрого доступа:

```
TextField text = new TextField();
text.setPromptText("Введите имя");
Label label = new Label("_Имя");
label.setLabelFor(text);
label.setMnemonicParsing(true);
```

Обратите внимание: так как текстовое поле не имеет надписи, назначение клавиши быстрого доступа осуществляется с помощью отдельного компонента `Label`. Связь между надписью и текстовым полем задается с помощью метода `setLabelFor()`. В нашем примере при нажатии комбинации клавиш `<Alt>+<И>` текстовое поле окажется в фокусе ввода.

Получим значение поля после нажатия кнопки:

```
Button button = new Button("Получить значение");
button.setOnAction(event -> {
 System.out.println(text.getText());
});
```

Для получения значения поля можно также воспользоваться методом `getCharacters()` из класса `TextField`. Формат метода:

```
public CharSequence getCharacters()
```

**Пример:**

```
System.out.println(text.getCharacters());
```

Класс `TextField` наследует все методы из класса `TextInputControl` (см. *разд. 12.2*), а также из других базовых классов (не забываете смотреть на иерархию наследования), и добавляет следующие свойства:

- `alignment` — задает выравнивание текста внутри свободной области поля. В качестве значения указываются следующие константы из перечисления `javafx.geometry.Pos`: `BASELINE_CENTER`, `BASELINE_LEFT`, `BASELINE_RIGHT`, `BOTTOM_CENTER`, `BOTTOM_LEFT`, `BOTTOM_RIGHT`, `CENTER`, `CENTER_LEFT`, `CENTER_RIGHT`, `TOP_CENTER`, `TOP_LEFT` и `TOP_RIGHT`. Значение по умолчанию: `CENTER_LEFT`. Методы доступа:

```
public final void setAlignment(Pos value)
public final Pos getAlignment()
public final ObjectProperty<Pos> alignmentProperty()
```

**Пример выравнивания текста по правому краю поля:**

```
TextField text3 = new TextField("Текст");
text3.setAlignment(Pos.CENTER_RIGHT);
```

- `prefColumnCount` — значение этого свойства используется для вычисления предпочтительной ширины поля. Методы доступа:

```
public final void setPrefColumnCount(int value)
public final int getPrefColumnCount()
public final IntegerProperty prefColumnCountProperty()
```

Значение по умолчанию хранится в константе `DEFAULT_PREF_COLUMN_COUNT`. Формат:

```
public static final int DEFAULT_PREF_COLUMN_COUNT
```

**Выведем значение константы:**

```
System.out.println(TextField.DEFAULT_PREF_COLUMN_COUNT); // 12
```

- `onAction` — позволяет назначить обработчик события нажатия пользователем клавиши `<Enter>`. Методы доступа:

```
public final void setOnAction(EventHandler<ActionEvent> value)
public final EventHandler<ActionEvent> getOnAction()
public final ObjectProperty<EventHandler<ActionEvent>>
 onActionProperty()
```

## ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH12_03_TextField`, приведенном в каталоге *CH12\_text* сопровождающего книгу электронного архива (см. *приложение*).

## 12.4. Класс *PasswordField*: поле для ввода пароля

Класс `PasswordField` расширяет класс `TextField` и реализует однострочное текстовое поле, предназначенное для ввода пароля. При вводе пароля в поле отображаются черные кружки, а не текст (см. рис. 12.2). Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.PasswordField;
```

Иерархия наследования:

```
Object — Node — Parent — Region — Control — TextInputControl —
 TextField — PasswordField
```

Конструктор класса:

```
PasswordField()
```

Пример создания поля и назначения клавиши быстрого доступа:

```
PasswordField passwordField = new PasswordField();
passwordField.setPromptText("Введите пароль");
Label label = new Label("_Пароль");
label.setMnemonicParsing(true);
label.setLabelFor(passwordField);
```

Пример получения значения после нажатия кнопки:

```
Button button = new Button("Получить значения");
pane.getChildren().add(button);
button.setOnAction(event -> {
 System.out.println(passwordField.getText());
});
```

Класс `PasswordField` наследует все методы из базовых классов и не добавляет никаких собственных методов. Однако он переопределяет методы `cut()` и `copy()`. Эти методы в классе `PasswordField` ничего не делают.

### **ЭЛЕКТРОННЫЙ АРХИВ**

Полный текст программы содержится в проекте `FX_CH12_03_TextField`, приведенном в каталоге `CH12_text` сопровождающего книгу электронного архива (см. приложение).

## 12.5. Класс *TextArea*: многострочное текстовое поле

Класс `TextArea` реализует многострочное текстовое поле (см. рис. 12.2). Перенос текста на новую строку осуществляется путем нажатия клавиши `<Enter>`, а из программы путем ввода символа перевода строки `\n`. Если текст не помещается в поле, то автоматически отображаются полосы прокрутки. Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.TextArea;
```

### Иерархия наследования:

Object — Node — Parent — Region — Control — TextInputControl — TextArea

### Конструкторы класса:

```
TextArea()
TextArea(String text)
```

### Пример отображения текста на двух строках и назначения клавиши быстрого доступа:

```
TextArea text = new TextArea("Строка1\nСтрока2");
text.setWrapText(true);
Label label = new Label("_Комментарий");
label.setLabelFor(text);
label.setMnemonicParsing(true);
```

### Пример получения значения после нажатия кнопки:

```
Button button = new Button("Получить значение");
button.setOnAction(event -> {
 System.out.println(text.getText());
});
```

**Получить значение поля можно также с помощью метода `getParagraphs()` из класса `TextArea`. Метод возвращает список всех строк без символов перевода строки `\n`. Формат метода:**

```
public ObservableList<CharSequence> getParagraphs()
```

### Пример перебора всех строк:

```
ObservableList<CharSequence> list = text.getParagraphs();
for (CharSequence s: list) {
 System.out.println(s);
}
```

**Класс `TextArea` наследует все методы из класса `TextInputControl` (см. *разд. 12.2*), а также из других базовых классов, и добавляет следующие свойства:**

- ❑ `wrapText` — если свойство имеет значение `true`, и текст не помещается на строке, он будет автоматически переноситься на новую строку. Значение по умолчанию: `false`. Методы доступа:

```
public final void setWrapText(boolean value)
public final boolean isWrapText()
public final BooleanProperty wrapTextProperty()
```

- ❑ `prefColumnCount` — значение этого свойства используется для вычисления предпочтительной ширины поля. Методы доступа:

```
public final void setPrefColumnCount(int value)
public final int getPrefColumnCount()
public final IntegerProperty prefColumnCountProperty()
```

Значение по умолчанию хранится в константе `DEFAULT_PREF_COLUMN_COUNT`. Формат:

```
public static final int DEFAULT_PREF_COLUMN_COUNT
```

Выведем значение константы:

```
System.out.println(TextArea.DEFAULT_PREF_COLUMN_COUNT); // 40
```

- `prefRowCount` — значение этого свойства используется для вычисления предпочтительной высоты поля. Методы доступа:

```
public final void setPrefRowCount(int value)
public final int getPrefRowCount()
public final IntegerProperty prefRowCountProperty()
```

Значение по умолчанию хранится в константе `DEFAULT_PREF_ROW_COUNT`. Формат:

```
public static final int DEFAULT_PREF_ROW_COUNT
```

Выведем значение константы:

```
System.out.println(TextArea.DEFAULT_PREF_ROW_COUNT); // 10
```

- `scrollLeft` — количество пикселей, на которое содержимое поля прокручено по горизонтали. Методы доступа:

```
public final void setScrollLeft(double value)
public final double getScrollLeft()
public final DoubleProperty scrollLeftProperty()
```

- `scrollTop` — количество пикселей, на которое содержимое поля прокручено по вертикали. Методы доступа:

```
public final void setScrollTop(double value)
public final double getScrollTop()
public final DoubleProperty scrollTopProperty()
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH12_03_TextField`, приведенном в каталоге `CH12_text` сопровождающего книгу электронного архива (см. приложение).

## 12.6. Класс *HTMLEditor*: текстовый редактор

Класс `HTMLEditor` реализует текстовый редактор, с помощью которого пользователь может создавать форматированный текст (рис. 12.3). Редактор содержит кнопки для работы с буфером обмена, для выравнивания текста по горизонтали, для создания нумерованного или маркированного списка, для форматирования текста (подчеркивание, зачеркивание, выделение полужирным и курсивом, создание абзаца и заголовков различного уровня), кроме того, он позволяет указать характеристики



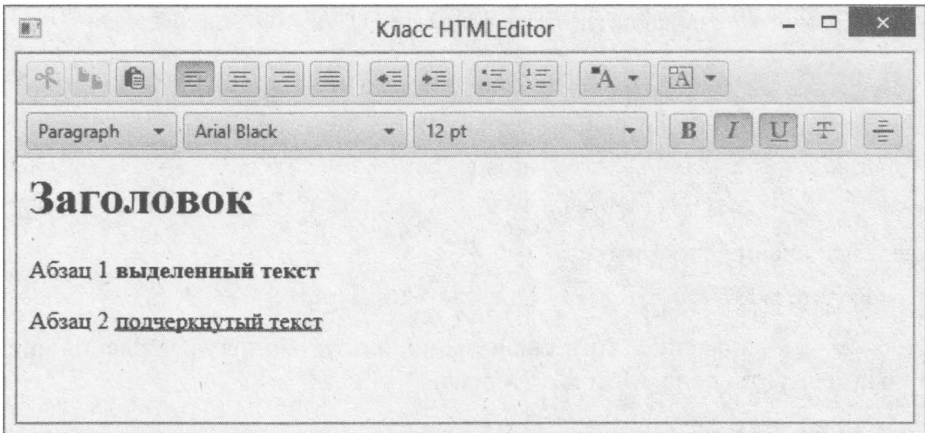


Рис. 12.3. Класс HTMLEditor

шрифта (название шрифта, размер и цвет) и фона. В основе редактора лежит текст в формате HTML, однако пользователь видит лишь результат его отображения, а не исходный HTML-код. Инструкция импорта:

```
// Модуль javafx.web
import javafx.scene.web.HTMLEditor;
```

#### Иерархия наследования:

Object — Node — Parent — Region — Control — HTMLEditor

#### Конструктор класса:

```
HTMLEditor()
```

#### Пример создания редактора и добавления HTML-кода:

```
HTMLEditor editor = new HTMLEditor();
editor.setHtmlText (
 "<html><head><title>Название</title></head><body>"
 + "<h1>Заголовок</h1>"
 + "<p>Абзац 1 выделенный текст</p>"
 + "<p>Абзац 2 <u>подчеркнутый текст</u></p>"
 + "</body></html>");
```

#### Пример получения значения в формате HTML после нажатия кнопки:

```
Button button = new Button("Получить HTML-текст");
button.setOnAction(event -> {
 System.out.println(editor.getHtmlText());
});
```

#### Класс HTMLEditor содержит следующие основные методы:

- ❑ `setHtmlText()` — задает текст в формате HTML, который будет отображаться в текстовом поле редактора. Формат метода:

```
public void setHtmlText(String htmlText)
```

❑ `getHtmlText()` — возвращает текст в формате HTML. Формат метода:

```
public String getHtmlText()
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH12_06_HTMLEditor`, приведенном в каталоге `CH12_text` сопровождающего книгу электронного архива (см. приложение).

## 12.7. Класс *WebView*: Web-браузер

Класс `WebView` реализует полноценный Web-браузер с поддержкой HTML 5 и JavaScript. Именно в `WebView` отображается HTML-код при использовании компонента `HTMLEditor`. Инструкция импорта:

```
// Модуль javafx.web
import javafx.scene.web.WebView;
```

Иерархия наследования:

Object — Node — Parent — `WebView`

Конструктор класса:

```
WebView()
```

Давайте отобразим в компоненте `WebView` документацию по Java (ее нужно предварительно скачать отдельно и скопировать в папку `C:\OpenJDK\jdk-11\docs`):

```
WebView webView = new WebView();
WebEngine webEngine = webView.getEngine();
webEngine.load("file:///C:/OpenJDK/jdk-11/docs/api/index.html");
```

Точно таким же образом можно загрузить любую страницу из Интернета. Теперь получим исходный HTML-код после нажатия кнопки:

```
Button button = new Button("Получить HTML-код");
button.setOnAction(event -> {
 text.setText((String) webEngine.executeScript(
 "document.documentElement.outerHTML"));
});
```

По умолчанию содержимое компонента `WebView` доступно только для чтения. Однако если в теге `<body>` указать атрибут `contentEditable` со значением `true`, то Web-страницу можно будет редактировать в окне Web-браузера. Давайте загрузим HTML-код из строки и добавим возможность редактирования содержимого `WebView`:

```
WebView webView = new WebView();
WebEngine webEngine = webView.getEngine();
webEngine.loadContent(
 "<html><head><title>Название</title></head>"
```

```
+ "<body contentEditable=\"true\">"
+ "<h1>Заголовок</h1>\n"
+ "<p>Абзац выделенный текст</p>"
+ "</body></html>");
```

Управлять компонентом позволяют следующие свойства:

❑ **width** — содержит текущую ширину компонента. Методы доступа:

```
public final double getWidth()
public ReadOnlyDoubleProperty widthProperty()
```

❑ **height** — содержит текущую высоту компонента. Методы доступа:

```
public final double getHeight()
public ReadOnlyDoubleProperty heightProperty()
```

❑ **prefWidth** — задает предпочтительную ширину компонента. Методы доступа:

```
public final void setPrefWidth(double value)
public final double getPrefWidth()
public DoubleProperty prefWidthProperty()
```

❑ **prefHeight** — задает предпочтительную высоту компонента. Методы доступа:

```
public final void setPrefHeight(double value)
public final double getPrefHeight()
public DoubleProperty prefHeightProperty()
```

Предпочтительные размеры компонента можно также задать с помощью метода `setPrefSize()`. Формат метода:

```
public void setPrefSize(double prefWidth, double prefHeight)
```

❑ **minWidth** — задает минимальную ширину компонента. Методы доступа:

```
public final void setMinWidth(double value)
public final double getMinWidth()
public DoubleProperty minWidthProperty()
```

❑ **minHeight** — задает минимальную высоту компонента. Методы доступа:

```
public final void setMinHeight(double value)
public final double getMinHeight()
public DoubleProperty minHeightProperty()
```

Минимальные размеры компонента можно также задать с помощью метода `setMinSize()`. Формат метода:

```
public void setMinSize(double minWidth, double minHeight)
```

❑ **maxWidth** — задает максимальную ширину компонента. Методы доступа:

```
public final void setMaxWidth(double value)
public final double getMaxWidth()
public DoubleProperty maxWidthProperty()
```

- ❑ **maxHeight** — задает максимальную высоту компонента. Методы доступа:

```
public final void setMaxHeight(double value)
public final double getMaxHeight()
public DoubleProperty maxHeightProperty()
```

**Максимальные размеры компонента можно также задать с помощью метода `setMaxSize()`. Формат метода:**

```
public void setMaxSize(double maxWidth, double maxHeight)
```

- ❑ **fontSmoothingType** — задает режим сглаживания шрифта. В качестве значения параметра можно указать константы `LCD` и `GRAY` из перечисления `FontSmoothingType`. Значение по умолчанию: `LCD`. Методы доступа:

```
import javafx.scene.text.FontSmoothingType;
public final void setFontSmoothingType(FontSmoothingType value)
public final FontSmoothingType getFontSmoothingType()
public final ObjectProperty<FontSmoothingType>
 fontSmoothingTypeProperty()
```

- ❑ **fontScale** — задает масштаб шрифта. Значение по умолчанию: 1.0 (соответствует масштабу 100 %). Методы доступа:

```
public final void setFontScale(double value)
public final double getFontScale()
public DoubleProperty fontScaleProperty()
```

- ❑ **zoom** — задает масштаб всей Web-страницы. Значение по умолчанию: 1.0 (соответствует масштабу 100 %). Методы доступа:

```
public final void setZoom(double value)
public final double getZoom()
public final DoubleProperty zoomProperty()
Установим масштаб 200 %:
webView.setZoom(2.0);
```

- ❑ **contextMenuEnabled** — если свойство имеет значение `false`, то контекстное меню отображаться не будет. Значение по умолчанию: `true`. Методы доступа:

```
public final void setContextMenuEnabled(boolean value)
public final boolean isContextMenuEnabled()
public final BooleanProperty contextMenuEnabledProperty()
```

### **ЭЛЕКТРОННЫЙ АРХИВ**

Полный текст программы содержится в проекте `FX_CH12_07_WebView`, приведенном в каталоге `CH12_text` сопровождающего книгу электронного архива (см. приложение).

## **12.8. Класс *WebEngine***

Класс `WebEngine` описывает Web-страницу, отображаемую компонентом `WebView`. Получить ссылку на объект этого класса позволяет метод `getEngine()` из класса `WebView`. Формат метода:

```
public final WebEngine getEngine()
```

Инструкция импорта класса `WebEngine`:

```
// Модуль javafx.web
import javafx.scene.web.WebEngine;
```

Конструкторы класса:

```
WebEngine()
WebEngine(String url)
```

## 12.8.1. Загрузка Web-страницы

Выполнить загрузку Web-страницы позволяют следующие методы из класса `WebEngine`:

- ❑ `load()` — позволяет загрузить Web-страницу, расположенную по указанному URL. Загрузка выполняется в фоновом потоке. Формат метода:

```
public void load(String url)
```

Пример:

```
WebView webView = new WebView();
WebEngine webEngine = webView.getEngine();
webEngine.setUserAgent("MyBot/1.0");
webEngine.load("http://localhost/");
```

- ❑ `loadContent()` — загружает код из строки. Загрузка выполняется в фоновом потоке. Форматы метода:

```
public void loadContent(String content)
public void loadContent(String content, String contentType)
```

Пример:

```
WebView webView = new WebView();
WebEngine webEngine = webView.getEngine();
webEngine.loadContent(
 "<html><head><title>Название</title></head>"
 + "<body>"
 + "<h1>Заголовок</h1>\n"
 + "<p>Абзац выделенный текст</p>"
 + "</body></html>");
```

- ❑ `reload()` — перезагружает Web-страницу. Формат метода:

```
public void reload()
```

С помощью свойства `userAgent` можно указать название Web-браузера, которое будет отправляться серверу в HTTP-заголовке `User-Agent`. Методы доступа:

```
public final void setUserAgent(String value)
public final String getUserAgent()
public final StringProperty userAgentProperty()
```

Получим название Web-браузера по умолчанию:

```
System.out.println(webEngine.getUserAgent());
// Mozilla/5.0 (Windows NT 6.2; Win64; x64)
// AppleWebKit/606.1 (KHTML, like Gecko) JavaFX/12 Safari/606.1
```

Изменим название Web-браузера:

```
webEngine.setUserAgent("MyBot/1.0");
```

Загрузка Web-страницы осуществляется в отдельном потоке. Чтобы иметь возможность обработать события загрузки, следует вызвать метод `getLoadWorker()` и получить ссылку на объект, класс которого реализует интерфейс `Worker<V>`. Напомню, что этот интерфейс мы уже рассматривали в *разд. 11.11.4*. Формат метода `getLoadWorker()`:

```
public final Worker<Void> getLoadWorker()
```

Обработаем статусы: `SUCCEEDED` (загрузка успешно выполнена), `FAILED` (ошибка при загрузке) и `CANCELLED` (загрузка отменена), а также создадим кнопку, при нажатии которой прервем загрузку с помощью метода `cancel()` из интерфейса `Worker<V>`:

```
WebView webView = new WebView();
WebEngine webEngine = webView.getEngine();
webEngine.setUserAgent("MyBot/1.0");
webEngine.getLoadWorker().stateProperty().addListener(
 (obj, oldValue, newValue) -> {
 if (newValue == Worker.State.SUCCEEDED) {
 System.out.println("Загрузка выполнена успешно");
 }
 else if (newValue == Worker.State.FAILED) {
 System.out.println("Ошибка");
 }
 else if (newValue == Worker.State.CANCELLED) {
 webEngine.loadContent("<h1>Загрузка отменена</h1>");
 }
 });
Button btnCancel = new Button("Отменить");
btnCancel.setOnAction(event -> {
 webEngine.getLoadWorker().cancel();
});
btnCancel.disableProperty().bind(
 webEngine.getLoadWorker().runningProperty().not());
webEngine.load("http://localhost/");
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH12_07_WebView`, приведенном в каталоге `CH12_text` сопровождающего книгу электронного архива (см. приложение).

## 12.8.2. Получение информации о Web-странице

Получить информацию о Web-странице позволяют следующие свойства из класса `WebEngine`:

- ❑ `location` — содержит URL Web-страницы или пустую строку (например, если загружаем Web-страницу из строки). Методы доступа:

```
public final String getLocation()
public final ReadOnlyStringProperty locationProperty()
```

- ❑ `title` — содержит заголовок Web-страницы (значение тега `<title>`) или значение `null`. Методы доступа:

```
public final String getTitle()
public final ReadOnlyStringProperty titleProperty()
```

- ❑ `document` — содержит объект `Document`, с помощью которого можно получить доступ к объектной модели HTML-документа. Методы доступа:

```
import org.w3c.dom.Document;
public final Document getDocument()
public final ReadOnlyObjectProperty<Document>
 documentProperty()
```

- ❑ `userStyleSheetLocation` — задает местонахождение таблицы стилей. Значение по умолчанию: `null`. Методы доступа:

```
public final void setUserStyleSheetLocation(String value)
public final String getUserStyleSheetLocation()
public final StringProperty userStyleSheetLocationProperty()
```

- ❑ `userDataDirectory` — задает каталог, который будет использоваться для хранения локальных данных пользователя. Установка нового значения должна выполняться до начала загрузки. Значение по умолчанию: `null`. Методы доступа:

```
public final void setUserDataDirectory(File value)
public final File getUserDataDirectory()
public final ObjectProperty<File> userDataDirectoryProperty()
```

Получим HTML-код после нажатия кнопки:

```
Button button = new Button("Получить HTML-код");
button.setOnAction(event -> {
 System.out.println((String) webEngine.executeScript(
 "document.documentElement.outerHTML"));
});
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH12_07_WebView`, приведенном в каталоге `CH12_text` сопровождающего книгу электронного архива (см. *приложение*).

### 12.8.3. Взаимодействие с JavaScript

Класс `WebEngine` позволяет выполнять скрипты, написанные на языке JavaScript. С помощью свойства `javaScriptEnabled` можно включить (значение `true`) или отключить (значение `false`) поддержку JavaScript. По умолчанию поддержка включена. Методы доступа:

```
public final void setJavaScriptEnabled(boolean value)
public final boolean isJavaScriptEnabled()
public final BooleanProperty javaScriptEnabledProperty()
```

Некоторые функции JavaScript не реализованы. Например, чтобы отобразить диалоговое окно `alert()`, нужно назначить обработчик события запроса окна из JavaScript-кода и внутри обработчика создать окно и отобразить его. Назначить обработчик позволяет свойство `onAlert` из класса `WebEngine`. Методы доступа:

```
import javafx.scene.web.WebEvent;
public final void setOnAlert(EventHandler<WebEvent<String>> handler)
public final EventHandler<WebEvent<String>> getOnAlert()
public final ObjectProperty<EventHandler<WebEvent<String>>>
 onAlertProperty()
```

Внутри обработчика через параметр доступен объект события класса `WebEvent<T>`. Этот класс содержит метод `getData()`, с помощью которого можно получить данные. В нашем случае метод вернет строку, которую нужно отобразить в диалоговом окне. Формат метода:

```
public T getData()
```

Давайте в HTML-коде создадим ссылку, при переходе по которой вызывается метод `alert()`:

```
<a href="#"
 onclick="window.alert('Текст'); return false;">window.alert()
```

Теперь внутри программы обработаем вызов метода `alert()` и просто выведем строку в консоль:

```
webEngine.setOnAlert(event -> {
 System.out.println(event.getData());
});
```

Обработать вызов метода `confirm()` позволяет свойство `confirmHandler`. Методы доступа:

```
import javafx.util.Callback;
public final void setConfirmHandler(Callback<String, Boolean> handler)
public final Callback<String, Boolean> getConfirmHandler()
public final ObjectProperty<Callback<String, Boolean>>
 confirmHandlerProperty()
```

Интерфейс `Callback` содержит единственный метод `call()`, который будет вызван при наступлении события. Метод принимает один параметр и должен вернуть ка-



кое-либо значение. В нашем случае метод принимает строку, которую нужно отобразить в диалоговом окне, и должен вернуть значение `true`, если пользователь нажал кнопку **ОК**, или значение `false` — если пользователь нажал кнопку **Cancel**.

Давайте в HTML-коде создадим ссылку, при переходе по которой вызывается пользовательская функция `openConfirm()`:

```
window.confirm()
```

Код функции:

```
function openConfirm() {
 var result = document.getElementById("result");
 if (window.confirm('Текст')) {
 result.innerHTML = "Вы нажали ОК";
 }
 else {
 result.innerHTML = "Вы нажали Cancel";
 }
}
```

В этом примере мы выводим результат в элемент с идентификатором `result`:

```
<div id="result"></div>
```

Теперь внутри программы обработаем вызов метода `confirm()` и просто выведем строку в консоль:

```
webEngine.setConfirmHandler(param -> {
 System.out.println(param);
 return false; // true — кнопка ОК, false — кнопка Cancel
});
```

Обработать вызов метода `prompt()` позволяет свойство `promptHandler`. Методы дотупа:

```
import javafx.util.Callback;
import javafx.scene.web.PromptData;
public final void setPromptHandler(
 Callback<PromptData, String> handler)
public final Callback<PromptData, String> getPromptHandler()
public final ObjectProperty<Callback<PromptData, String>>
 promptHandlerProperty()
```

В этом случае метод `call()` принимает объект класса `PromptData` и должен возвращать строку, введенную пользователем в диалоговом окне, или значение `null`, если пользователь нажал кнопку **Cancel**. Класс `PromptData` содержит методы `getMessage()`, который возвращает текст подсказки пользователю, и `getDefaultValue()` — который возвращает значение по умолчанию. Форматы методов:

```
public final String getMessage()
public final String getDefaultValue()
```

Давайте в HTML-коде создадим ссылку, при переходе по которой вызывается пользовательская функция `openPrompt()`:

```
window.prompt()
```

Код функции:

```
function openPrompt() {
 var result = document.getElementById("result");
 var r = window.prompt('Текст', '0');
 if (r == null) {
 result.innerHTML = "Вы нажали Cancel";
 }
 else {
 result.innerHTML = "Вы ввели: " + r;
 }
}
```

В этом примере мы также выводим результат в элемент с идентификатором `result`. Теперь внутри программы обрабатываем вызов метода `prompt()` и просто выведем сообщение и значение по умолчанию в консоль:

```
webEngine.setPromptHandler(param -> {
 System.out.println(param.getMessage());
 System.out.println(param.getDefaultValue());
 return "0"; // String – кнопка ОК, null – кнопка Cancel
});
```

Обработать вызов метода `open()` позволяет свойство `createPopupHandler`. Методы доступа:

```
import javafx.util.Callback;
import javafx.scene.web.PopupFeatures;
public final void setCreatePopupHandler(
 Callback<PopupFeatures, WebEngine> handler)
public final Callback<PopupFeatures, WebEngine> getCreatePopupHandler()
public final ObjectProperty<Callback<PopupFeatures, WebEngine>>
 createPopupHandlerProperty()
```

В этом случае метод `call()` принимает объект класса `PopupFeatures` и должен возвращать объект класса `WebEngine`. Класс `PopupFeatures` содержит методы: `hasMenu()`, `hasToolbar()` и `hasStatus()`, с помощью которых можно узнать, нужно ли отображать меню, панели инструментов и строку состояния соответственно. Кроме того, класс `PopupFeatures` содержит метод `isResizable()`, который позволяет узнать, может ли окно изменять размеры. Форматы методов:

```
public final boolean hasMenu()
public final boolean hasToolbar()
public final boolean hasStatus()
public final boolean isResizable()
```

Давайте в HTML-коде создадим ссылку, при переходе по которой вызывается пользовательская функция `openMyWindow()`:

```
Открыть окно
```

**Код функции:**

```
var myWindow = null;
function openMyWindow(e) {
 myWindow = window.open("windowContent.html", "window1",
 "left=0,top=0,width=500,height=500");
}
```

В этом примере мы загружаем файл `windowContent.html` в новое окно и сохраняем ссылку на окно в глобальной переменной `myWindow`, чтобы в дальнейшем иметь возможность управлять окном, — например, закрыть его из программы. Теперь внутри программы обработаем вызов метода `open()`, просто выведем значения в консоль и вернем ссылку на первоначальный объект `WebEngine`:

```
webEngine.setCreatePopupHandler(param -> {
 System.out.println(param.hasMenu());
 System.out.println(param.hasToolBar());
 System.out.println(param.hasStatus());
 System.out.println(param.isResizable());
 // Создаем окно с WebView и возвращаем ссылку на WebEngine
 return webEngine;
});
```

**С помощью свойства `onVisibilityChanged` можно назначить обработчик события изменения видимости окна. Методы доступа:**

```
public final void setOnVisibilityChanged(
 EventHandler<WebEvent<Boolean>> handler)
public final EventHandler<WebEvent<Boolean>> getOnVisibilityChanged()
public final ObjectProperty<EventHandler<WebEvent<Boolean>>>
 onVisibilityChangedProperty()
```

**Внутри обработчика с помощью метода `getData()` можно получить статус окна:**

```
webEngine.setOnVisibilityChanged(event -> {
 System.out.println(event.getData());
});
```

**Свойство `onResized` позволяет назначить обработчик события изменения размеров и положения окна из скрипта. Методы доступа:**

```
import javafx.geometry.Rectangle2D;
public final void setOnResized(
 EventHandler<WebEvent<Rectangle2D>> handler)
public final EventHandler<WebEvent<Rectangle2D>> getOnResized()
public final ObjectProperty<EventHandler<WebEvent<Rectangle2D>>>
 onResizedProperty()
```

Внутри обработчика через параметр доступен объект класса `Rectangle2D`, который содержит размеры и положение окна. Давайте в HTML-коде создадим ссылку, при переходе по которой вызывается пользовательская функция `changeSize()`:

```
<a href="#"
 onclick="changeSize(); return false;">Изменить размеры окна
```

**Код функции:**

```
function changeSize() {
 window.resizeTo(300, 300);
}
```

Теперь внутри программы обработаем событие и просто выведем значения в консоль:

```
webEngine.setOnResized(event -> {
 Rectangle2D r = event.getData();
 System.out.println("getWidth(): " + r.getWidth());
 System.out.println("getHeight(): " + r.getHeight());
});
```

Назначить обработчик события изменения значения для строки состояния позволяет свойство `onStatusChanged`. Методы доступа:

```
public final void setOnStatusChanged(
 EventHandler<WebEvent<String>> handler)
public final EventHandler<WebEvent<String>> getOnStatusChanged()
public final ObjectProperty<EventHandler<WebEvent<String>>>
 onStatusChangedProperty()
```

Давайте в HTML-коде создадим ссылку, при переходе по которой вызывается пользовательская функция `changeStatus()`:

```
Изменить статус
```

**Код функции:**

```
function changeStatus() {
 window.status = "Текст в строке состояния";
}
```

Теперь внутри программы обработаем событие и просто выведем значение в консоль:

```
webEngine.setOnStatusChanged(event -> {
 System.out.println("Статус: " + event.getData());
});
```

Метод `executeScript()` позволяет выполнить код на языке JavaScript в контексте Web-страницы. Формат метода:

```
public Object executeScript(String script)
```

При нажатии кнопки выведем текст в элемент с идентификатором `result`:

```
Button button2 = new Button("Выполнить JavaScript-код");
button2.setOnAction(event -> {
 try {
 webEngine.executeScript(
 "document.getElementById(\"result\").innerHTML = \" +
 \"Привет из Java-кода\");
 }
 catch (netscape.javascript.JSException e) {
 System.out.println("Ошибка при выполнении скрипта");
 }
});
```

Метод `executeScript()` может возвращать результат выполнения скрипта. Если привести этот результат к типу `JSObject`, то появится возможность выполнять с объектом различные операции. Инструкция импорта:

```
// Модуль jdk.jsobject
import netscape.javascript.JSObject;
```

Обратиться к свойству объекта и получить его значение позволяет метод `getMember()` из класса `JSObject`. Формат метода:

```
public abstract Object getMember(String name) throws JSException
// value = obj.name
```

Изменить значение свойства объекта позволяет метод `setMember()` из класса `JSObject`. Формат метода:

```
public abstract void setMember(String name, Object value)
 throws JSException
// obj.name = value
```

Чтобы вызвать какой-либо метод объекта и передать ему параметры, нужно воспользоваться методом `call()` из класса `JSObject`. Формат метода:

```
public abstract Object call(String methodName, Object... args)
 throws JSException
// value = obj.methodName(args[0], ..., args[N])
```

С помощью методов `setSlot()` и `getSlot()` из класса `JSObject` можно задать и получить значение по индексу соответственно. Форматы методов:

```
public abstract void setSlot(int index, Object value)
 throws JSException
// obj[index] = value
public abstract Object getSlot(int index) throws JSException
// value = obj[index]
```

Полный список методов класса `JSObject` смотрите в документации.

При нажатии кнопки получим ссылку на объект `document`. Через эту ссылку получим заголовок Web-страницы, затем изменим заголовок и снова получим его, что-

бы удостовериться в успешно выполненной операции. Далее найдем элемент с идентификатором `result`, вызвав метод `getElementById()` объекта `document`, и получим его содержимое с помощью свойства `innerHTML`:

```
button3.setOnAction(event -> {
 try {
 Object obj = webEngine.executeScript("document");
 if (obj instanceof JSObject) {
 JSObject document = (JSObject) obj;
 System.out.println(document.getMember("title"));
 document.setMember("title", "Новый заголовок");
 System.out.println(document.getMember("title"));
 Object obj2 = document.call("getElementById", "result");
 if (obj2 instanceof JSObject) {
 JSObject el = (JSObject) obj2;
 System.out.println(el.getMember("innerHTML"));
 }
 }
 }
 catch (netscape.javascript.JSException e) {
 System.out.println("Ошибка при выполнении скрипта");
 }
});
```

Если нужно обрабатывать события какого-либо элемента Web-страницы в Java-программе, то можно назначить обработчик события с помощью метода `addEventListener()` из интерфейса `org.w3c.dom.events.EventTarget`. Формат метода:

```
public void addEventListener(String type, EventListener listener,
 boolean useCapture)
```

В параметре `type` указывается тип события в виде строки без префикса `on` — например, `click` для события `onclick`. В параметре `listener` задается ссылка на обработчик события. Давайте в HTML-коде создадим кнопку с идентификатором `button1`:

```
<input type="button" id="button1" name="button1" value="Нажми меня" />
```

Назначим обработчик для события нажатия кнопки при получении в процессе загрузки статуса `SUCCEEDED`:

```
// import org.w3c.dom.Document;
// import org.w3c.dom.Element;
// import org.w3c.dom.events.Event;
// import org.w3c.dom.events.EventListener;
// import org.w3c.dom.events.EventTarget;
webEngine.getLoadWorker().stateProperty().addListener(
 (obj, oldValue, newValue) -> {
 if (newValue == Worker.State.SUCCEEDED) {
 Document doc = webEngine.getDocument();
```

```

Element el = doc.getElementById("button1");
if (el != null) {
 ((EventTarget) el).addEventListener("click",
 new EventListener() {
 @Override
 public void handleEvent(Event e) {
 System.out.println("Нажата кнопка button1");
 }
 }, false);
}
});

```

Теперь при нажатии кнопки будет выводиться сообщение в окне консоли.

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH12_07_WebView`, приведенном в каталоге `CH12_text` сопровождающего книгу электронного архива (см. *приложение*).

## 12.8.4. Получение информации о просмотренных страницах

Зная, как можно выполнить скрипт на языке JavaScript из программы, мы можем вернуться к предыдущему просмотренному документу всего одной строкой кода:

```
webEngine.executeScript("history.back()");
```

Точно таким же способом мы можем воспользоваться методами `forward()` и `go()` объекта `history`, чтобы перейти к следующему документу или произвольному документу по индексу соответственно.

Получить доступ к списку истории просмотра документов позволяет метод `getHistory()` из класса `WebEngine`. Формат метода:

```

// Модуль javafx.web
import javafx.scene.web.WebHistory;
public WebHistory getHistory()

```

Класс `WebHistory` содержит следующие свойства:

- ❑ `currentIndex` — содержит индекс текущего документа в списке истории. Методы доступа:

```

public int getCurrentIndex()
public ReadOnlyIntegerProperty currentIndexProperty()

```

- ❑ `maxSize` — задает максимальный размер списка истории. Методы доступа:

```

public void setMaxSize(int value)
public int getMaxSize()
public IntegerProperty maxSizeProperty()

```

Класс `WebHistory` содержит следующие методы:

- ❑ `getEntries()` — возвращает ссылку на неизменяемый список истории. Формат метода:

```
import javafx.collections.ObservableList;
public ObservableList<WebHistory.Entry> getEntries()
```

Выведем информацию обо всех элементах списка:

```
WebHistory history = webEngine.getHistory();
ObservableList<WebHistory.Entry> list = history.getEntries();
for (WebHistory.Entry item : list) {
 System.out.println(item.getUrl());
 System.out.println(item.getTitle());
 System.out.println(item.getLastVisitedDate());
}
```

- ❑ `go()` — позволяет перемещаться по списку истории. В качестве значения параметра `offset` задается смещение относительно текущей позиции. Формат метода:

```
public void go(int offset) throws IndexOutOfBoundsException
```

Вернемся к предыдущему документу:

```
WebHistory history = webEngine.getHistory();
if (history.getCurrentIndex() > 0) {
 history.go(-1);
}
```

### **ЭЛЕКТРОННЫЙ АРХИВ**

Полный текст программы содержится в проекте `FX_CH12_07_WebView`, приведенном в каталоге `CH12_text` сопровождающего книгу электронного архива (см. приложение).





# ГЛАВА 13



## Списки

В JavaFX имеется широкий выбор компонентов, позволяющих отображать одномерный список строк в свернутом или развернутом состояниях. Списки поддерживают концепцию «модель/представление», позволяющую отделить данные от их отображения. В роли модели выступает объект, реализующий интерфейс `ObservableList<E>`, а в роли представления — компоненты `ChoiceBox<T>`, `ComboBox<T>` и `ListView<T>`.

### 13.1. Интерфейс *ObservableList<E>*

В предыдущих главах мы уже не раз встречались с интерфейсом `ObservableList<E>`, например, при добавлении узлов в список контейнера:

```
VBox pane = new VBox(15.0);
Button button = new Button("Нажми меня");
pane.getChildren().add(button);
```

Метод `getChildren()` как раз и возвращает объект, реализующий интерфейс `ObservableList<E>`:

```
public ObservableList<Node> getChildren()
```

Инструкция импорта:

```
// Модуль javafx.base
import javafx.collections.ObservableList;
```

Интерфейс `ObservableList<E>` наследует следующие интерфейсы:

```
Iterable<E>, Collection<E>, List<E>, Observable
```

Методы из интерфейсов `Iterable<E>`, `Collection<E>` и `List<E>` должны быть вам хорошо знакомы. Интерфейс `Observable` (см. *разд. 11.2.3*) содержит методы `addListener()` и `removeListener()`, которые позволяют назначить и удалить обработчик события изменения списка соответственно. Форматы методов:

```
public void addListener(InvalidationListener listener)
public void removeListener(InvalidationListener listener)
```

### 13.1.1. Создание списка

Создать объект, реализующий интерфейс `ObservableList<E>`, позволяют следующие статические методы из класса `FXCollections`:

```
// Модуль javafx.base
import javafx.collections.FXCollections;
```

❑ `observableArrayList()` — создает список на основе `ArrayList<E>`. Форматы метода:

```
public static <E> ObservableList<E> observableArrayList()
public static <E> ObservableList<E>
 observableArrayList(E... items)
public static <E> ObservableList<E>
 observableArrayList(Collection<? extends E> col)
public static <E> ObservableList<E>
 observableArrayList(Callback<E,Observable[]> extractor)
```

Первый формат создает пустой список. Второй формат создает список на основе массива или значений через запятую, а третий — на основе коллекции. Создадим список на основе коллекции:

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.<Integer>addAll(arr, 1, 2, 3);

ObservableList<Integer> list =
 FXCollections.<Integer>observableArrayList(arr);
list.addAll(5, 6);
System.out.println(list); // [1, 2, 3, 5, 6]
System.out.println(arr); // [1, 2, 3]
```

Четвертый формат полезен при работе с JavaFX-свойствами или с объектами, содержащими JavaFX-свойства, если мы хотим иметь возможность обрабатывать события изменения значения свойства:

```
DoubleProperty p = new SimpleDoubleProperty(20.0);
DoubleProperty p2 = new SimpleDoubleProperty(5.0);

ObservableList<DoubleProperty> list =
 FXCollections.<DoubleProperty>observableArrayList(
 new Callback<DoubleProperty, Observable[]>() {
 @Override
 public Observable[] call(DoubleProperty param) {
 return new Observable[] {param};
 }
 });
list.addListener((Observable observable) -> {
 System.out.println("Список изменился");
});
```

```
list.addAll(p, p2);
p2.setValue(33.0);
```

В этом примере мы получим сообщение `Список изменился` два раза. Первый раз при добавлении значений в список, а второй раз — при изменении значения свойства. Если бы мы в этом примере для создания списка использовали первый формат вместо четвертого, то получили бы только одно сообщение о добавлении значений в список;

- `observableList()` — создает объект на основе списка. Все изменения значений объекта отразятся на исходном списке. Если мы изменяем элементы через исходный список, то обработчики событий вызваны не будут. Форматы метода:

```
public static <E> ObservableList<E>
 observableList(List<E> list)
public static <E> ObservableList<E>
 observableList(List<E> list,
 Callback<E, Observable[]> extractor)
```

**Пример:**

```
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.<Integer>addAll(arr, 1, 2, 3);

ObservableList<Integer> list =
 FXCollections.<Integer>observableList(arr);
list.addAll(4, 5);
System.out.println(list); // [1, 2, 3, 4, 5]
System.out.println(arr); // [1, 2, 3, 4, 5]
```

- `emptyObservableList()` — создает пустой неизменяемый список, который удобно возвращать из метода при отсутствии значений. Попытка добавить значение в список приведет к исключению `UnsupportedOperationException`. Формат метода:

```
public static <E> ObservableList<E> emptyObservableList()
```

- `singletonObservableList()` — создает неизменяемый список из одного элемента. Попытка добавить или изменить значение приведет к исключению. Формат метода:

```
public static <E> ObservableList<E> singletonObservableList(E e)
```

- `unmodifiableObservableList()` — создает неизменяемую оболочку над указанным списком. Попытка изменить список через оболочку приведет к исключению. Формат метода:

```
public static <E> ObservableList<E>
 unmodifiableObservableList(ObservableList<E> list)
```

- `synchronizedObservableList()` — создает изменяемую синхронизированную оболочку над указанным списком. Формат метода:

```
public static <E> ObservableList<E>
 synchronizedObservableList(ObservableList<E> list)
```

- ❑ `checkedObservableList()` — создает изменяемую строго типизированную оболочку над указанным списком. Формат метода:

```
public static <E> ObservableList<E>
 checkedObservableList(ObservableList<E> list,
 Class<E> type)
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH13_01_ObservableList`, приведенном в каталоге `CH13_lists` сопровождающего книгу электронного архива (см. приложение).

## 13.1.2. Методы из интерфейса *ObservableList<E>*

Интерфейс `ObservableList<E>` содержит следующие основные методы:

- ❑ `addAll()` — добавляет указанные элементы в конец списка и возвращает значение `true`, если элементы были добавлены. Этим методом мы очень часто пользовались в примерах при добавлении нескольких узлов в контейнер. Формат метода:

```
public boolean addAll(E... elements)
```

Пример:

```
ObservableList<Integer> list =
 FXCollections.<Integer>observableArrayList(1, 2, 3);
System.out.println(list.addAll(4, 5, 6)); // true
for (Integer n : list) {
 System.out.print(n + " ");
} // 1 2 3 4 5 6
```

- ❑ `setAll()` — очищает список и добавляет указанные элементы. Возвращает значение `true`, если элементы были добавлены. Форматы метода:

```
public boolean setAll(E... elements)
public boolean setAll(Collection<? extends E> col)
```

Пример:

```
ObservableList<Integer> list =
 FXCollections.<Integer>observableArrayList(1, 2, 3);
System.out.println(list); // [1, 2, 3]
list.setAll(4, 5, 6);
list.forEach(elem -> System.out.print(elem + " "));
// 4 5 6
ArrayList<Integer> arr = new ArrayList<Integer>();
Collections.<Integer>addAll(arr, 8, 9, 10);
list.setAll(arr);
System.out.println(list); // [8, 9, 10]
```

- ❑ **remove()** — удаляет элементы от индекса `from` до индекса `to` (не включая элемент с этим индексом). Формат метода:

```
public void remove(int from, int to)
```

**Пример:**

```
ObservableList<Integer> list =
 FXCollections.<Integer>observableArrayList(1, 2, 3, 4, 5);
list.remove(1, 4);
System.out.println(list); // [1, 5]
```

- ❑ **removeAll()** — удаляет из списка указанные элементы. Возвращает значение `true`, если список изменился. Формат метода:

```
public boolean removeAll(E... elements)
```

**Пример:**

```
ObservableList<Integer> list =
 FXCollections.<Integer>observableArrayList(1, 2, 3, 4, 5, 1);
System.out.println(list.removeAll(1, 4, 3)); // true
System.out.println(list); // [2, 5]
System.out.println(list.removeAll(1, 4, 3)); // false
```

- ❑ **retainAll()** — оставляет в списке только указанные элементы, а остальные удаляет. Возвращает значение `true`, если список изменился. Формат метода:

```
public boolean retainAll(E... elements)
```

**Пример:**

```
ObservableList<Integer> list =
 FXCollections.<Integer>observableArrayList(1, 2, 3, 4, 5, 1);
System.out.println(list.retainAll(1, 4, 3)); // true
System.out.println(list); // [1, 3, 4, 1]
System.out.println(list.retainAll(1, 4, 3)); // false
```

- ❑ **filtered()** — создает над исходным списком неизменяемую оболочку, в которую попадут элементы, соответствующие условию `predicate`. Формат метода:

```
import javafx.collections.transformation.FilteredList;
default FilteredList<E> filtered(Predicate<E> predicate)
```

**Пример отбора элементов, значение которых больше трех:**

```
ObservableList<Integer> list =
 FXCollections.<Integer>observableArrayList(1, 2, 3, 4, 5, 1);
FilteredList<Integer> filter = list.filtered(elem -> elem > 3);
System.out.println(filter); // [4, 5]
```

- ❑ **sorted()** — создает над исходным списком неизменяемую оболочку, в которой элементы исходного списка отсортированы. Форматы метода:

```
import javafx.collections.transformation.SortedList;
default SortedList<E> sorted()
default SortedList<E> sorted(Comparator<E> comparator)
```

### Пример:

```
ObservableList<Integer> list =
 FXCollections.<Integer>observableArrayList(1, 5, 2, 4, 3, 1);
SortedList<Integer> sortedList = list.sorted();
System.out.println(sortedList); // [1, 1, 2, 3, 4, 5]
SortedList<Integer> sortedList2 =
 list.sorted((x, y) -> y.compareTo(x));
System.out.println(sortedList2); // [5, 4, 3, 2, 1, 1]
```

Не забывайте, что интерфейс `ObservableList<E>` наследует все методы из интерфейсов: `Iterable<E>`, `Collection<E>` и `List<E>`. Эти методы должны быть вам хорошо знакомы, поэтому в этой книге мы их приводить не будем.

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH13_01_ObservableList`, приведенном в каталоге `CH13_lists` сопровождающего книгу электронного архива (см. приложение).

## 13.1.3. Назначение и удаление обработчиков. Интерфейс `ListChangeListener<E>`

Назначить обработчик события изменения списка позволяет метод `addListener()`.

Форматы метода:

```
// Интерфейс Observable
import javafx.beans.InvalidationListener;
public void addListener(InvalidationListener listener)
// Интерфейс ObservableList<E>
import javafx.collections.ListChangeListener;
public void addListener(ListChangeListener<? super E> listener)
```

Удалить обработчик позволяет метод `removeListener()`. Форматы метода:

```
// Интерфейс Observable
public void removeListener(InvalidationListener listener)
// Интерфейс ObservableList<E>
public void removeListener(ListChangeListener<? super E> listener)
```

Напомним, что интерфейс `InvalidationListener` мы уже рассматривали в разд. 11.2.3.

Пример назначения и удаления обработчика:

```
ObservableList<Integer> list =
 FXCollections.<Integer>observableArrayList(1, 2, 3, 4);
InvalidationListener listener = (Observable observable) -> {
 System.out.println("Список изменился");
};
list.addListener(listener);
list.remove(0);
System.out.println(list); // [2, 3, 4]
list.removeListener(listener);
```

**Функциональный интерфейс** `ChangeListener<E>` содержит метод `onChanged()`.  
**Формат метода:**

```
public void onChanged(ListChangeListener.Change<? extends E> c)
```

**Пример назначения и удаления обработчика:**

```
ObservableList<Integer> list =
 FXCollections.<Integer>observableArrayList(1, 2, 3, 4);
ChangeListener<Integer> listener =
 new ListChangeListener<Integer>() {
 @Override
 public void onChanged(
 ListChangeListener.Change<? extends Integer> c) {
 System.out.println("Список изменился");
 }
 };
list.addListener(listener);
list.remove(0, 2);
System.out.println(list); // [3, 4]
list.removeListener(listener);
```

**Внутри обработчика** через параметр доступен экземпляр статического класса `ListChangeListener.Change<E>`, который содержит следующие методы:

❑ `getList()` — возвращает ссылку на список, который был изменен. **Формат метода:**

```
public ObservableList<E> getList()
```

❑ `next()` — переходит к следующему изменению и возвращает значение `true`, если изменение доступно, и `false` — если изменений больше нет. **Формат метода:**

```
public abstract boolean next()
```

Прежде чем работать с другими методами, необходимо предварительно вызвать метод `next()`. Чтобы обработать все изменения, обычно вызов метода `next()` указывается внутри обработчика в блоке условия цикла `while`:

```
while (c.next()) {
 // Обрабатываем изменения
}
```

❑ `reset()` — возвращает указатель в позицию перед первым изменением. Прежде чем работать с другими методами, необходимо предварительно вызвать метод `next()`, чтобы перейти к первому изменению. **Формат метода:**

```
public abstract void reset()
```

❑ `wasPermutated()` — возвращает значение `true`, если была произведена перестановка элементов списка, например, при сортировке. **Формат метода:**

```
public boolean wasPermutated()
```



- ❑ `wasAdded()` — возвращает значение `true`, если были добавлены элементы. **Формат метода:**  

```
public boolean wasAdded()
```
- ❑ `wasRemoved()` — возвращает значение `true`, если элементы были удалены. **Формат метода:**  

```
public boolean wasRemoved()
```
- ❑ `wasReplaced()` — возвращает значение `true`, если элементы были заменены. **Обратите внимание:** операция замены состоит из двух операций: удаления и добавления, поэтому методы `wasRemoved()` и `wasAdded()` также вернут значение `true`. **Формат метода:**  

```
public boolean wasReplaced()
```
- ❑ `wasUpdated()` — возвращает значение `true`, если элемент был изменен (изменено значение JavaFX-свойства). **Формат метода:**  

```
public boolean wasUpdated()
```
- ❑ `getFrom()` — возвращает начальный индекс диапазона изменений. **Формат метода:**  

```
public abstract int getFrom()
```
- ❑ `getTo()` — возвращает конечный индекс диапазона изменений (не включая элемент с этим индексом). **Обратите внимание:** в некоторых случаях (например, при удалении элементов) начальный индекс может быть равен конечному индексу. **Формат метода:**  

```
public abstract int getTo()
```
- ❑ `getPermutation()` — возвращает новый индекс элемента при перестановке по старому индексу `i`. **Формат метода:**  

```
public int getPermutation(int i)
```
- ❑ `getAddedSize()` — возвращает количество добавленных элементов. **Формат метода:**  

```
public int getAddedSize()
```
- ❑ `getAddedSubList()` — возвращает список добавленных элементов. **Формат метода:**  

```
public List<E> getAddedSubList()
```
- ❑ `getRemovedSize()` — возвращает количество удаленных элементов. **Формат метода:**  

```
public int getRemovedSize()
```
- ❑ `getRemoved()` — возвращает список удаленных элементов. **Формат метода:**  

```
public abstract List<E> getRemoved()
```

Пример обработки изменений списка приведен в листинге 13.1.

Листинг 13.1. Интерфейс `ListChangeListener<E>`

```
IntegerProperty p = new SimpleIntegerProperty(1);
IntegerProperty p2 = new SimpleIntegerProperty(2);
IntegerProperty p3 = new SimpleIntegerProperty(3);
IntegerProperty p4 = new SimpleIntegerProperty(4);
IntegerProperty p5 = new SimpleIntegerProperty(5);

ObservableList<IntegerProperty> list =
 FXCollections.<IntegerProperty>observableArrayList(
 (IntegerProperty param) -> {
 return new Observable[] {param};
 }
);
list.addListener(
 (ListChangeListener.Change<? extends IntegerProperty> c) -> {
 while (c.next()) {
 if (c.wasPermutated()) {
 System.out.println("Перестановка элементов");
 System.out.println("From " + c.getFrom());
 System.out.println("To " + c.getTo());
 for (int i = c.getFrom(), j = c.getTo(); i < j; i++) {
 System.out.println(i + " -> " + c.getPermutation(i));
 }
 }
 else if (c.wasReplaced()) {
 System.out.println("Замена элемента");
 System.out.println(c.getAddedSize());
 System.out.println(c.getAddedSubList());
 System.out.println(c.getRemovedSize());
 System.out.println(c.getRemoved());
 System.out.println("From " + c.getFrom());
 System.out.println("To " + c.getTo());
 }
 else if (c.wasAdded()) {
 System.out.println("Добавлены элементы");
 System.out.println(c.getAddedSize());
 System.out.println(c.getAddedSubList());
 System.out.println("From " + c.getFrom());
 System.out.println("To " + c.getTo());
 }
 else if (c.wasRemoved()) {
 System.out.println("Удалены элементы");
 System.out.println(c.getRemovedSize());
 System.out.println(c.getRemoved());
 }
 }
 }
);
```

```

 else if (c.wasUpdated()) {
 System.out.println("Обновлен элемент");
 System.out.println("From " + c.getFrom());
 System.out.println("To " + c.getTo());
 }
}
});
list.addAll(p, p2, p3, p4); // Добавлены элементы
list.remove(0, 2); // Удалены элементы
list.replaceAll(elem -> { // Замена элемента
 if (elem.get() == 4) {
 return p5;
 }
 return elem;
});
p5.set(1); // Обновлен элемент
list.sort((a, b) -> { // Перестановка элементов
 return a.getValue().compareTo(b.getValue());
});

```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH13_01_ObservableList`, приведенном в каталоге `CH13_lists` сопровождающего книгу электронного архива (см. приложение).

## 13.1.4. Методы из класса *FXCollections*

Для работы со списками предназначены следующие статические методы из класса `FXCollections`:

```

// Модуль javafx.base
import javafx.collections.FXCollections;

```

❑ `fill()` — заполняет список `list` значением `obj`. Формат метода:

```

public static <T> void fill(
 ObservableList<? super T> list, T obj)

```

Пример:

```

ObservableList<Integer> list =
 FXCollections.observableArrayList(1, 2, 3, 4, 5);
FXCollections.<Integer>fill(list, 8);
System.out.println(list); // [8, 8, 8, 8, 8]

```

❑ `concat()` — создает список на основе других списков. Формат метода:

```

public static <E> ObservableList<E>
 concat(ObservableList<E>... lists)

```

### Пример:

```
ObservableList<Integer> list =
 FXCollections.<Integer>observableArrayList(1, 2, 3);
ObservableList<Integer> list2 =
 FXCollections.<Integer>observableArrayList(4, 5);
@SuppressWarnings("unchecked")
ObservableList<Integer> list3 =
 FXCollections.<Integer>concat(list, list2);
System.out.println(list3); // [1, 2, 3, 4, 5]
```

- ❑ **copy()** — копирует элементы из списка src в начало списка dest. Формат метода:

```
public static <T> void copy(ObservableList<? super T> dest,
 List<? extends T> src)
```

### Пример:

```
ObservableList<Integer> list =
 FXCollections.<Integer>observableArrayList(1, 2, 3);
ArrayList<Integer> list2 = new ArrayList<Integer>();
Collections.<Integer>addAll(list2, 4, 5);
FXCollections.<Integer>copy(list, list2);
System.out.println(list); // [4, 5, 3]
```

- ❑ **shuffle()** — перемешивает элементы списка случайным образом. Форматы метода:

```
public static void shuffle(ObservableList<?> list)
public static void shuffle(ObservableList list, Random rnd)
```

### Пример:

```
ObservableList<Integer> list =
 FXCollections.<Integer>observableArrayList(1, 2, 3, 4, 5);
FXCollections.shuffle(list);
System.out.println(list); // [2, 4, 3, 5, 1]
```

- ❑ **reverse()** — меняет порядок следования элементов на противоположный, как бы переворачивает список. Формат метода:

```
public static void reverse(ObservableList list)
```

### Пример:

```
ObservableList<Integer> list =
 FXCollections.<Integer>observableArrayList(1, 2, 3, 4, 5);
FXCollections.reverse(list);
System.out.println(list); // [5, 4, 3, 2, 1]
```

- ❑ **rotate()** — сдвигает все элементы на указанное расстояние. Элементы из конца списка будут перемещены в начало. Метод как бы вращает элементы внутри списка, при этом не меняя их последовательность. Формат метода:

```
public static void rotate(ObservableList list, int distance)
```

**Пример:**

```
ObservableList<Integer> list =
 FXCollections.<Integer>observableArrayList(1, 2, 3, 4, 5);
FXCollections.rotate(list, 2);
System.out.println(list); // [4, 5, 1, 2, 3]
```

- ❑ **replaceAll()** — производит замену всех вхождений значения `oldVal` новым значением `newVal`. Метод возвращает значение `true`, если были замены в списке, и `false` — в противном случае. Формат метода:

```
public static <T> boolean replaceAll(ObservableList<T> list,
 T oldVal, T newVal)
```

**Пример:**

```
ObservableList<Integer> list =
 FXCollections.<Integer>observableArrayList(1, 2, 1, 4, 1);
FXCollections.<Integer>replaceAll(list, 1, 33);
System.out.println(list); // [33, 2, 33, 4, 33]
```

- ❑ **sort()** — сортирует список. Форматы метода:

```
public static <T extends Comparable<? super T>>
 void sort(ObservableList<T> list)
public static <T> void sort(ObservableList<T> list,
 Comparator<? super T> c)
```

**Пример сортировки списка в прямом и обратном порядке:**

```
ObservableList<Integer> list =
 FXCollections.<Integer>observableArrayList(5, 2, 3, 4, 1);
FXCollections.<Integer>sort(list);
System.out.println(list); // [1, 2, 3, 4, 5]
FXCollections.<Integer>sort(list, (a, b) -> b.compareTo(a));
System.out.println(list); // [5, 4, 3, 2, 1]
```

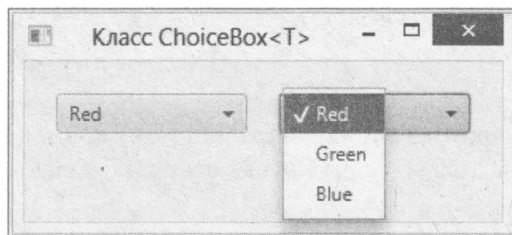
**ЭЛЕКТРОННЫЙ АРХИВ**

Полный текст программы содержится в проекте `FX_CH13_01_ObservableList`, приведенном в каталоге `CH13_lists` сопровождающего книгу электронного архива (см. приложение).

## 13.2. Класс *ChoiceBox<T>*: раскрывающийся список

Класс `ChoiceBox<T>` реализует раскрывающийся список с возможностью выбора только одного пункта (рис. 13.1). Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.ChoiceBox;
```



**Рис. 13.1.** Класс ChoiceBox<T>: раскрывающийся список

## Иерархия наследования:

Object — Node — Parent — Region — Control — ChoiceBox<T>

## Конструкторы класса:

```
ChoiceBox()
ChoiceBox(ObservableList<T> items)
```

Первый конструктор создает пустой объект, а второй конструктор позволяет сразу добавить список. Пример:

```
ObservableList<String> list =
 FXCollections.<String>observableArrayList();
for (int i = 1; i < 11; i++) {
 list.add("Пункт " + i);
}
ChoiceBox<String> choiceBox = new ChoiceBox<String>();
choiceBox.setItems(list);
choiceBox.getSelectionModel().select(1);
choiceBox.setPrefWidth(150.0);
```

## Получим выбранный пункт списка после нажатия кнопки:

```
Button button = new Button("Получить значение");
button.setOnAction(event -> {
 System.out.println(choiceBox.getValue());
});
```

## Управлять компонентом позволяют следующие свойства:

- ❑ **items** — содержит ссылку на список, элементы которого будут отображаться в компоненте. Методы доступа:

```
public final void setItems(ObservableList<T> value)
public final ObservableList<T> getItems()
public final ObjectProperty<ObservableList<T>> itemsProperty()
```

### Пример добавления элементов и выбора первого пункта списка:

```
ChoiceBox<String> choiceBox2 = new ChoiceBox<String>();
choiceBox2.getItems().addAll("Red", "Green", "Blue");
choiceBox2.setValue(choiceBox2.getItems().get(0));
```

- ❑ **value** — содержит значение выбранного элемента списка или значение null, если ничего не выбрано. Методы доступа:

```
public final void setValue(T value)
public final T getValue()
public ObjectProperty<T> valueProperty()
```

**Пример отслеживания изменения выбранного значения в списке (такой способ можно использовать, только если список содержит уникальные элементы):**

```
choiceBox.valueProperty().addListener(
 (obj, oldValue, newValue) -> {
 System.out.println("Изменение значения " + newValue);
 });
```

- **selectionModel** — содержит ссылку на модель выбора. Методы доступа:

```
public final void setSelectionModel(
 SingleSelectionModel<T> value)
public final SingleSelectionModel<T> getSelectionModel()
public final ObjectProperty<SingleSelectionModel<T>>
 selectionModelProperty()
```

**Пример отслеживания изменения выбранного индекса и значения в списке:**

```
SingleSelectionModel<String> sel = choiceBox.getSelectionModel();
sel.selectedIndexProperty().addListener(
 (obj, oldValue, newValue) -> {
 if (newValue == null || (int) newValue < 0) return;
 System.out.println("Изменение индекса " + newValue);
 System.out.println("Значение " +
 choiceBox.getItems().get((int) newValue));
 });
sel.selectedItemProperty().addListener(
 (obj, oldValue, newValue) -> {
 System.out.println("Изменение значения " + newValue);
 });
```

- **converter** — задает объект, который будет выполнять преобразование типов.

**Методы доступа:**

```
public final void setConverter(StringConverter<T> value)
public final StringConverter<T> getConverter()
public ObjectProperty<StringConverter<T>> converterProperty()
```

**Отообразим в компоненте список целых чисел:**

```
// import javafx.util.converter.IntegerStringConverter;
ChoiceBox<Integer> choiceBox3 = new ChoiceBox<Integer>();
choiceBox3.getItems().addAll(1, 2, 3, 4, 5);
choiceBox3.setValue(3);
choiceBox3.setConverter(new IntegerStringConverter());
```

- **showing** — содержит значение `true`, если список выбора показывается на экране, и `false` — если скрыт (отображается только выбранный пункт). Методы доступа:

```
public final boolean isShowing()
public final ReadOnlyBooleanProperty showingProperty()
```

**Пример обработки отображения и сокрытия списка выбора:**

```
choiceBox.showingProperty().addListener(
 (obj, oldValue, newValue) -> {
 System.out.println("showing " + newValue);
 });
```

- **onAction** — позволяет назначить обработчик события выбора нового пункта из списка. Свойство доступно, начиная с версии 8.60. Методы доступа:

```
public final void setOnAction(EventHandler<ActionEvent> value)
public final EventHandler<ActionEvent> getOnAction()
public final ObjectProperty<EventHandler<ActionEvent>>
 onActionProperty()
```

#### **ПРИМЕЧАНИЕ**

В версии 8.60 были добавлены свойства `onShowing`, `onShown`, `onHiding` и `onHidden`, а также статические константы: `ON_SHOWING`, `ON_SHOWN`, `ON_HIDING` и `ON_HIDDEN` для указания типа события.

Управлять отображением и сокрытием списка выбора из программы позволяют следующие методы из класса `ChoiceBox<T>`:

- **show()** — отображает список выбора на экране. Формат метода:

```
public void show()
```

- **hide()** — скрывает список выбора. Формат метода:

```
public void hide()
```

#### **ЭЛЕКТРОННЫЙ АРХИВ**

Полный текст программы содержится в проекте `FX_CH13_02_ChoiceBox`, приведенном в каталоге `CH13_lists` сопровождающего книгу электронного архива (см. *приложение*).

## **13.3. Класс `SingleSelectionModel<T>`: выбор одного элемента**

Абстрактный класс `SingleSelectionModel<T>` описывает модель выбора одного элемента из списка или таблицы. Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.SingleSelectionModel;
```

**Иерархия наследования:**

Object — SelectionModel<T> — SingleSelectionModel<T>

Класс `SingleSelectionModel<T>` наследует следующие свойства из класса `SelectionModel<T>`:



- ❑ `selectedIndex` — содержит индекс выбранного элемента или значение `-1`, означающее, что ничего не выбрано. Методы доступа:

```
public final int getSelectedIndex()
public final ReadOnlyIntegerProperty selectedIndexProperty()
```

- ❑ `selectedItem` — содержит значение выбранного элемента или значение `null`, означающее, что ничего не выбрано. Методы доступа:

```
public final T getSelectedItem()
public final ReadOnlyObjectProperty<T> selectedItemProperty()
```

#### Пример обработки изменения значений свойств:

```
ChoiceBox<String> choiceBox = new ChoiceBox<String>();
choiceBox.getItems().addAll("Red", "Green", "Blue");
choiceBox.getSelectionModel().select(1);

SingleSelectionModel<String> sel = choiceBox.getSelectionModel();
sel.selectedIndexProperty().addListener(
 (obj, oldValue, newValue) -> {
 System.out.println("Изменение индекса " + newValue);
 });
sel.selectedItemProperty().addListener(
 (obj, oldValue, newValue) -> {
 System.out.println("Изменение значения " + newValue);
 });
```

**Класс** `SingleSelectionModel<T>` наследует и реализует следующие методы из класса `SelectionModel<T>`:

- ❑ `select()` — выбирает элемент по индексу или находит первый элемент с указанным значением и выбирает его. Форматы метода:

```
public void select(int index)
public void select(T obj)
```

#### Пример:

```
choiceBox.getSelectionModel().select(0);
choiceBox.getSelectionModel().select("Red");
```

- ❑ `selectFirst()` — выбирает первый элемент списка. Формат метода:

```
public void selectFirst()
```

- ❑ `selectLast()` — выбирает последний элемент списка. Формат метода:

```
public void selectLast()
```

- ❑ `selectNext()` — выбирает следующий элемент списка. Формат метода:

```
public void selectNext()
```

- ❑ `selectPrevious()` — выбирает предыдущий элемент списка. Формат метода:

```
public void selectPrevious()
```

- ❑ `clearAndSelect()` — отменяет выбор всех элементов, а затем выбирает элемент с указанным индексом. Формат метода:  

```
public void clearAndSelect(int index)
```
- ❑ `clearSelection()` — отменяет выбор всех элементов или только элемента с указанным индексом. Форматы метода:  

```
public void clearSelection()
public void clearSelection(int index)
```
- ❑ `isEmpty()` — возвращает значение `true`, если нет выбранных элементов, и `false` — в противном случае. Формат метода:  

```
public boolean isEmpty()
```
- ❑ `isSelected()` — возвращает значение `true`, если элемент с указанным индексом выбран, и `false` — в противном случае. Формат метода:  

```
public boolean isSelected(int index)
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH13_03_SelectionModel`, приведенном в каталоге `CH13_lists` сопровождающего книгу электронного архива (см. приложение).

## 13.4. Класс *ComboBoxBase<T>*

Абстрактный класс `ComboBoxBase<T>` является базовым для классов: `ComboBox<T>`, `ColorPicker` и `DatePicker`. Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.ComboBoxBase;
```

Иерархия наследования:

`Object` — `Node` — `Parent` — `Region` — `Control` — `ComboBoxBase<T>`

В классе `ComboBoxBase<T>` определены следующие статические константы:

- ❑ `ON_SHOWING` — событие перед отображением списка выбора. Формат:  

```
public static final EventType<Event> ON_SHOWING
```
- ❑ `ON_SHOWN` — событие отображения списка выбора. Формат:  

```
public static final EventType<Event> ON_SHOWN
```
- ❑ `ON_HIDING` — событие перед сокрытием списка выбора. Формат:  

```
public static final EventType<Event> ON_HIDING
```
- ❑ `ON_HIDDEN` — событие сокрытия списка выбора. Формат:  

```
public static final EventType<Event> ON_HIDDEN
```

**Класс** `ComboBoxBase<T>` содержит следующие свойства:

- **value** — содержит значение выбранного элемента списка или значение `null`, если ничего не выбрано. Если список является редактируемым, то содержит значение, введенное пользователем в текстовое поле, или значение последнего выбранного элемента. Методы доступа:

```
public final void setValue(T value)
public final T getValue()
public ObjectProperty<T> valueProperty()
```

**Пример** отслеживания изменения выбранного значения в списке (такой способ можно использовать, только если список содержит уникальные элементы):

```
ObservableList<String> list =
 FXCollections.<String>observableArrayList();
for (int i = 1; i < 11; i++) {
 list.add("Пункт " + i);
}
ComboBox<String> comboBox = new ComboBox<String>();
comboBox.setItems(list);
comboBox.getSelectionModel().select(1);
comboBox.valueProperty().addListener
 (obj, oldValue, newValue) -> {
 System.out.println("Изменение значения " + newValue);
});
```

- **editable** — если свойство имеет значение `true`, то пользователь сможет ввести произвольное значение в текстовое поле, а если `false` — то сможет только выбрать пункт из списка. Значение по умолчанию: `false`. Методы доступа:

```
public final void setEditable(boolean value)
public final boolean isEditable()
public BooleanProperty editableProperty()
```

**Пример:**

```
ComboBox<String> comboBox2 = new ComboBox<String>();
comboBox2.getItems().addAll("Red", "Green", "Blue");
comboBox2.setEditable(true);
```

- **promptText** — позволяет задать текст подсказки, который будет отображаться в текстовом поле, если не выбрано значение. Методы доступа:

```
public final void setPromptText(String value)
public final String getPromptText()
public final StringProperty promptTextProperty()
```

**Пример:**

```
comboBox2.setPromptText("Текст подсказки");
```

- **showing** — содержит значение `true`, если список выбора показывается на экране, и `false` — если скрыт. Методы доступа:

```
public final boolean isShowing()
public ReadOnlyBooleanProperty showingProperty()
```

**Пример обработки отображения и сокрытия списка выбора:**

```
comboBox.showingProperty().addListener(
 (obj, oldValue, newValue) -> {
 System.out.println("showing " + newValue);
 });
```

- **armed** — содержит значение `true`, если пользователь нажал кнопку мыши над компонентом. Если указатель мыши покинет область компонента, или пользователь отпустит кнопку мыши над компонентом, то свойство будет иметь значение `false`. Методы доступа:

```
public final boolean isArmed()
public BooleanProperty armedProperty()
```

- **onAction** — позволяет назначить обработчик события выбора нового пункта из списка или ввода пользователем значения в текстовое поле и нажатия клавиши `<Enter>`. При этом изменяется значение свойства `value`. Методы доступа:

```
public final void setOnAction(EventHandler<ActionEvent> value)
public final EventHandler<ActionEvent> getOnAction()
public final ObjectProperty<EventHandler<ActionEvent>>
 onActionProperty()
```

- **onShowing** — позволяет назначить обработчик события `ON_SHOWING` (событие перед отображением списка выбора). Методы доступа:

```
public final void setOnShowing(EventHandler<Event> value)
public final EventHandler<Event> getOnShowing()
public final ObjectProperty<EventHandler<Event>>
 onShowingProperty()
```

- **onShown** — позволяет назначить обработчик события `ON_SHOWN` (событие отображения списка выбора). Методы доступа:

```
public final void setOnShown(EventHandler<Event> value)
public final EventHandler<Event> getOnShown()
public final ObjectProperty<EventHandler<Event>>
 onShownProperty()
```

- **onHiding** — позволяет назначить обработчик события `ON_HIDING` (событие перед сокрытием списка выбора). Методы доступа:

```
public final void setOnHiding(EventHandler<Event> value)
public final EventHandler<Event> getOnHiding()
public final ObjectProperty<EventHandler<Event>>
 onHidingProperty()
```

- **onHidden** — позволяет назначить обработчик события `ON_HIDDEN` (событие сокрытия списка выбора). Методы доступа:

```
public final void setOnHidden(EventHandler<Event> value)
public final EventHandler<Event> getOnHidden()
public final ObjectProperty<EventHandler<Event>>
 onHiddenProperty()
```

Управлять отображением и сокрытием списка выбора из программы позволяют следующие методы из класса `ComboBoxBase<T>`:

❑ `show()` — отображает список выбора на экране. Формат метода:

```
public void show()
```

❑ `hide()` — скрывает список выбора. Формат метода:

```
public void hide()
```

❑ `arm()` — имитирует нажатие кнопки мыши над компонентом. Свойство `armed` будет иметь значение `true`. Формат метода:

```
public void arm()
```

❑ `disarm()` — имитирует выход указателя мыши из области компонента или отпущения кнопки мыши над компонентом. Свойство `armed` будет иметь значение `false`. Формат метода:

```
public void disarm()
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH13_04_ComboBoxBase`, приведенном в каталоге `CH13_lists` сопровождающего книгу электронного архива (см. приложение).

## 13.5. Класс `ComboBox<T>`: раскрывающийся список

Класс `ComboBox<T>` реализует раскрывающийся список с возможностью ввода произвольного значения в текстовое поле (рис. 13.2). Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.ComboBox;
```

Иерархия наследования:

```
Object — Node — Parent — Region — Control — ComboBoxBase<T> — ComboBox<T>
```

Конструкторы класса:

```
ComboBox()
ComboBox(ObservableList<T> items)
```

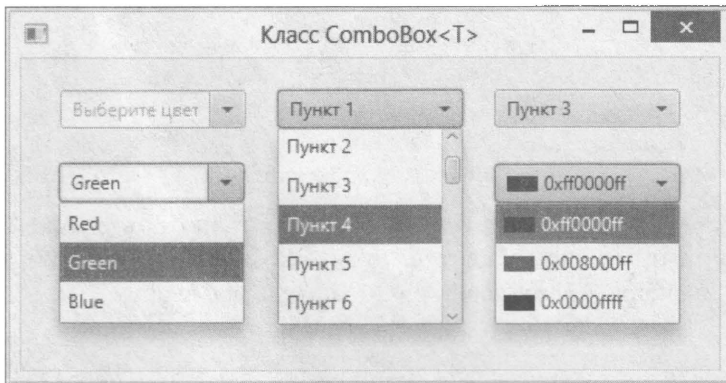
Пример создания раскрывающегося списка с возможностью ввода значения в текстовое поле и вывода текста подсказки:

```
ComboBox<String> comboBox = new ComboBox<String>();
comboBox.setEditable(true);
```

```
comboBox.getItems().addAll("Red", "Green", "Blue");
comboBox.setPromptText("Выберите цвет или введите название");
comboBox.setPrefWidth(250.0);
```

**Получим значение и индекс элемента списка при нажатии кнопки:**

```
Button button = new Button("Получить значение");
button.setOnAction(event -> {
 System.out.println(comboBox.getValue());
 System.out.println("Индекс " +
 comboBox.getSelectionModel().getSelectedIndex());
});
```



**Рис. 13.2.** Класс `ComboBox<T>`: раскрывающийся список

Если пользователь выберет пункт из списка, то мы получим текст элемента и его индекс внутри списка. Если ничего не введено, и никакой пункт не выбран, то значением будет `null` (в версии 8) или пустая строка (в версии 9 и выше), а индекс будет равен `-1`. Если пользователь ввел какое-либо значение в текстовое поле, и этого значения нет в списке, то метод `getValue()` вернет значение из текстового поля, а вот метод `getSelectedIndex()` — индекс последнего выбранного элемента (в версии 8) или значение `-1` (в версии 9 и выше).

Класс `ComboBox<T>` наследует все методы из класса `ComboBoxBase<T>` (см. *разд. 13.4*), а также из других базовых классов. Управлять компонентом позволяют следующие свойства:

- ❑ `items` — содержит ссылку на список, элементы которого будут отображаться в компоненте. Методы доступа:

```
public final void setItems(ObservableList<T> value)
public final ObservableList<T> getItems()
public ObjectProperty<ObservableList<T>> itemsProperty()
```

- ❑ `selectionModel` — содержит ссылку на модель выбора. Выбрать из списка можно только один элемент. Напомню, что класс `SingleSelectionModel<T>` мы уже рассматривали в *разд. 13.3*. Методы доступа:

```

public final void setSelectionModel(
 SingleSelectionModel<T> value)
public final SingleSelectionModel<T> getSelectionModel()
public final ObjectProperty<SingleSelectionModel<T>>
 selectionModelProperty()

```

- ❑ **converter** — задает объект, который будет выполнять преобразование типов. Методы доступа:

```

public final void setConverter(StringConverter<T> value)
public final StringConverter<T> getConverter()
public ObjectProperty<StringConverter<T>> converterProperty()

```

**Отообразим в компоненте список целых чисел:**

```

// import javafx.util.converter.IntegerStringConverter;
ComboBox<Integer> comboBox3 = new ComboBox<Integer>();
comboBox3.getItems().addAll(1, 2, 3, 4, 5);
comboBox3.getSelectionModel().selectFirst();
comboBox3.setConverter(new IntegerStringConverter());

```

- ❑ **visibleRowCount** — задает количество видимых пунктов в списке выбора. Если список содержит больше элементов, то автоматически будет отображена полоса прокрутки. Значение по умолчанию: 10. Методы доступа:

```

public final void setVisibleRowCount(int value)
public final int getVisibleRowCount()
public final IntegerProperty visibleRowCountProperty()

```

**Пример:**

```

ObservableList<String> list =
 FXCollections.<String>observableArrayList();
for (int i = 1; i < 21; i++) {
 list.add("Пункт " + i);
}
ComboBox<String> comboBox2 = new ComboBox<String>(list);
comboBox2.getSelectionModel().selectFirst();
comboBox2.setVisibleRowCount(5);

```

- ❑ **editor** — содержит ссылку на текстовое поле (компонент `TextField`), используемое для ввода значения, при условии, что свойство `editable` имеет значение `true`. Методы доступа:

```

public final TextField getEditor()
public final ReadOnlyObjectProperty<TextField> editorProperty()

```

**Получим текст, содержащийся в поле:**

```

System.out.println(comboBox.getEditor().getText());

```

- ❑ **placeholder** — позволяет указать узел, который будет отображаться в открывающемся списке выбора, когда не установлен список или список пустой. Методы доступа:

```
public final void setPlaceholder(Node value)
public final Node getPlaceholder()
public final ObjectProperty<Node> placeholderProperty()
```

### Пример отображения текстовой надписи:

```
ComboBox<String> comboBox4 = new ComboBox<String>();
comboBox4.setPlaceholder(new Label("Нет пунктов"));
comboBox4.setPrefWidth(150.0);
```

- ❑ **cellFactory** — позволяет создать пользовательский стиль каждой ячейки списка выбора. Например, мы можем слева от текста надписи отобразить небольшое изображение или любой другой узел (используя методы из класса `Labeled`). Методы доступа:

```
import javafx.util.Callback;
import javafx.scene.control.ListCell;
import javafx.scene.control.ListView;
public final void setCellFactory(
 Callback< ListView<T>, ListCell<T> > value)
public final Callback< ListView<T>, ListCell<T> >
 getCellFactory()
public ObjectProperty< Callback< ListView<T>, ListCell<T> > >
 cellFactoryProperty()
```

- ❑ **buttonCell** — задает пользовательский стиль ячейки, отображаемой слева от кнопки, когда список выбора свернут. В этой ячейке будет отображаться выбранный пункт списка. Методы доступа:

```
public final void setButtonCell(ListCell<T> value)
public final ListCell<T> getButtonCell()
public ObjectProperty<ListCell<T>> buttonCellProperty()
```

В качестве примера использования свойств `cellFactory` и `buttonCell` создадим список цветов (листинг 13.2). Каждая ячейка списка выбора слева будет содержать прямоугольник с заливкой указанным цветом, а справа — код цвета в виде строки (рис. 13.3).

#### Листинг 13.2. Свойства `cellFactory` и `buttonCell`

```
Callback< ListView<Color>, ListCell<Color> > cellFactory =
 new Callback< ListView<Color>, ListCell<Color> >() {
 @Override
 public ListCell<Color> call(ListView<Color> listView) {
 return new ListCell<Color>() {
 private final Rectangle rect = new Rectangle(50.0, 10.0);

 @Override
 protected void updateItem(Color color, boolean empty) {
 super.updateItem(color, empty);
```



```

 if (color == null || empty) {
 setGraphic(null);
 setText("");
 } else {
 rect.setFill(color);
 setGraphic(rect);
 setText(color.toString());
 }
 }
}

};

}

};

ComboBox<Color> comboBox = new ComboBox<Color>();
comboBox.getItems().addAll(Color.RED, Color.GREEN, Color.BLUE);
comboBox.setButtonCell(cellFactory.call(null));
comboBox.setCellFactory(cellFactory);
comboBox.getSelectionModel().selectFirst();
comboBox.setPrefWidth(250.0);

comboBox.valueProperty().addListener(
 (obj, oldValue, newValue) -> {
 System.out.println("Изменение значения " + newValue);
 });

```

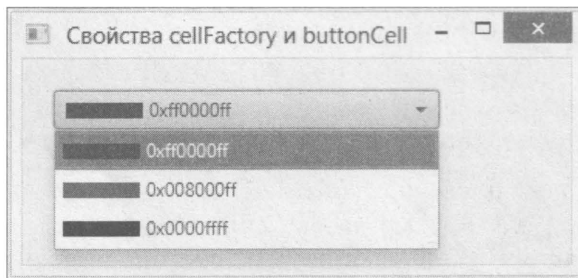


Рис. 13.3. Результат выполнения программы из листинга 13.2

Начиная с JavaFX 9, класс `ComboBox<T>` содержит следующие методы, которые можно использовать, если свойство `editable` имеет значение `true`:

- ❑ `commitValue()` — применяет ввод и отправляет данные на преобразование типов (см. *разд. 12.2.7*). Эквивалентно нажатию пользователем клавиши `<Enter>` внутри текстового поля. Формат метода:

```
public final void commitValue()
```

- ❑ `cancelEdit()` — отменяет ввод и вставляет в поле предыдущее значение. Эквивалентно нажатию пользователем клавиши `<Esc>` внутри текстового поля. Формат метода:

```
public final void cancelEdit()
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH13_04_ComboBoxBase`, приведенном в каталоге `CH13_lists` сопровождающего книгу электронного архива (см. приложение).

## 13.6. Класс *ColorPicker*: выбор цвета

Класс `ColorPicker` реализует раскрывающийся список, из которого можно выбрать цвет (рис. 13.4). Список содержит несколько predefined цветов, а также ссылку, при переходе по которой отобразится диалоговое окно **Custom Colors** (рис. 13.5), в котором можно выбрать произвольный цвет. Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.ColorPicker;
```

Иерархия наследования:

Object — Node — Parent — Region — Control — `ComboBoxBase<Color>` — `ColorPicker`

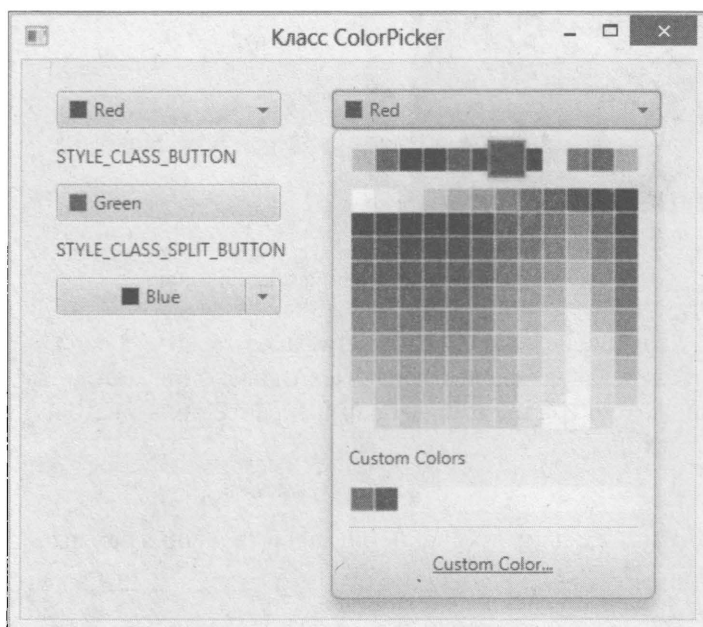


Рис. 13.4. Класс `ColorPicker`: выбор цвета

Конструкторы класса:

```
import javafx.scene.paint.Color;
ColorPicker()
ColorPicker(Color color)
```

Первый конструктор создает компонент с белым цветом по умолчанию, а второй конструктор позволяет указать произвольный цвет. Пример создания компонента и выбора красного цвета:

```
ColorPicker colorPicker = new ColorPicker();
colorPicker.setValue(Color.RED);
```

Назначим обработчик события изменения цвета и получим внутри обработчика новое значение:

```
colorPicker.setOnAction(event -> {
 Color color = colorPicker.getValue();
 System.out.println(color);
});
```

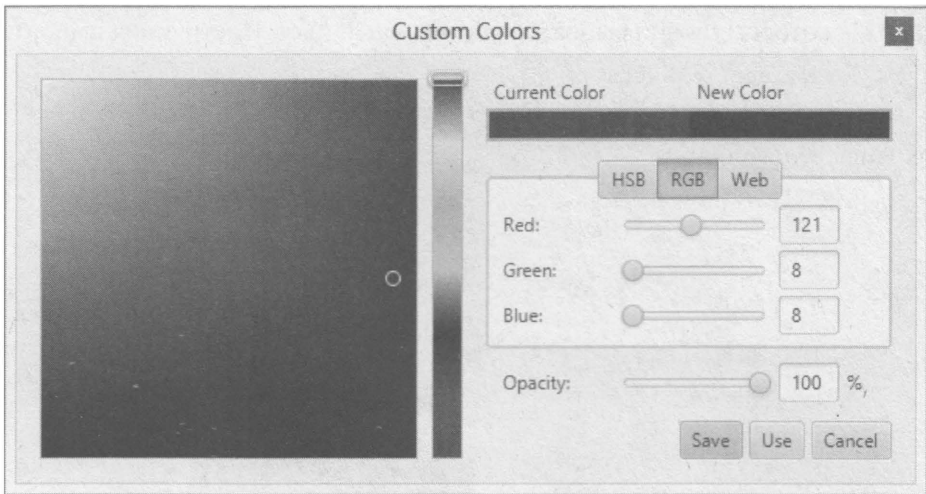


Рис. 13.5. Диалоговое окно Custom Colors

Класс `ColorPicker` наследует все методы из класса `ComboBoxBase<T>` (см. *разд. 13.4*), а также из других базовых классов и добавляет только один метод `getCustomColors()`, который возвращает список цветов, выбранных ранее пользователем в окне **Custom Colors**. Формат метода:

```
public final ObservableList<Color> getCustomColors()
```

В этот список мы можем добавить какие-либо цвета из программы:

```
colorPicker.getCustomColors().addAll(Color.CHOCOLATE, Color.BLUEVIOLET);
```

В классе `ColorPicker` определены две статические константы, задающие стиль компонента: `STYLE_CLASS_SPLIT_BUTTON` и `STYLE_CLASS_BUTTON`. Формат:

```
public static final String STYLE_CLASS_SPLIT_BUTTON
public static final String STYLE_CLASS_BUTTON
```

Выведем значения констант:

```
System.out.println(ColorPicker.STYLE_CLASS_SPLIT_BUTTON);
// split-button
System.out.println(ColorPicker.STYLE_CLASS_BUTTON);
// button
```

Изменим стиль компонента:

```
ColorPicker colorPicker3 = new ColorPicker(Color.RED);
colorPicker3.getStyleClass().add(ColorPicker.STYLE_CLASS_SPLIT_BUTTON);
colorPicker3.setPrefWidth(250.0);
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH13_04_ComboBoxBase`, приведенном в каталоге `CH13_lists` сопровождающего книгу электронного архива (см. приложение).

## 13.7. Класс *DatePicker*: выбор даты

Класс `DatePicker` реализует раскрывающийся список, из которого можно выбрать дату (рис. 13.6). По умолчанию имеется также возможность ввода даты пользователем в текстовое поле с клавиатуры. Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.DatePicker;
```

Иерархия наследования:

`Object` — `Node` — `Parent` — `Region` — `Control` — `ComboBoxBase<LocalDate>` — `DatePicker`

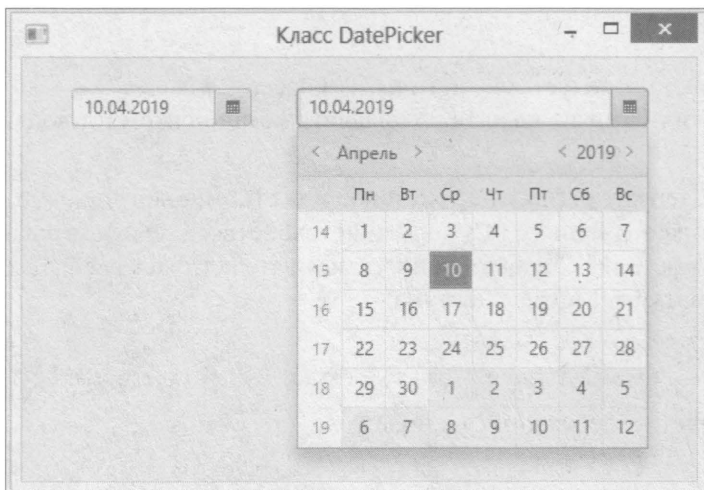


Рис. 13.6. Класс `DatePicker`: выбор даты

Конструкторы класса:

```
import java.time.LocalDate;
DatePicker()
DatePicker(LocalDate localDate)
```

Первый конструктор создает объект без выбранной даты, а второй конструктор позволяет указать дату по умолчанию. Пример создания компонента и выбора текущей даты:

```
DatePicker datePicker = new DatePicker();
datePicker.setValue(LocalDate.now());
```

Назначим обработчик события изменения даты и получим внутри обработчика новое значение:

```
datePicker.setOnAction(event -> {
 LocalDate date = datePicker.getValue();
 System.out.println(date);
});
```

Получим значение после нажатия кнопки (обратите внимание: если дата не выбрана, то метод `getValue()` вернет значение `null`):

```
// import java.time.format.DateTimeFormatter;
Button button = new Button("Получить значение");
button.setOnAction(event -> {
 LocalDate date = datePicker.getValue();
 if (date == null) {
 System.out.println("Дата не выбрана");
 return;
 }
 System.out.println(date);
 System.out.println(
 date.format(DateTimeFormatter.ofPattern("dd.MM.yyyy")));
});
```

Класс `DatePicker` наследует все методы из класса `ComboBoxBase<T>` (см. *разд. 13.4*), а также из других базовых классов. Управлять компонентом позволяют следующие свойства:

- `editor` — содержит ссылку на текстовое поле (компонент `TextField`), используемое для ввода значения. По умолчанию свойство `editable` в классе `DatePicker` имеет значение `true`, следовательно, пользователь может ввести дату в текстовое поле с клавиатуры. Методы доступа:

```
public final TextField getEditor()
public final ReadOnlyObjectProperty<TextField> editorProperty()
```

Получим текст, содержащийся в поле:

```
System.out.println(datePicker.getEditor().getText());
```

- `converter` — задает объект, который будет выполнять преобразование типов. Методы доступа:

```
public final void setConverter(
 StringConverter<LocalDate> value)
public final StringConverter<LocalDate> getConverter()
public final ObjectProperty<StringConverter<LocalDate>>
 converterProperty()
```

- `showWeekNumbers` — если свойство имеет значение `true`, то будут отображаться номера недель. Значение по умолчанию: `false`. Методы доступа:

```
public final void setShowWeekNumbers(boolean value)
public final boolean isShowWeekNumbers()
public final BooleanProperty showWeekNumbersProperty()
```

**Создадим компонент без возможности ввода пользователем даты с клавиатуры и отобразим номера недель:**

```
DatePicker datePicker2 = new DatePicker(LocalDate.now());
datePicker2.setEditable(false);
datePicker2.setShowWeekNumbers(true);
```

□ **chronology** — позволяет изменить календарную систему. Методы доступа:

```
import java.time.chrono.Chronology;
public final void setChronology(Chronology value)
public final Chronology getChronology()
public final ObjectProperty<Chronology> chronologyProperty()
```

**Выведем название календарной системы, используемой по умолчанию:**

```
Chronology ch = datePicker.getChronology();
System.out.println(ch.getId()); // ISO
System.out.println(ch.getCalendarType()); // iso8601
```

□ **dayCellFactory** — позволяет изменить отдельные ячейки календаря. Например, можно выделить праздничные дни или день рождения, сделать какую-либо дату недоступной для выбора, вывести всплывающую подсказку и т. д. Методы доступа:

```
import javafx.util.Callback;
import javafx.scene.control.DateCell;
public final void setDayCellFactory(
 Callback<DatePicker, DateCell> value)
public final Callback<DatePicker, DateCell> getDayCellFactory()
public final ObjectProperty<Callback<DatePicker, DateCell>>
 dayCellFactoryProperty()
```

**В качестве примера использования свойства `dayCellFactory` сделаем следующий день недоступным для выбора, предыдущий день — выделим зеленым цветом и добавим всплывающую подсказку, а также сделаем цвет шрифта красным для субботы и воскресенья:**

```
// import javafx.scene.control.Tooltip;
// import java.time.DayOfWeek;
// import javafx.util.Callback;
// import javafx.scene.control.DateCell;
Callback<DatePicker, DateCell> cellFactory =
 new Callback<DatePicker, DateCell>() {
 public DateCell call(DatePicker datePicker) {
 return new DateCell() {
 @Override
 public void updateItem(LocalDate item, boolean empty) {
 super.updateItem(item, empty);
```

```

 if (item == null || empty) return;
 if (item.equals(LocalDate.now().plusDays(1))) {
 setDisable(true);
 }
 else if (item.equals(LocalDate.now().minusDays(1))) {
 setTooltip(new Tooltip("Вчера"));
 setStyle("-fx-background-color: #008000;");
 }
 if (item.getDayOfWeek() == DayOfWeek.SATURDAY ||
 item.getDayOfWeek() == DayOfWeek.SUNDAY) {
 setTextFill(Color.RED);
 }
 }
}
};
}
};

DatePicker datePicker3 = new DatePicker(LocalDate.now());
datePicker3.setDayCellFactory(cellFactory);
datePicker3.setOnAction(event -> {
 System.out.println(datePicker.getValue());
});

```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH13_04_ComboBoxBase`, приведенном в каталоге `CH13_lists` сопровождающего книгу электронного архива (см. приложение).

## 13.8. Класс *ListView<T>*: список

Класс `ListView<T>` реализует постоянно отображающийся вертикальный (по умолчанию) или горизонтальный список (рис. 13.7), с возможностью выбора одного или нескольких элементов, а также с возможностью редактирования отдельного элемента пользователем. Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.ListView;
```

Иерархия наследования:

```
Object — Node — Parent — Region — Control — ListView<T>
```

### 13.8.1. Создание объекта

Конструкторы класса `ListView<T>`:

```
ListView()
ListView(ObservableList<T> items)
```

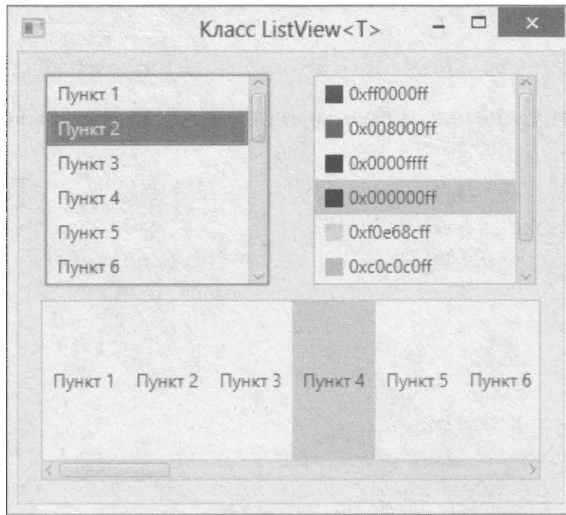


Рис. 13.7. Класс ListView&lt;T&gt;: список

Первый конструктор создает компонент с пустым списком и настройками по умолчанию, а второй конструктор позволяет сразу указать список, элементы которого будут отображаться в компоненте. Пример создания вертикального списка с возможностью выбора одного элемента:

```
ObservableList<String> list =
 FXCollections.<String>observableArrayList();
for (int i = 1; i < 21; i++) {
 list.add("Пункт " + i);
}

ListView<String> listView = new ListView<String>();
listView.getSelectionModel().setSelectionMode(SelectionMode.SINGLE);
listView.setItems(list);
listView.setOrientation(Orientation.VERTICAL);
listView.getSelectionModel().selectFirst();
```

После нажатия кнопки получим индекс и значение выбранного элемента:

```
Button button = new Button("Получить значение");
button.setOnAction(event -> {
 int index = listView.getSelectionModel().getSelectedIndex();
 System.out.println("Индекс " + index);
 if (index >= 0)
 System.out.println("Значение " +
 listView.getItems().get(index));
});
```

Класс ListView<T> поддерживает возможность выбора сразу нескольких элементов списка. Для выбора нескольких элементов нужно дополнительно удерживать нажатой клавишу <Ctrl> или <Shift>. Давайте заменим инструкцию:

```
listView.getSelectionModel().setSelectionMode(SelectionMode.SINGLE);
```



следующей инструкцией:

```
listView.getSelectionModel().setSelectionMode(SelectionMode.MULTIPLE);
```

Теперь после нажатия кнопки получим индексы и значения всех выбранных элементов:

```
button.setOnAction(event -> {
 ObservableList<Integer> arr =
 listView.getSelectionModel().getSelectedIndices();
 System.out.println("Индексы выбранных элементов:");
 for (Integer index : arr) {
 System.out.println(index);
 }
 ObservableList<String> arr2 =
 listView.getSelectionModel().getSelectedItems();
 System.out.println("Значения выбранных элементов:");
 for (String value : arr2) {
 System.out.println(value);
 }
});
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH13_08_ListView`, приведенном в каталоге `CH13_lists` сопровождающего книгу электронного архива (см. *приложение*).

## 13.8.2. Основные свойства и методы

Класс `ListView<T>` содержит следующие основные свойства:

- `items` — содержит ссылку на список, элементы которого будут отображаться в компоненте. Методы доступа:

```
public final void setItems(ObservableList<T> value)
public final ObservableList<T> getItems()
public final ObjectProperty<ObservableList<T>> itemsProperty()
```

- `selectionModel` — содержит ссылку на модель выбора. Выбрать из списка можно один элемент (режим `SelectionMode.SINGLE`) или несколько (режим `SelectionMode.MULTIPLE`). Методы доступа:

```
import javafx.scene.control.SelectionModel;
public final void setSelectionModel(
 SelectionModel<T> value)
public final SelectionModel<T> getSelectionModel()
public final ObjectProperty<SelectionModel<T>>
 selectionModelProperty()
```

- `orientation` — задает ориентацию компонента. В качестве значения параметра `value` можно указать константы: `VERTICAL` (значение по умолчанию) или `HORIZONTAL` из перечисления `Orientation`. Методы доступа:

```
import javafx.geometry.Orientation;
public final void setOrientation(Orientation value)
public final Orientation getOrientation()
public final ObjectProperty<Orientation> orientationProperty()
```

- ❑ **placeholder** — позволяет указать узел, который будет отображаться, когда список пустой. Методы доступа:

```
public final void setPlaceholder(Node value)
public final Node getPlaceholder()
public final ObjectProperty<Node> placeholderProperty()
```

#### Пример отображения текстовой надписи:

```
ListView<String> listView2 = new ListView<String>();
listView2.setPlaceholder(new Label("Нет пунктов"));
listView2.setPrefHeight(150.0);
```

- ❑ **fixedCellSize** — позволяет задать фиксированную высоту (для вертикальной ориентации) или ширину (для горизонтальной ориентации) ячеек списка. Значение по умолчанию:  $-1.0$  (означает, что размер ячеек может быть разным). Методы доступа:

```
public final void setFixedCellSize(double value)
public final double getFixedCellSize()
public final DoubleProperty fixedCellSizeProperty()
```

- ❑ **onScrollTo** — задает обработчик события прокрутки списка к элементу с указанным индексом. Это событие генерируется после вызова метода `scrollTo()`. Методы доступа:

```
import javafx.scene.control.ScrollToEvent;
public void setOnScrollTo(
 EventHandler<ScrollToEvent<Integer>> value)
public EventHandler<ScrollToEvent<Integer>> getOnScrollTo()
public ObjectProperty<EventHandler<ScrollToEvent<Integer>>>
 onScrollToProperty()
```

Форматы метода `scrollTo()`:

```
public void scrollTo(int index)
public void scrollTo(T object)
```

Первый формат вызывает прокрутку к элементу с указанным индексом, а второй формат — к первому элементу с указанным значением. Получить индекс элемента внутри обработчика события позволяет метод `getScrollTarget()` из класса `ScrollToEvent<T>`. Формат метода:

```
public T getScrollTarget()
```

#### Пример назначения обработчика и вывода индекса элемента:

```
listView.setOnScrollTo(event -> {
 System.out.println("onScrollTo " + event.getScrollTarget());
});
```

- `cellFactory` — позволяет создать пользовательский стиль каждой ячейки списка. Например, мы можем слева от текста надписи отобразить небольшое изображение или любой другой узел (используя методы из класса `Labeled`). Методы доступа:

```
import javafx.util.Callback;
import javafx.scene.control.ListCell;
public final void setCellFactory(
 Callback< ListView<T>, ListCell<T> > value)
public final Callback< ListView<T>, ListCell<T> >
 getCellFactory()
public final ObjectProperty<Callback<ListView<T>, ListCell<T>>>
 cellFactoryProperty()
```

В качестве примера использования свойства `cellFactory` создадим список цветов. Каждая ячейка списка слева будет содержать прямоугольник с заливкой указанным цветом, а справа — код цвета в виде строки:

```
Callback< ListView<Color>, ListCell<Color> > cellFactory =
 new Callback< ListView<Color>, ListCell<Color> >() {
 @Override
 public ListCell<Color> call(ListView<Color> listView) {
 return new ListCell<Color>() {
 private final Rectangle rect = new Rectangle(50.0, 10.0);

 @Override
 protected void updateItem(Color color, boolean empty) {
 super.updateItem(color, empty);
 if (color == null || empty) {
 setGraphic(null);
 setText("");
 } else {
 rect.setFill(color);
 setGraphic(rect);
 setText(color.toString());
 }
 }
 };
 }
 };

ListView<Color> listView = new ListView<Color>();
listView.getSelectionModel().setSelectionMode(SelectionMode.SINGLE);
listView.getItems().addAll(Color.RED, Color.GREEN, Color.BLUE);
listView.setOrientation(Orientation.VERTICAL);
listView.getSelectionModel().selectFirst();
listView.setCellFactory(cellFactory);
```

**ЭЛЕКТРОННЫЙ АРХИВ**

Полный текст программы содержится в проекте `FX_CH13_08_ListView`, приведенном в каталоге `CH13_lists` сопровождающего книгу электронного архива (см. *приложение*).

### 13.8.3. Класс *FocusModel<T>*: управление фокусом ввода

Свойство `focusModel` из класса `ListView<T>` содержит ссылку на объект, с помощью которого можно управлять фокусом ввода. Методы доступа:

```
import javafx.scene.control.FocusModel;
public final void setFocusModel(FocusModel<T> value)
public final FocusModel<T> getFocusModel()
public final ObjectProperty< FocusModel<T> > focusModelProperty()
```

Класс `FocusModel<T>` содержит следующие свойства:

□ `focusedIndex` — содержит индекс элемента, находящегося в фокусе ввода, или значение `-1`, если такого элемента нет. Методы доступа:

```
public final int getFocusedIndex()
public final ReadOnlyIntegerProperty focusedIndexProperty()
```

□ `focusedItem` — содержит значение элемента, находящегося в фокусе ввода, или `null`, если такого элемента нет. Методы доступа:

```
public final T getFocusedItem()
public final ReadOnlyObjectProperty<T> focusedItemProperty()
```

Пример назначения обработчика:

```
listView.getFocusModel().focusedIndexProperty().addListener(
 (obj, oldValue, newValue) -> {
 System.out.println("Index " + newValue);
 });
```

Пример получения значения после нажатия кнопки:

```
button.setOnAction(event -> {
 FocusModel<String> focusModel = listView.getFocusModel();
 System.out.println("Index " + focusModel.getFocusedIndex());
 System.out.println("Item " + focusModel.getFocusedItem());
});
```

Класс `FocusModel<T>` содержит следующие методы:

□ `focus()` — устанавливает фокус ввода на элемент с указанным индексом. Формат метода:

```
public void focus(int index)
```

Пример выделения элемента и установки фокуса ввода после нажатия кнопки:

```
button2.setOnAction(event -> {
 listView.requestFocus();
```

```
listView.getSelectionModel().clearAndSelect(5);
listView.setFocusModel().focus(5);
});
```

- ❑ `focusNext()` — устанавливает фокус ввода на следующий элемент (при условии, что он существует). Формат метода:

```
public void focusNext()
```

- ❑ `focusPrevious()` — устанавливает фокус ввода на предыдущий элемент (при условии, что он существует). Формат метода:

```
public void focusPrevious()
```

- ❑ `isFocused()` — возвращает значение `true`, если элемент с указанным индексом находится в фокусе ввода, и `false` — в противном случае. Формат метода:

```
public boolean isFocused(int index)
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH13_08_ListView`, приведенном в каталоге `CH13_lists` сопровождающего книгу электронного архива (см. приложение).

## 13.8.4. Редактирование элементов

Чтобы пользователь мог редактировать значения элементов списка, необходимо присвоить свойству `editable` из класса `ListView<T>` значение `true`. По умолчанию свойство имеет значение `false`. Методы доступа:

```
public final void setEditable(boolean value)
public final boolean isEditable()
public final BooleanProperty editableProperty()
```

Кроме того, нужно задать значение свойству `cellFactory`. Можно реализовать свой собственный способ редактирования или воспользоваться готовыми классами из пакета `javafx.scene.control.cell`. Например, классом `TextFieldListCell<T>`, который в режиме редактирования отображает внутри ячейки текстовое поле. Из этого класса нам понадобится статический метод `forListView()`. Форматы метода:

```
import javafx.scene.control.cell.TextFieldListCell;
public static Callback< ListView<String>, ListCell<String> >
 forListView()
public static <T> Callback< ListView<T>, ListCell<T> >
 forListView(StringConverter<T> converter)
```

Создадим редактируемый список:

```
ObservableList<String> list =
 FXCollections.<String>observableArrayList();
for (int i = 1; i < 21; i++) {
 list.add("Пункт " + i);
}
```

```

ListView<String> listView = new ListView<String>();
listView.setItems(list);
listView.setCellFactory(TextFieldListCell.forListView());
listView.getSelectionModel().selectFirst();
listView.getFocusModel().focus(0);
listView.setEditable(true);

```

**Класс `ListView<T>` содержит следующие статические методы, возвращающие типы событий редактирования списка:**

❑ `editAnyEvent()` — любое событие редактирования. Формат метода:

```

public static <T> EventType< ListView.EditEvent<T> >
 editAnyEvent()

```

❑ `editStartEvent()` — событие начала редактирования (пользователь выполнил двойной щелчок левой кнопкой мыши на пункте списка или выделил пункт и нажал клавишу `<Enter>`). Формат метода:

```

public static <T> EventType< ListView.EditEvent<T> >
 editStartEvent()

```

❑ `editCommitEvent()` — событие успешного окончания редактирования (например, пользователь ввел значение в текстовое поле и нажал клавишу `<Enter>`). Формат метода:

```

public static <T> EventType< ListView.EditEvent<T> >
 editCommitEvent()

```

❑ `editCancelEvent()` — событие отказа от редактирования (например, пользователь внутри текстового поля нажал клавишу `<Esc>` или выделил другой пункт списка в режиме редактирования). Формат метода:

```

public static <T> EventType< ListView.EditEvent<T> >
 editCancelEvent()

```

**Пример назначения обработчика и получения значений:**

```

listView.addEventHandler(
 ListView.<String>editCommitEvent(), event -> {
 System.out.println("editCommitEvent()");
 System.out.println(event.getIndex());
 System.out.println(event.getNewValue());
 System.out.println(event.getSource());
 });

```

**Внутри обработчика события через параметр `event` доступен экземпляр класса `ListView.EditEvent<T>`, который имеет следующие методы:**

❑ `getIndex()` — возвращает индекс редактируемого элемента. Формат метода:

```

public int getIndex()

```

❑ `getNewValue()` — возвращает новое значение при успешном окончании редактирования. Внутри обработчиков других событий метод возвращает значение `null`. Формат метода:

```

public T getNewValue()

```

- ❑ `getSource()` — возвращает ссылку на компонент `ListView<T>`. Формат метода:

```
public ListView<T> getSource()
```

Получить индекс редактируемого элемента можно также с помощью свойства `editingIndex` из класса `ListView<T>`. Если нет редактируемого элемента, то свойство будет иметь значение `-1`. Методы доступа:

```
public final int getEditingIndex()
public final ReadOnlyIntegerProperty editingIndexProperty()
```

Пример назначения обработчика:

```
listView.editingIndexProperty().addListener(
 (obj, oldValue, newValue) -> {
 System.out.println("editingIndex " + newValue);
 });
```

Управлять обработчиками событий редактирования позволяют также следующие свойства из класса `ListView<T>`:

- ❑ `onEditStart` — назначает обработчик события начала редактирования (пользователь выполнил двойной щелчок левой кнопкой мыши на пункте списка или выделил пункт и нажал клавишу `<Enter>`). Методы доступа:

```
public final void setOnEditStart(
 EventHandler< ListView.EditEvent<T> > value)
public final EventHandler< ListView.EditEvent<T> >
 getOnEditStart()
public final ObjectProperty<EventHandler<ListView.EditEvent<T>>>
 onEditStartProperty()
```

- ❑ `onEditCommit` — назначает обработчик события успешного окончания редактирования (например, пользователь ввел значение в текстовое поле и нажал клавишу `<Enter>`). Методы доступа:

```
public final void setOnEditCommit(
 EventHandler< ListView.EditEvent<T> > value)
public final EventHandler<ListView.EditEvent<T>>
 getOnEditCommit()
public final ObjectProperty<EventHandler<ListView.EditEvent<T>>>
 onEditCommitProperty()
```

**Обратите внимание:** если вы назначаете обработчик через свойство `onEditCommit`, то переопределяете обработчик по умолчанию, который изменяет значение элемента списка. Следовательно, нужно будет самостоятельно изменить значение, например, после проверки условия. Если просто нужно отследить событие, а не переопределять обработчик по умолчанию, то можно для назначения обработчика воспользоваться, например, методом `addEventHandler()` из класса `Node`. Давайте изменим значение элемента только в том случае, если поле заполнено:

```
listView.setOnEditCommit(event -> {
 if (event.getNewValue().length() != 0) {
 listView.getItems().set(event.getIndex(),
 event.getNewValue());
 }
});
```

- ❑ `onEditCancel` — назначает обработчик события отказа от редактирования (например, пользователь внутри текстового поля нажал клавишу <Esc> или выделил другой пункт списка в режиме редактирования). Методы доступа:

```
public final void setOnEditCancel(
 EventHandler<ListView.EditEvent<T>> value)
public final EventHandler<ListView.EditEvent<T>>
 getOnEditCancel()
public final ObjectProperty<EventHandler<ListView.EditEvent<T>>>
 onEditCancelProperty()
```

Перейти в режим редактирования элемента списка из программы позволяет метод `edit()` из класса `ListView<T>`. Формат метода:

```
public void edit(int itemIndex)
```

Пример:

```
listView.edit(0);
```

Если редактирование выполняется из программы способом, который не отслеживается, то нужно вызвать метод `refresh()`, чтобы обновить список. Метод доступен, начиная с версии 8.60. Формат метода:

```
public void refresh()
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH13_08_ListView`, приведенном в каталоге `CH13_lists` сопровождающего книгу электронного архива (см. *приложение*).

## 13.9. Класс *MultipleSelectionModel<T>*: выбор нескольких элементов

Абстрактный класс `MultipleSelectionModel<T>` описывает модель выбора одного или нескольких элементов из списка или таблицы. Для выбора нескольких элементов нужно дополнительно удерживать нажатой клавишу <Ctrl> или <Shift>. Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.MultipleSelectionModel;
```

Иерархия наследования:

```
Object — SelectionModel<T> — MultipleSelectionModel<T>
```



Управлять режимом выбора позволяет свойство `selectionMode` из класса `MultipleSelectionModel<T>`. В качестве значения параметра `value` можно указать константы: `SINGLE` (выбор только одного элемента) или `MULTIPLE` (выбор нескольких элементов) из перечисления `SelectionMode`. Методы доступа:

```
import javafx.scene.control.SelectionMode;
public final void setSelectionMode(SelectionMode value)
public final SelectionMode getSelectionMode()
public final ObjectProperty<SelectionMode> selectionModelProperty()
```

**Пример создания списка с возможностью выбора нескольких элементов:**

```
ObservableList<String> list =
 FXCollections.<String>observableArrayList();
for (int i = 1; i < 21; i++) {
 list.add("Пункт " + i);
}

ListView<String> listView = new ListView<String>(list);
listView.getSelectionModel().setSelectionMode(SelectionMode.MULTIPLE);
listView.getSelectionModel().selectFirst();
listView.setFocusModel().focus(0);
```

**Класс `MultipleSelectionModel<T>` наследует и реализует все методы из класса `SelectionModel<T>` (напомним, что эти методы мы уже рассматривали в *разд. 13.3*) и добавляет следующие методы:**

❑ `getSelectedIndices()` — возвращает список с индексами выбранных элементов. Список доступен только для чтения. Формат метода:

```
public abstract ObservableList<Integer> getSelectedIndices()
```

❑ `getSelectedItems()` — возвращает список со значениями выбранных элементов. Список доступен только для чтения. Формат метода:

```
public abstract ObservableList<T> getSelectedItems()
```

**Выведем индексы и значения всех выбранных элементов после нажатия кнопки:**

```
button.setOnAction(event -> {
 ObservableList<Integer> arr =
 listView.getSelectionModel().getSelectedIndices();
 System.out.println("Индексы выбранных элементов:");
 for (Integer index : arr) {
 System.out.println(index);
 }
 ObservableList<String> arr2 =
 listView.getSelectionModel().getSelectedItems();
 System.out.println("Значения выбранных элементов:");
 for (String value : arr2) {
 System.out.println(value);
 }
});
```

- ❑ `selectRange()` — выбирает диапазон от `start` до `end` (не включая элемент с этим индексом). Обратите внимание: предыдущий выбор не сбрасывается. Формат метода:

```
public void selectRange(int start, int end)
```

Пример выбора только элементов с индексами 0, 1, 2, 3 и 4:

```
listView.getSelectionModel().clearSelection();
listView.getSelectionModel().selectRange(0, 5);
```

- ❑ `selectIndices()` — выбирает один или несколько элементов по индексам. Обратите внимание: предыдущий выбор не сбрасывается. Формат метода:

```
public abstract void selectIndices(int index, int... indices)
```

Пример выбора только элементов с индексами 1, 3 и 5:

```
listView.getSelectionModel().clearSelection();
listView.getSelectionModel().selectIndices(1, 3, 5);
```

- ❑ `selectAll()` — выбирает все элементы списка. Формат метода:

```
public abstract void selectAll()
```

### **ЭЛЕКТРОННЫЙ АРХИВ**

Полный текст программы содержится в проекте `FX_CH13_03_SelectionModel`, приведенном в каталоге `CH13_lists` сопровождающего книгу электронного архива (см. приложение).



# ГЛАВА 14



## Таблицы и иерархические списки

Помимо компонентов, позволяющих отображать одномерный список строк, существует компонент `TableView<S>` для отображения табличных данных. Кроме того, с помощью компонентов: `TreeView<T>` и `TreeTableView<S>` можно отобразить данные, которые имеют очень сложную структуру, например иерархическую.

### 14.1. Класс `TableView<S>`: таблица

Класс `TableView<S>` реализует таблицу (рис. 14.1). Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.TableView;
```

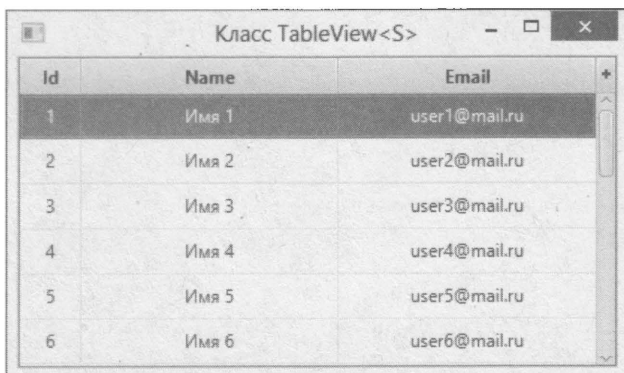
Иерархия наследования:

`Object` — `Node` — `Parent` — `Region` — `Control` — `TableView<S>`

#### 14.1.1. Создание таблицы

Конструкторы класса `TableView<S>`:

```
TableView()
TableView(ObservableList<S> items)
```



Id	Name	Email
1	Имя 1	user1@mail.ru
2	Имя 2	user2@mail.ru
3	Имя 3	user3@mail.ru
4	Имя 4	user4@mail.ru
5	Имя 5	user5@mail.ru
6	Имя 6	user6@mail.ru

Рис. 14.1. Класс `TableView<S>`: таблица

Первый конструктор создает пустую таблицу, а второй конструктор позволяет сразу указать список, элементы которого будут отображаться в таблице. Давайте создадим класс `User`, который будем использовать в дальнейших примерах (листинг 14.1).

**Листинг 14.1. Содержимое файла `src\application\User.java`**

```
package application;

import javafx.beans.property.IntegerProperty;
import javafx.beans.property.SimpleIntegerProperty;
import javafx.beans.property.SimpleStringProperty;
import javafx.beans.property.StringProperty;

public class User {
 private IntegerProperty id;
 private StringProperty name;
 private StringProperty email;

 public User(Integer id, String name, String email) {
 this.id = new SimpleIntegerProperty(this, "id", id);
 this.name = new SimpleStringProperty(this, "name", name);
 this.email = new SimpleStringProperty(this, "email", email);
 }

 public void setId(Integer value) {
 id.set(value);
 }

 public Integer getId() {
 return id.get();
 }

 public IntegerProperty idProperty() {
 return id;
 }

 public void setName(String value) {
 name.set(value);
 }

 public String getName() {
 return name.get();
 }

 public StringProperty nameProperty() {
 return name;
 }

 public void setEmail(String value) {
 email.set(value);
 }
}
```

```
public String getEmail() {
 return email.get();
}

public StringProperty emailProperty() {
 return email;
}

@Override
public String toString() {
 return "User [id=" + id.get() + ", name=" + name.get() +
 ", email=" + email.get() + "]";
}
}
```

Итак, класс `User` содержит три поля: `id` (идентификатор пользователя, например, в базе данных, целое число), `name` (имя пользователя, строка) и `email` (адрес электронной почты, строка). Мы ограничимся лишь тремя полями, хотя в реальной жизни их может быть гораздо больше. Все поля являются JavaFX-свойствами — это нужно, чтобы изменение значения поля автоматически привело к изменению значения в ячейке таблицы.

Теперь создадим список из нескольких объектов класса `User`:

```
ObservableList<User> list = FXCollections.<User>observableArrayList();
for (int i = 1; i < 21; i++) {
 list.add(new User(i, "Имя " + i, "user" + i + "@mail.ru"));
}
```

Добавить этот список в компонент позволяет конструктор класса `TableView<S>`, а также свойство `items`. Методы доступа:

```
public final void setItems(ObservableList<S> value)
public final ObservableList<S> getItems()
public final ObjectProperty<ObservableList<S>> itemsProperty()
```

Пример добавления списка и выделения первой строки:

```
TableView<User> tableView = new TableView<User>();
tableView.setItems(list);
tableView.getSelectionModel().selectFirst();
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH14_01_TableView`, приведенном в каталоге `CH14_tables` сопровождающего книгу электронного архива (см. приложение).

## 14.1.2. Класс `TableColumn<S, T>`: создание столбцов

Просто добавить список в компонент недостаточно. Предварительно нужно описать столбцы таблицы и назначить правила отображения значений в ячейках столбца. Описание столбцов таблицы выполняется с помощью класса `TableColumn<S, T>`. Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.TableColumn;
```

### Иерархия наследования:

Object — TableColumnBase<S, T> — TableColumn<S, T>

### Конструкторы класса:

```
TableColumn()
TableColumn(String text)
```

**Параметр text задает название столбца, которое отобразится в строке заголовка. Изменить или получить это название в дальнейшем позволяет свойство text из класса TableColumnBase<S, T>. Методы доступа:**

```
public final void setText(String value)
public final String getText()
public final StringProperty textProperty()
```

**Перед названием столбца в строке заголовка можно отобразить узел — например, компонент ImageView с изображением. Для этого нужно присвоить значение свойству graphic из класса TableColumnBase<S, T>. Методы доступа:**

```
public final void setGraphic(Node value)
public final Node getGraphic()
public final ObjectProperty<Node> graphicProperty()
```

### Пример:

```
columnId.setGraphic(new ImageView(createImage(12, 12, Color.RED)));
```

**Задать правило отображения значений в ячейках столбца позволяет свойство cellValueFactory из класса TableColumn<S, T>. Методы доступа:**

```
import javafx.util.Callback;
import javafx.scene.control.TableColumn.CellDataFeatures;
import javafx.beans.value.ObservableValue;
public final void setCellValueFactory(
 Callback< TableColumn.CellDataFeatures<S, T>,
 ObservableValue<T> > value)
public final Callback< TableColumn.CellDataFeatures<S, T>,
 ObservableValue<T> > getCellValueFactory()
public final ObjectProperty< Callback<
 TableColumn.CellDataFeatures<S, T>,
 ObservableValue<T> > > cellValueFactoryProperty()
```

**Первый способ создать необходимый объект заключается в использовании класса PropertyValueFactory<S, T>. Инструкция импорта:**

```
// Модуль javafx.controls
import javafx.scene.control.cell.PropertyValueFactory;
```

### Конструктор класса:

```
PropertyValueFactory(String property)
```

Параметр `property` задает название JavaFX-свойства в виде строки. Внутри класса должен быть метод с названием `<property>Property()`, который возвращает ссылку на JavaFX-свойство, — например, `nameProperty()` для свойства `name`.

Давайте создадим столбец и свяжем его со свойством `name` из класса `User`:

```
TableColumn<User, String> columnName =
 new TableColumn<User, String>("Name");
columnName.setCellValueFactory(
 new PropertyValueFactory<User, String>("name"));
```

Второй способ заключается в реализации интерфейса `Callback<P, R>`. Внутри метода `call()` нам следует вернуть ссылку на JavaFX-свойство, значение которого должно отображаться в ячейке. Так как метод `call()` возвращает объект `ObservableValue<T>`, то будет автоматически добавлен обработчик события изменения значения свойства. Поэтому любое изменение значения свойства внутри класса приведет к изменению значения внутри ячейки таблицы. Давайте создадим еще один столбец и свяжем его со свойством `email` из класса `User`:

```
TableColumn<User, String> columnEmail =
 new TableColumn<User, String>("Email");
columnEmail.setCellValueFactory(
 new Callback<CellDataFeatures<User, String>,
 ObservableValue<String> >() {
 public ObservableValue<String>
 call(CellDataFeatures<User, String> cellData) {
 return cellData.getValue().emailProperty();
 }
 });
```

Вместо такого громоздкого кода можно использовать лямбда-выражение:

```
TableColumn<User, String> columnEmail =
 new TableColumn<User, String>("Email");
columnEmail.setCellValueFactory(
 cellData -> cellData.getValue().emailProperty());
```

Класс `TableColumn.CellDataFeatures<S, T>` содержит следующие методы:

❑ `getValue()` — возвращает ссылку на текущий элемент списка (в нашем случае — на экземпляр класса `User`). Формат метода:

```
public S getValue()
```

❑ `getTableView()` — возвращает ссылку на объект таблицы. Формат метода:

```
public TableView<S> getTableView()
```

❑ `getTableColumn()` — возвращает ссылку на объект столбца. Формат метода:

```
public TableColumn<S,T> getTableColumn()
```

При использовании свойств числовых типов (например, `IntegerProperty` и `DoubleProperty`) следует учитывать, что они реализуют интерфейс



`ObservableValue<Number>`, а не интерфейсы `ObservableValue<Integer>` и `ObservableValue<Double>`. Поэтому внутри обработчиков событий мы вынуждены выполнять приведение типов:

```
IntegerProperty property = new SimpleIntegerProperty(10);
property.addListener((obj, oldValue, newValue) -> {
 Integer n = (Integer) newValue;
 System.out.println("Изменение n = " + n);
});
property.set(15);
```

Чтобы избежать приведения типов, можно при назначении обработчика воспользоваться методом `asObject()`, который создаст связанное объектное свойство типа `ObjectProperty<Integer>`:

```
IntegerProperty property = new SimpleIntegerProperty(10);
property.asObject().addListener((obj, oldValue, newValue) -> {
 Integer n = newValue;
 System.out.println("Изменение n = " + n);
});
property.set(15);
```

Давайте этим способом зададим правило для столбца с идентификатором, так оно у нас имеет тип `IntegerProperty`:

```
TableColumn<User, Integer> columnId =
 new TableColumn<User, Integer>("Id");
columnId.setCellValueFactory(
 cellData -> cellData.getValue().idProperty().asObject());
```

Этот код будет работать, но в нем кроется подвох. Проблема заключается в том, что при каждом обращении к свойству будет создаваться новый объект. Если этих объектов будет много, например сто тысяч, то код станет выполняться очень медленно. Представьте себе ситуацию сортировки ста тысяч значений, при которой создается сто тысяч новых объектов, — сортировка будет выполняться очень долго. Поэтому лучше изначально создавать свойство с типом `ObjectProperty<Integer>` или при создании столбца указывать тип `Number` вместо типа `Integer`. Давайте перделаем предыдущий пример:

```
TableColumn<User, Number> columnId =
 new TableColumn<User, Number>("Id");
columnId.setCellValueFactory(
 cellData -> cellData.getValue().idProperty());
```

Если по каким-либо причинам мы не хотим или не можем использовать `JavaFX`-свойства внутри класса, то следует либо воспользоваться классом `PropertyValueFactory<S, T>`, либо обернуть значение поля в объект класса `ReadOnlyObjectWrapper<T>`. В первом случае, если в классе существует метод с названием `<property>Property()`, то будет вызываться именно он, иначе выполняется

поиск методов с названиями: `get<Property>()` или `is<Property>()`. Если методов с такими названиями в классе нет, то генерируется исключение. Во втором случае значение поля будет доступно только для чтения. Следует учитывать, что при использовании обычных типов данных изменение значения поля внутри класса не приведет к автоматическому изменению значения в ячейке таблицы. Отслеживать все изменения нужно будет самостоятельно. Пример:

```
// import javafx.beans.property.ReadOnlyObjectWrapper;
TableColumn<User, String> columnEmail =
 new TableColumn<User, String>("Email");
columnEmail.setCellValueFactory(cellData -> {
 return new ReadOnlyObjectWrapper<String>(
 cellData.getValue().getEmail());
});
```

Столбцы мы описали, теперь их нужно добавить в список столбцов компонента `TableView<S>`. Получить ссылку на этот список позволяет метод `getColumns()` из класса `TableView<S>`. Формат метода:

```
public final ObservableList< TableColumn<S, ?> > getColumns()
```

Пример добавления столбцов:

```
tableView.getColumns().add(columnId);
tableView.getColumns().add(columnName);
tableView.getColumns().add(columnEmail);
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH14_01_TableView`, приведенном в каталоге `CH14_tables` сопровождающего книгу электронного архива (см. приложение).

## 14.1.3. Объединение столбцов в группу

Класс `TableColumn<S, T>` также содержит метод `getColumns()`, но он используется для создания группы столбцов. Формат метода:

```
public final ObservableList< TableColumn<S, ?> > getColumns()
```

Пример создания группы столбцов:

```
TableView<User> tableView2 = new TableView<User>();
tableView2.setPlaceholder(new Label("Нет элементов"));
tableView2.setPrefHeight(150.0);

TableColumn<User, Integer> columnGroup =
 new TableColumn<>("Группа столбцов");
TableColumn<User, String> column1 = new TableColumn<>("Name");
column1.setPrefWidth(200.0);
TableColumn<User, String> column2 = new TableColumn<>("Email");
column2.setPrefWidth(200.0);
```

```
columnGroup.getColumns().add(column1);
columnGroup.getColumns().add(column2);

tableView2.getColumns().add(columnGroup);
```

В этом примере мы изменили значение свойства `placeholder` из класса `TableView<S>`, которое задает узел, отображающийся, если компонент не содержит списка или список пуст. Методы доступа:

```
public final void setPlaceholder(Node value)
public final Node getPlaceholder()
public final ObjectProperty<Node> placeholderProperty()
```

Получить ссылку на объект группы через объект столбца позволяет свойство `parentColumn` из класса `TableColumnBase<S, T>`. Если столбец не принадлежит группе, то свойство содержит значение `null`. Методы доступа:

```
public final TableColumnBase<S, ?> getParentColumn()
public final ReadOnlyObjectProperty<TableColumnBase<S, ?> >
 parentColumnProperty()
```

**Пример:**

```
System.out.println(column1.getParentColumn().getText());
// Группа столбцов
System.out.println(columnGroup.getParentColumn()); // null
```

Для получения ссылки на объект таблицы через объект столбца предназначено свойство `tableView` из класса `TableColumn<S, T>`. Если столбец не принадлежит таблице (например, объект столбца создан, но не добавлен в список столбцов), то свойство содержит значение `null`. Методы доступа:

```
public final TableView<S> getTableView()
public final ReadOnlyObjectProperty<TableView<S>> tableViewProperty()
```

**Пример:**

```
System.out.println(columnGroup.getTableView());
// TableView@7a8fce48[styleClass=table-view]
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH14_01_TableView`, приведенном в каталоге `CH14_tables` сопровождающего книгу электронного архива (см. приложение).

## 14.1.4. Управление размерами столбцов

Управлять шириной столбца позволяют следующие свойства из класса `TableColumnBase<S, T>`:

□ `width` — содержит текущую ширину столбца. Методы доступа:

```
public final double getWidth()
public final ReadOnlyDoubleProperty widthProperty()
```

- ❑ **prefWidth** — предпочтительная ширина столбца. Методы доступа:

```
public final void setPrefWidth(double value)
public final double getPrefWidth()
public final DoubleProperty prefWidthProperty()
```

- ❑ **minWidth** — минимальная ширина столбца. Методы доступа:

```
public final void setMinWidth(double value)
public final double getMinWidth()
public final DoubleProperty minWidthProperty()
```

- ❑ **maxWidth** — максимальная ширина столбца. Методы доступа:

```
public final void setMaxWidth(double value)
public final double getMaxWidth()
public final DoubleProperty maxWidthProperty()
```

- ❑ **resizable** — если свойство имеет значение `true`, то ширину столбца можно менять (пользователь может изменить ширину столбца, взявшись мышью за границу в строке заголовка столбцов), а если `false` — то ширину изменить будет нельзя. Методы доступа:

```
public final void setResizable(boolean value)
public final boolean isResizable()
public final BooleanProperty resizableProperty()
```

Укажем ширину столбца с идентификатором:

```
columnId.setMaxWidth(80.0);
columnId.setMinWidth(40.0);
columnId.setPrefWidth(80.0);
```

Управлять размерами позволяют следующие свойства из класса `TableView<S>`:

- ❑ **columnResizePolicy** — задает политику изменения ширины столбцов. Методы доступа:

```
import javafx.scene.control.TableView.ResizeFeatures;
import javafx.util.Callback;
public final void setColumnResizePolicy(
 Callback<TableView.ResizeFeatures, Boolean> callback)
public final Callback<TableView.ResizeFeatures, Boolean>
 getColumnResizePolicy()
public final ObjectProperty< Callback<
 TableView.ResizeFeatures, Boolean> >
 columnResizePolicyProperty()
```

Можно самостоятельно реализовать интерфейс `Callback` или воспользоваться следующими статическими константами из класса `TableView<S>`:

- `UNCONSTRAINED_RESIZE_POLICY` — после изменения ширины столбца сдвигает все последующие столбцы на это расстояние вправо или влево (значение по умолчанию);

- `CONSTRAINED_RESIZE_POLICY` — при изменении ширины столбца суммарная ширина всех столбцов будет равна ширине таблицы.

Пример:

```
tableView.setColumnResizePolicy(
 TableView.CONSTRAINED_RESIZE_POLICY);
tableView.setPrefWidth(300.0);
```

- ❑ `fixedCellSize` — позволяет задать фиксированную высоту ячеек таблицы. Если свойство имеет отрицательное значение или 0, то фиксированная высота не используется. Методы доступа:

```
public final void setFixedCellSize(double value)
public final double getFixedCellSize()
public final DoubleProperty fixedCellSizeProperty()
```

Изменить ширину указанного столбца на некоторую величину `delta` позволяет метод `resizeColumn()` из класса `TableView<S>`. Формат метода:

```
public boolean resizeColumn(TableColumn<S, ?> column, double delta)
```

Уменьшим ширину столбца с именем пользователя на 5 px:

```
tableView.resizeColumn(columnName, -5.0);
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH14_01_TableView`, приведенном в каталоге `CH14_tables` сопровождающего книгу электронного архива (см. *приложение*).

## 14.1.5. Изменение видимости столбца

Управлять видимостью столбца на экране позволяет свойство `visible` из класса `TableColumnBase<S, T>`. Если свойство имеет значение `true`, то столбец отображается на экране, а если `false` — то он будет скрыт. Методы доступа:

```
public final void setVisible(boolean value)
public final boolean isVisible()
public final BooleanProperty visibleProperty()
```

Изменим видимость столбца с идентификатором после нажатия кнопки:

```
button.setOnAction(event -> {
 columnId.setVisible(!columnId.isVisible());
});
```

Если нужно, чтобы пользователь самостоятельно мог скрывать и отображать столбцы, можно присвоить свойству `tableMenuButtonVisible` из класса `TableView<S>` значение `true` (по умолчанию свойство имеет значение `false`). В этом случае в заголовке отобразится кнопка, при нажатии которой будет выведено меню с названиями столбцов (рис. 14.2). Чтобы скрыть столбец, достаточно в этом меню сбросить флажок напротив его названия, а чтобы снова отобразить — нужно установить флажок.

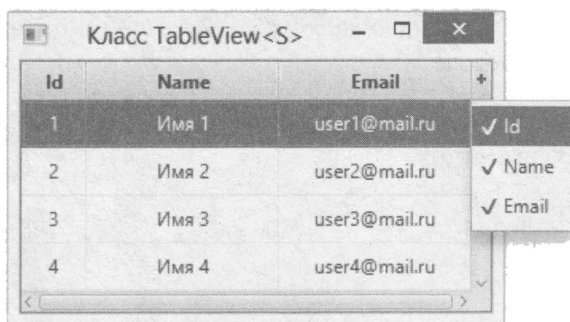


Рис. 14.2. Меню, позволяющее скрыть или отобразить столбцы таблицы

### Методы доступа:

```
public final void setTableMenuButtonVisible(boolean value)
public final boolean isTableMenuButtonVisible()
public final BooleanProperty tableMenuButtonVisibleProperty()
```

### Пример:

```
tableView.setTableMenuButtonVisible(true);
```

Получить информацию о видимых столбцах таблицы позволяют следующие методы из класса `TableView<S>`:

- ❑ `getVisibleLeafColumns()` — возвращает неизменяемый список, содержащий ссылки на отображаемые столбцы (получить список всех столбцов позволяет метод `getColumns()`). Формат метода:

```
public ObservableList<TableColumn<S, ?>> getVisibleLeafColumns()
```

- ❑ `getVisibleLeafColumn()` — возвращает ссылку на столбец, отображаемый в позиции с индексом `column`, или значение `null`. Формат метода:

```
public TableColumn<S, ?> getVisibleLeafColumn(int column)
```

- ❑ `getVisibleLeafIndex()` — возвращает индекс указанного столбца или значение `-1`, если столбец не отображается. Формат метода:

```
public int getVisibleLeafIndex(TableColumn<S, ?> column)
```

## 14.1.6. Управление прокруткой

Управлять прокруткой позволяют следующие свойства из класса `TableView<S>`:

- ❑ `onScrollTo` — задает обработчик события прокрутки к строке таблицы в результате выполнения метода `scrollTo()`. Методы доступа:

```
public void setOnScrollTo(
 EventHandler< ScrollToEvent<Integer> > value)
public EventHandler<ScrollToEvent<Integer>> getOnScrollTo()
public ObjectProperty<EventHandler<ScrollToEvent<Integer>>>
 onScrollToProperty()
```

Внутри обработчика с помощью метода `getScrollTarget()` можно получить индекс строки. Формат метода:

```
public T getScrollTarget()
```

Пример создания обработчика:

```
tableView.setOnScrollTo(event -> {
 System.out.println("OnScrollTo " + event.getScrollTarget());
});
```

Форматы метода `scrollTo()`:

```
public void scrollTo(int index)
public void scrollTo(S object)
```

Первый формат метода принимает индекс строки, к которой нужно выполнить прокрутку, а второй формат вначале выполняет поиск индекса строки по указанному объекту, а затем выполняет прокрутку. Пример:

```
tableView.scrollTo(0);
```

- `onScrollToColumn` — задает обработчик события прокрутки к столбцу таблицы, в результате выполнения методов `scrollToColumn()` и `scrollToColumnIndex()`. Методы доступа:

```
public void setOnScrollToColumn(
 EventHandler<ScrollToEvent<TableColumn<S, ?>>> value)
public EventHandler<ScrollToEvent<TableColumn<S, ?>>>
 getOnScrollToColumn()
public ObjectProperty<EventHandler<ScrollToEvent<
 TableColumn<S, ?>>>> onScrollToColumnProperty()
```

Внутри обработчика с помощью метода `getScrollTarget()` можно получить ссылку на столбец. Пример назначения обработчика:

```
tableView.setOnScrollToColumn(event -> {
 System.out.println("OnScrollToColumn " +
 event.getScrollTarget());
});
```

Форматы методов `scrollToColumn()` и `scrollToColumnIndex()`:

```
public void scrollToColumn(TableColumn<S, ?> column)
public void scrollToColumnIndex(int columnIndex)
```

Методы выполняют прокрутку таким образом, чтобы указанный столбец оказался в зоне видимости. Пример:

```
tableView.scrollToColumn(columnEmail);
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH14_01_TableView`, приведенном в каталоге `CH14_tables` сопровождающего книгу электронного архива (см. приложение).

### 14.1.7. Класс *TableViewSelectionMode<S>*: управление выбором строк и отдельных ячеек

Таблицы поддерживают выбор как одной строки (режим `SelectionMode.SINGLE`), так и нескольких строк (режим `SelectionMode.MULTIPLE`). Существует также режим выбора отдельных ячеек. Для выбора нескольких строк (или ячеек) нужно дополнительно удерживать нажатой клавишу `<Ctrl>` или `<Shift>`. Управлять моделью выбора позволяет свойство `selectionModel` из класса `TableView<S>`. Методы доступа:

```
public final void setSelectionMode(
 TableView.TableViewSelectionMode<S> value)
public final TableView.TableViewSelectionMode<S> getSelectionMode()
public final ObjectProperty<TableView.TableViewSelectionMode<S>>
 selectionModelProperty()
```

Установим для таблицы режим выбора нескольких строк:

```
tableView.getSelectionModel().setSelectionMode(SelectionMode.MULTIPLE);
```

Модель выбора описывается с помощью абстрактного статического класса `TableViewSelectionMode<S>`. Иерархия наследования:

```
Object — SelectionModel<T> — MultipleSelectionModel<T> —
 TableSelectionMode<S> — TableView.TableViewSelectionMode<S>
```

Класс `TableViewSelectionMode<S>` наследует и реализует все методы из классов: `SelectionModel<T>` (см. *разд. 13.3*) и `MultipleSelectionModel<T>` (см. *разд. 13.9*). Все эти методы мы уже рассматривали, поэтому не будем повторяться. Получим индексы и значения выбранных строк после нажатия кнопки:

```
button.setOnAction(event -> {
 ObservableList<Integer> arr =
 tableView.getSelectionModel().getSelectedIndices();
 System.out.println("Индексы выбранных строк:");
 for (Integer index : arr) {
 System.out.println(index);
 }
 ObservableList<User> arr2 =
 tableView.getSelectionModel().getSelectedItems();
 System.out.println("Значения выбранных строк:");
 for (User value : arr2) {
 System.out.println(value);
 }
});
```

Помимо выбора всех ячеек строки, таблицы поддерживают режим выбора отдельных ячеек. Управлять этим режимом позволяет свойство `cellSelectionEnabled` из класса `TableSelectionMode<S>`. Если свойство имеет значение `true`, то будет включен режим выбора отдельных ячеек таблицы. Значение по умолчанию: `false`. Методы доступа:



```
public final void setCellSelectionEnabled(boolean value)
public final boolean isCellSelectionEnabled()
public final BooleanProperty cellSelectionEnabledProperty()
```

**Пример включения режима выбора отдельных ячеек:**

```
tableView.getSelectionModel().setCellSelectionEnabled(true);
```

**Управлять выбором отдельных ячеек позволяют следующие методы:**

- ❑ **select()** — выбирает ячейку в строке с индексом `row` из столбца `column`. Предыдущий выбор не сбрасывается. Форматы метода:

```
public void select(int row, TableColumnBase<S, ?> column)
public abstract void select(int row, TableColumn<S, ?> column)
```

**Пример:**

```
tableView.getSelectionModel().select(0, columnEmail);
```

- ❑ **selectAboveCell()** — выбирает ячейку выше. Предыдущий выбор не сбрасывается. Формат метода:

```
public abstract void selectAboveCell()
```

- ❑ **selectBelowCell()** — выбирает ячейку ниже. Предыдущий выбор не сбрасывается. Формат метода:

```
public abstract void selectBelowCell()
```

- ❑ **selectLeftCell()** — выбирает ячейку слева. Предыдущий выбор не сбрасывается. Формат метода:

```
public abstract void selectLeftCell()
```

- ❑ **selectRightCell()** — выбирает ячейку справа. Предыдущий выбор не сбрасывается. Формат метода:

```
public abstract void selectRightCell()
```

- ❑ **selectRange()** — выбирает диапазон ячеек. Предыдущий выбор не сбрасывается. Формат метода:

```
public void selectRange(
 int minRow, TableColumnBase<S, ?> minColumn,
 int maxRow, TableColumnBase<S, ?> maxColumn)
```

**Пример:**

```
tableView.getSelectionModel().selectRange(
 0, columnId, 5, columnName);
```

- ❑ **clearAndSelect()** — выбирает ячейку в строке с индексом `row` из столбца `column`, при этом сбрасывая предыдущий выбор. Форматы метода:

```
public void clearAndSelect(
 int row, TableColumnBase<S, ?> column)
public abstract void clearAndSelect(
 int row, TableColumn<S, ?> column)
```

❑ `clearSelection()` — сбрасывает выбор указанной ячейки. Форматы метода:

```
public void clearSelection(
 int row, TableColumnBase<S, ?> column)
public abstract void clearSelection(
 int row, TableColumn<S, ?> column)
```

❑ `isSelected()` — возвращает значение `true`, если указанная ячейка выбрана, и `false` — в противном случае. Форматы метода:

```
public boolean isSelected(
 int row, TableColumnBase<S, ?> column)
public abstract boolean isSelected(
 int row, TableColumn<S, ?> column)
```

❑ `getSelectedCells()` — возвращает список всех выбранных ячеек. Формат метода:

```
import javafx.scene.control.TablePosition;
public abstract ObservableList<TablePosition> getSelectedCells()
```

Класс `TablePosition<S, T>` содержит следующие методы:

❑ `getRow()` — возвращает индекс строки. Формат метода:

```
public int getRow()
```

❑ `getColumn()` — возвращает индекс столбца. Формат метода:

```
public int getColumn()
```

❑ `getTableColumn()` — возвращает ссылку на объект столбца или значение `null`. Формат метода:

```
public final TableColumn<S, T> getTableColumn()
```

❑ `getTableView()` — возвращает ссылку на объект таблицы. Формат метода:

```
public final TableView<S> getTableView()
```

Выведем информацию обо всех выбранных ячейках:

```
@SuppressWarnings("rawtypes")
ObservableList<TablePosition> arr =
 tableView.getSelectionModel().getSelectedCells();
for (@SuppressWarnings("rawtypes") TablePosition pos : arr) {
 System.out.print(pos.getRow() + " ");
 System.out.print(pos.getColumn() + " ");
 System.out.print(pos.getTableColumn().getText() + " ");
 System.out.print(
 pos.getTableColumn().getCellObservableValue(pos.getRow()).getValue());
 System.out.println(" " +
 pos.getTableColumn().getCellData(pos.getRow()));
}
```

В этом примере для получения значения выбранной ячейки мы воспользовались следующими методами из класса `TableColumnBase<S, T>`:

- `getCellData()` — возвращает значение по индексу строки или указанному объекту. Форматы метода:

```
public final T getCellData(int index)
public final T getCellData(S item)
```

- `getCellObservableValue()` — возвращает объект `ObservableValue<T>` по индексу строки или указанному объекту. Форматы метода:

```
public final ObservableValue<T>
 getCellObservableValue(int index)
public final ObservableValue<T> getCellObservableValue(S item)
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH14_01_TableView`, приведенном в каталоге `CH14_tables` сопровождающего книгу электронного архива (см. приложение).

## 14.1.8. Класс `TableViewFocusModel<S>`: управление фокусом ввода

Свойство `focusModel` из класса `TableView<S>` содержит ссылку на объект, с помощью которого можно управлять фокусом ввода. Методы доступа:

```
public final void setFocusModel(
 TableView.TableViewFocusModel<S> value)
public final TableView.TableViewFocusModel<S> getFocusModel()
public final ObjectProperty< TableView.TableViewFocusModel<S> >
 focusModelProperty()
```

Иерархия наследования класса `TableViewFocusModel<S>`:

```
Object — FocusModel<T> — TableFocusModel<S, TableColumn<S, ?>> —
 TableView.TableViewFocusModel<S>
```

Класс `TableViewFocusModel<S>` наследует все методы из класса `FocusModel<T>` (см. разд. 13.8.3) и добавляет следующие методы:

- `focus()` — устанавливает фокус ввода на указанную ячейку. Форматы метода:

```
public void focus(int row, TableColumn<S, ?> column)
public void focus(TablePosition pos)
```

Пример:

```
focusModel.focus(0, columnEmail);
```

- `focusAboveCell()` — устанавливает фокус на ячейку выше. Формат метода:

```
public void focusAboveCell()
```

- `focusBelowCell()` — устанавливает фокус на ячейку ниже. Формат метода:

```
public void focusBelowCell()
```

- `focusLeftCell()` — устанавливает фокус на ячейку слева. Формат метода:

```
public void focusLeftCell()
```

❑ `focusRightCell()` — устанавливает фокус на ячейку справа. Формат метода:

```
public void focusRightCell()
```

❑ `isFocused()` — возвращает значение `true`, если указанная ячейка находится в фокусе ввода, и `false` — в противном случае. Формат метода:

```
public boolean isFocused(int row, TableColumn<S, ?> column)
```

### ПРИМЕЧАНИЕ

Чтобы иметь возможность устанавливать фокус ввода на отдельную ячейку, нужно чтобы свойство `cellSelectionEnabled` из класса `TableSelectionModel<S>` имело значение `true`. В противном случае фокус ввода будет иметь вся строка таблицы, а не отдельная ячейка строки.

Отслеживать изменение фокуса ввода отдельной ячейки позволяет свойство `focusedCell`. Методы доступа:

```
import javafx.scene.control.TablePosition;
public final TablePosition getFocusedCell()
public final ReadOnlyObjectProperty<TablePosition> focusedCellProperty()
```

Пример назначения обработчика:

```
tableView.getFocusModel().focusedCellProperty().addListener(
 (obj, oldValue, newValue) -> {
 System.out.println("Cell " + newValue);
 });
```

Если свойство `cellSelectionEnabled` из класса `TableSelectionModel<S>` имеет значение `false`, то можно, например, отслеживать изменение индекса строки:

```
tableView.getFocusModel().focusedIndexProperty().addListener(
 (obj, oldValue, newValue) -> {
 System.out.println("Index " + newValue);
 });
```

После нажатия кнопки выведем информацию о ячейках или строках, находящихся в фокусе ввода:

```
button.setOnAction(event -> {
 tableView.requestFocus();
 TableView<User> focusModel =
 tableView.getFocusModel();
 if (tableView.getSelectionModel().isCellSelectionEnabled()) {
 System.out.println("Cell " + focusModel.getFocusedCell());
 }
 else {
 System.out.println("Index " + focusModel.getFocusedIndex());
 System.out.println("Item " + focusModel.getFocusedItem());
 }
});
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH14_01_TableView`, приведенном в каталоге `CH14_tables` сопровождающего книгу электронного архива (см. приложение).

## 14.1.9. Изменение свойств ячеек столбца

Изменить характеристики ячеек столбца позволяет свойство `cellFactory` из класса `TableColumn<S, T>`. Методы доступа:

```
import javafx.util.Callback;
import javafx.scene.control.TableCell;
public final void setCellFactory(
 Callback<TableColumn<S, T>, TableCell<S, T>> value)
public final Callback<TableColumn<S, T>, TableCell<S, T>>
 getCellFactory()
public final ObjectProperty<Callback<TableColumn<S, T>,
 TableCell<S, T>>> cellFactoryProperty()
```

Иерархия наследования для класса `TableCell<S, T>`:

```
Object – Node – Parent – Region – Control – Labeled – Cell<T> –
 IndexedCell<T> – TableCell<S, T>
```

Давайте выполним выравнивание значений по центру в столбце с идентификатором:

```
Callback< TableColumn<User, Number>,
 TableCell<User, Number> > cellFactoryId =
 new Callback<TableColumn<User, Number>, TableCell<User, Number>>() {

 @Override
 public TableCell<User, Number> call(TableColumn<User, Number> p) {
 return new TableCell<User, Number>() {
 {
 setAlignment(Pos.TOP_CENTER);
 }
 @Override
 protected void updateItem(Number value, boolean empty) {
 super.updateItem(value, empty);
 if (value == null || empty) {
 setText("");
 }
 else {
 setText(value.toString());
 }
 }
 };
 }
};

columnId.setCellFactory(cellFactoryId);
```

Точно таким же образом мы можем изменить цвет фона ячеек, задать характеристики шрифта, отобразить какой-либо узел рядом с текстом (или вместо текста), разрешить перенос текста на новую строку (`setWrapText(true)`) и др. Класс `TableCell<S, T>` наследует все методы из класса `Labeled` (см. *разд. 11.3*), поэтому мы можем работать с ячейкой таблицы как с текстовой надписью.

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH14_01_TableView`, приведенном в каталоге `CH14_tables` сопровождающего книгу электронного архива (см. *приложение*).

## 14.1.10. Изменение свойств ячеек строки

Изменить характеристики сразу всех ячеек строки позволяет свойство `rowFactory` из класса `TableView<S>`. Методы доступа:

```
import javafx.util.Callback;
import javafx.scene.control.TableRow;

public final void setRowFactory(
 Callback<TableView<S>, TableRow<S>> value)
public final Callback<TableView<S>, TableRow<S>> getRowFactory()
public final ObjectProperty<Callback<TableView<S>, TableRow<S>>>
 rowFactoryProperty()
```

Иерархия наследования для класса `TableRow<T>`:

```
Object — Node — Parent — Region — Control — Labeled — Cell<T> —
 IndexedCell<T> — TableRow<T>
```

Давайте выделим строку, содержащую информацию о пользователе с идентификатором 8, зеленым цветом:

```
Callback<TableView<User>, TableRow<User>> rowFactory =
 new Callback<TableView<User>, TableRow<User>>() {
 @Override
 public TableRow<User> call(TableView<User> p) {
 return new TableRow<User>() {
 @Override
 protected void updateItem(User user, boolean empty) {
 super.updateItem(user, empty);
 if (user == null || empty) {
 setStyle("");
 } else {
 if (user.getId() == 8) {
 setStyle("-fx-background-color: green;");
 }
 else {
 setStyle("");
 }
 }
 }
 }
 }
 }
```

```

 }
 };
}
};
tableView.setRowFactory(rowFactory);

```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH14_01_TableView`, приведенном в каталоге `CH14_tables` сопровождающего книгу электронного архива (см. *приложение*).

## 14.1.11. Сортировка элементов

Таблицы по умолчанию поддерживают сортировку элементов внутри столбца. Для этого достаточно щелкнуть левой кнопкой мыши на заголовке столбца. В результате в заголовке отобразится черный треугольник, показывающий направление сортировки. Чтобы изменить направление сортировки, нужно щелкнуть левой кнопкой мыши на заголовке еще раз. Можно выполнять сортировку сразу по нескольким столбцам, удерживая нажатой клавишу `<Shift>` при щелчке левой кнопкой мыши.

Назначить обработчик события сортировки позволяет свойство `onSort` из класса `TableView<S>`. Методы доступа:

```

import javafx.scene.control.SortEvent;

public void setOnSort(
 EventHandler< SortEvent< TableView<S> > > value)
public EventHandler< SortEvent< TableView<S> > > getOnSort()
public ObjectProperty< EventHandler<SortEvent<TableView<S>>> >
 onSortProperty()

```

Пример назначения обработчика:

```

tableView.setOnSort(event -> {
 System.out.println("Сортировка");
});

```

Можно также воспользоваться следующим кодом:

```

tableView.addEventHandler(SortEvent.ANY, event -> {
 System.out.println("SortEvent.ANY");
});

```

Управлять политикой сортировки позволяет свойство `sortPolicy` из класса `TableView<S>`. Методы доступа:

```

public final void setSortPolicy(
 Callback<TableView<S>, Boolean> callback)
public final Callback<TableView<S>, Boolean> getSortPolicy()
public final ObjectProperty<Callback<TableView<S>, Boolean>>
 sortPolicyProperty()

```

Запретим сортировку:

```

tableView.setSortPolicy(table -> false);

```

**Пример реализации сортировки:**

```
tableView.setSortPolicy(new Callback<TableView<User>, Boolean>() {
 @Override
 public Boolean call(TableView<User> table) {
 if (table.getComparator() != null)
 FXCollections.sort(table.getItems(),
 table.getComparator());
 return true;
 }
});
```

В этом примере мы воспользовались свойством `comparator` из класса `TableView<S>`, которое возвращает способ сортировки или значение `null`. Методы доступа:

```
public final Comparator<S> getComparator()
public final ReadOnlyObjectProperty<Comparator<S>> comparatorProperty()
```

Получить ссылку на список со столбцами, требующими сортировки, позволяет метод `getSortOrder()` из класса `TableView<S>`. Формат метода:

```
public final ObservableList<TableColumn<S, ?>> getSortOrder()
```

В этот список мы можем добавить несколько столбцов, что приведет к сортировке сразу по двум и более столбцам. Вначале сортировка производится по первому столбцу в списке, а в случае равенства значений производится сортировка по следующему столбцу. Вызвать событие сортировки позволяет метод `sort()` из класса `TableView<S>`. Формат метода:

```
public void sort()
```

**Пример запуска сортировки по возрастанию по столбцу `Email` после нажатия кнопки:**

```
button.setOnAction(event -> {
 columnEmail.setSortType(SortType.ASCENDING);
 tableView.getSortOrder().clear();
 tableView.getSortOrder().add(columnEmail);
 tableView.sort();
});
```

В этом примере с помощью свойства `sortType` из класса `TableColumn<S, T>` мы указали направление сортировки. Методы доступа:

```
public final void setSortType(TableColumn.SortType value)
public final TableColumn.SortType getSortType()
public final ObjectProperty<TableColumn.SortType> sortTypeProperty()
```

Перечисление `TableColumn.SortType` содержит константы: `ASCENDING` (сортировка по возрастанию) и `DESCENDING` (сортировка по убыванию).

Класс `TableColumn<S, T>` наследует следующие свойства из класса `TableColumnBase<S, T>`:

☐ `sortable` — разрешает (значение `true`) или запрещает (значение `false`) сортировку для столбца. Методы доступа:



```
public final void setSortable(boolean value)
public final boolean isSortable()
public final BooleanProperty sortableProperty()
```

- ❑ **comparator** — задает способ сравнения значений столбца при сортировке. Методы доступа:

```
public final void setComparator(Comparator<T> value)
public final Comparator<T> getComparator()
public final ObjectProperty<Comparator<T>> comparatorProperty()
```

- ❑ **sortNode** — позволяет указать узел, который будет отображаться в заголовке столбца вместо значка в виде треугольника, показывающего направление сортировки. Методы доступа:

```
public final void setSortNode(Node value)
public final Node getSortNode()
public final ObjectProperty<Node> sortNodeProperty()
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH14_01_TableView`, приведенном в каталоге `CH14_tables` сопровождающего книгу электронного архива (см. *приложение*).

## 14.1.12. Редактирование элементов

Чтобы пользователь мог редактировать значения элементов таблицы, необходимо присвоить свойству `editable` из класса `TableView<S>` значение `true`. По умолчанию свойство имеет значение `false`. Методы доступа:

```
public final void setEditable(boolean value)
public final boolean isEditable()
public final BooleanProperty editableProperty()
```

Давайте создадим таблицу и разрешим редактирование:

```
TableView<User> tableView = new TableView<User>();
tableView.setItems(list);
tableView.setEditable(true);
tableView.getSelectionModel().selectFirst();
tableView.setColumnResizePolicy(TableView.CONSTRAINED_RESIZE_POLICY);
tableView.setPrefWidth(300.0);
```

Свойство `editable` есть также в классе `TableColumn<S, T>` (наследуется из класса `TableColumnBase<S, T>`). По умолчанию свойство имеет значение `true`. Чтобы запретить редактирование ячеек столбца, нужно присвоить свойству значение `false`. Методы доступа:

```
public final void setEditable(boolean value)
public final boolean isEditable()
public final BooleanProperty editableProperty()
```

Кроме изменения значения свойства `editable`, нужно задать значение свойству `cellFactory`. Можно реализовать свой собственный способ редактирования или воспользоваться готовыми классами из пакета `javafx.scene.control.cell`. Например, классом `TextFieldTableCell<S, T>`, который в режиме редактирования отображает внутри ячейки текстовое поле. Из этого класса нам понадобится статический метод `forTableColumn()`. Форматы метода:

```
import javafx.scene.control.cell.TextFieldTableCell;
public static <S> Callback<TableColumn<S, String>,
 TableCell<S, String>> forTableColumn()
public static <S, T> Callback<TableColumn<S, T>,
 TableCell<S, T>> forTableColumn(StringConverter<T> converter)
```

Первый формат метода работает только с типом `String`. Создадим столбцы `Name` и `Email`:

```
TableColumn<User, String> columnName =
 new TableColumn<User, String>("Name");
columnName.setCellValueFactory(
 cellData -> cellData.getValue().nameProperty());
columnName.setCellFactory(TextFieldTableCell.<User>forTableColumn());

TableColumn<User, String> columnEmail =
 new TableColumn<User, String>("Email");
columnEmail.setCellValueFactory(
 cellData -> cellData.getValue().emailProperty());
columnEmail.setCellFactory(TextFieldTableCell.<User>forTableColumn());
```

Второй формат метода позволяет указать способ преобразования значения в строку и строки в значение. Создадим столбец `Id`:

```
TableColumn<User, Number> columnId =
 new TableColumn<User, Number>("Id");
columnId.setCellValueFactory(
 cellData -> cellData.getValue().idProperty());
columnId.setCellFactory(TextFieldTableCell.<User, Number>forTableColumn(
 new StringConverter<Number>() {
 @Override
 public Number fromString(String string) {
 try {
 return Integer.valueOf(string);
 } catch (Exception e) {
 return null;
 }
 }
 @Override
 public String toString(Number object) {
 return object == null ? "" : object.toString();
 }
 }
));
```

Класс `TableColumn<S, T>` содержит следующие статические методы, возвращающие типы событий редактирования:

□ `editAnyEvent()` — любое событие редактирования. Формат метода:

```
public static <S, T> EventType<TableColumn.CellEditEvent<S, T>>
 editAnyEvent()
```

□ `editStartEvent()` — событие начала редактирования (пользователь выполнил двойной щелчок левой кнопкой мыши на ячейке или выделил ячейку и нажал клавишу `<Enter>`). Формат метода:

```
public static <S, T> EventType<TableColumn.CellEditEvent<S, T>>
 editStartEvent()
```

□ `editCommitEvent()` — событие успешного окончания редактирования (например, пользователь ввел значение в текстовое поле и нажал клавишу `<Enter>`). Формат метода:

```
public static <S, T> EventType<TableColumn.CellEditEvent<S, T>>
 editCommitEvent()
```

□ `editCancelEvent()` — событие отказа от редактирования (например, пользователь внутри текстового поля нажал клавишу `<Esc>` или выделил другую ячейку в режиме редактирования). Формат метода:

```
public static <S, T> EventType<TableColumn.CellEditEvent<S, T>>
 editCancelEvent()
```

Назначить обработчик для событий редактирования позволяет метод `addEventHandler()` из класса `TableColumnBase<S, T>`. Формат метода:

```
public <E extends Event> void addEventHandler(
 EventType<E> eventType, EventHandler<E> eventHandler)
```

Для удаления обработчика предназначен метод `removeEventHandler()` из класса `TableColumnBase<S, T>`. Формат метода:

```
public <E extends Event> void removeEventHandler(
 EventType<E> eventType, EventHandler<E> eventHandler)
```

**Пример назначения обработчика и получения значений:**

```
columnId.addEventHandler(
 TableColumn.<User, Number>editCommitEvent(), event -> {
 System.out.println(event.getNewValue());
 System.out.println(event.getOldValue());
 System.out.println(event.getRowValue());
 System.out.println(event.getTableView());
 System.out.println(event.getTableColumn());
 System.out.println(event.getTablePosition());
 System.out.println(event.getTablePosition().getRow());
 System.out.println(event.getTablePosition().getColumn());
 System.out.println(event.getTableView().getEditingCell());
 });
```

Внутри обработчика события через параметр `event` доступен экземпляр класса `TableColumn.CellEditEvent<S, T>`, который имеет следующие методы:

- ❑ `getNewValue()` — возвращает новое значение при успешном окончании редактирования. Внутри обработчиков других событий метод возвращает значение `null`. Формат метода:

```
public T getNewValue()
```

- ❑ `getOldValue()` — возвращает старое значение или значение `null`. Формат метода:

```
public T getOldValue()
```

- ❑ `getRowValue()` — возвращает ссылку на объект, значения которого отображаются в строке с редактируемой ячейкой (в нашем примере экземпляр класса `User`). Формат метода:

```
public S getRowValue()
```

- ❑ `getTableView()` — возвращает ссылку на объект таблицы. Формат метода:

```
public TableView<S> getTableView()
```

- ❑ `getTableColumn()` — возвращает ссылку на объект столбца. Формат метода:

```
public TableColumn<S, T> getTableColumn()
```

- ❑ `getTablePosition()` — возвращает ссылку на объект, с помощью которого можно получить информацию о положении редактируемой ячейки, — например, индексы строки и столбца. Формат метода:

```
public TablePosition<S, T> getTablePosition()
```

Получить информацию о положении редактируемой ячейки можно также с помощью свойства `editingCell` из класса `TableView<S>`. Если нет редактируемой ячейки, то свойство будет иметь значение `null`. Методы доступа:

```
public final TablePosition<S, ?> getEditingCell()
public final ReadOnlyObjectProperty<TablePosition<S, ?>>
 editingCellProperty()
```

Управлять обработчиками событий редактирования позволяют также следующие свойства из класса `TableColumn<S, T>`:

- ❑ `onEditStart` — назначает обработчик события начала редактирования. Методы доступа:

```
public final void setOnEditStart(
 EventHandler<TableColumn.CellEditEvent<S, T>> value)
public final EventHandler<TableColumn.CellEditEvent<S, T>>
 getOnEditStart()
public final ObjectProperty<EventHandler<
 TableColumn.CellEditEvent<S, T>>> onEditStartProperty()
```

- ❑ `onEditCommit` — назначает обработчик события успешного окончания редактирования. Методы доступа:

```

public final void setOnEditCommit(
 EventHandler<TableColumn.CellEditEvent<S, T>> value)
public final EventHandler<TableColumn.CellEditEvent<S, T>>
 getOnEditCommit()
public final ObjectProperty<EventHandler<
 TableColumn.CellEditEvent<S, T>>> onEditCommitProperty()

```

**Обратите внимание:** если вы назначаете обработчик через свойство `onEditCommit`, то переопределяете обработчик по умолчанию, который изменяет значение элемента. Следовательно, нужно будет самостоятельно изменить значение, например, после проверки условия. Если надо просто отследить событие, а не переопределять обработчик по умолчанию, то можно для назначения обработчика воспользоваться методом `addEventHandler()` из класса `TableColumnBase<S, T>`. Давайте назначим обработчик для столбца `Id` и внутри него проверим допустимость нового значения и либо откажемся от изменения значения, либо изменим значение в списке:

```

columnId.setOnEditCommit(event -> {
 if (event.getNewValue() == null) {
 tableView.refresh();
 }
 else {
 event.getRowValue().setId((Integer)event.getNewValue());
 }
});

```

В этом примере мы воспользовались методом `refresh()` из класса `TableView<S>`. Этот метод вызывает обновление видимой части таблицы. Если пользователь вводит в ячейку значение, которое невозможно преобразовать в целое число, то метод `getNewValue()` вернет значение `null`, что нас не устраивает в качестве нового значения. Поэтому мы вызываем обновление и возвращаем ячейке значение, которое она имела до редактирования. Метод доступен, начиная с версии 8.60. Формат метода:

```
public void refresh()
```

- ❑ `onEditCancel` — назначает обработчик события отказа от редактирования. Методы доступа:

```

public final void setOnEditCancel(
 EventHandler<TableColumn.CellEditEvent<S, T>> value)
public final EventHandler<TableColumn.CellEditEvent<S, T>>
 getOnEditCancel()
public final ObjectProperty<EventHandler<
 TableColumn.CellEditEvent<S, T>>> onEditCancelProperty()

```

**Перейти в режим редактирования ячейки** из программы позволяет метод `edit()` из класса `TableView<S>`. Обратите внимание: ячейка таблицы должна быть в области видимости, иначе эффекта не будет. Формат метода:

```
public void edit(int row, TableColumn<S, ?> column)
```

Пример:

```
tableView.edit(0, columnEmail);
```

Если нужно выйти из режима редактирования, то в первом параметре указывается отрицательное значение, а во втором параметре — значение `null`.

### **ЭЛЕКТРОННЫЙ АРХИВ**

Полный текст программы содержится в проекте `FX_CH14_01_TableView`, приведенном в каталоге `CH14_tables` сопровождающего книгу электронного архива (см. приложение).

## **14.1.13. Изменение порядка следования столбцов**

По умолчанию пользователь может изменить положение столбца, взявшись мышью за заголовок столбца и перетащив его в другое место. Чтобы запретить перетаскивание отдельного столбца, нужно присвоить значение `false` свойству `reorderable` из класса `TableColumnBase<S, T>`. Свойство доступно, начиная с JavaFX 9. Методы доступа:

```
public final void setReorderable(boolean value)
public final boolean isReorderable()
public final BooleanProperty reorderableProperty()
```

Запретим перетаскивание столбца с идентификатором:

```
columnId.setReorderable(false);
```

Чтобы изменить порядок следования столбцов из программы, достаточно изменить порядок следования элементов внутри списка, возвращаемого методом `getColumns()` из класса `TableView<S>`.

### **ЭЛЕКТРОННЫЙ АРХИВ**

Полный текст программы содержится в проекте `FX_CH14_01_TableView`, приведенном в каталоге `CH14_tables` сопровождающего книгу электронного архива (см. приложение).

## **14.2. Класс *TreeView<T>*: иерархический список**

Класс `TreeView<T>` реализует иерархический список (рис. 14.3). Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.TreeView;
```

Иерархия наследования:

```
Object — Node — Parent — Region — Control — TreeView<T>
```

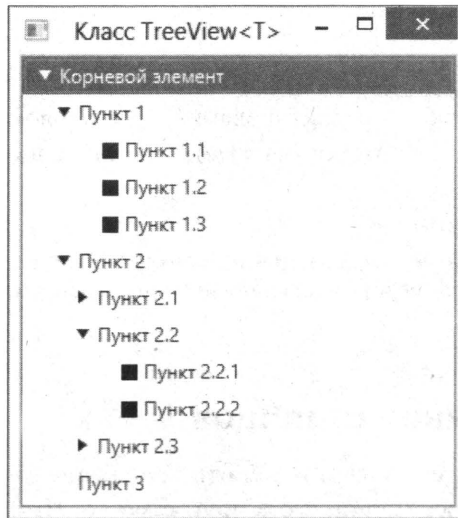


Рис. 14.3. Класс `TreeView<T>`: иерархический список

## 14.2.1. Создание иерархического списка

Конструкторы класса `TreeView<T>`:

```
TreeView()
TreeView(TreeItem<T> root)
```

Первый конструктор создает пустой список, а второй конструктор позволяет указать корневой элемент. Пример:

```
TreeView<String> treeView = new TreeView<String>();
TreeItem<String> root = new TreeItem<String>("Корневой элемент");
root.setExpanded(true);
for (int i = 1; i < 6; i++) {
 root.getChildren().add(new TreeItem<String>("Пункт " + i));
}
TreeItem<String> item = new TreeItem<String>("Пункт 6");
item.setGraphic(new Rectangle(10.0, 10.0, Color.RED));
item.getChildren().add(new TreeItem<String>("Пункт 6.1"));
root.getChildren().add(item);
treeView.setRoot(root);
```

Класс `TreeView<T>` содержит следующие основные свойства:

- ❑ `root` — содержит ссылку на корневой элемент или значение `null`. Методы доступа:

```
public final void setRoot(TreeItem<T> value)
public final TreeItem<T> getRoot()
public final ObjectProperty<TreeItem<T>> rootProperty()
```

- ❑ `showRoot` — если свойство имеет значение `true` (значение по умолчанию), то корневой элемент будет отображаться, а если `false` — то он будет скрыт. Методы доступа:

```
public final void setShowRoot(boolean value)
public final boolean isShowRoot()
public final BooleanProperty showRootProperty()
```

- ❑ `expandedItemCount` — содержит количество раскрытых элементов. Методы доступа:

```
public final int getExpandedItemCount()
public final ReadOnlyIntegerProperty expandedItemCountProperty()
```

- ❑ `cellFactory` — задает характеристики ячеек списка. Методы доступа:

```
import javafx.util.Callback;
import javafx.scene.control.TreeCell;
public final void setCellFactory(
 Callback<TreeView<T>, TreeCell<T>> value)
public final Callback<TreeView<T>, TreeCell<T>> getCellFactory()
public final ObjectProperty<Callback<TreeView<T>,
 TreeCell<T>>> cellFactoryProperty()
```

- ❑ `fixedCellSize` — позволяет задать фиксированную высоту для всех ячеек списка. Значение по умолчанию: `-1.0`. Методы доступа:

```
public final void setFixedCellSize(double value)
public final double getFixedCellSize()
public final DoubleProperty fixedCellSizeProperty()
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH14_02_TreeView`, приведенном в каталоге `CH14_tables` сопровождающего книгу электронного архива (см. *приложение*).

## 14.2.2. Класс *TreeItem<T>*: элемент иерархического списка

Отдельный элемент иерархического списка реализуется с помощью класса `TreeItem<T>`. Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.TreeItem;
```

Иерархия наследования:

```
Object — TreeItem<T>
```

Конструкторы класса:

```
TreeItem()
TreeItem(T value)
TreeItem(T value, Node graphic)
```



Первый конструктор создает объект с настройками по умолчанию, второй конструктор позволяет задать значение элемента, а третий конструктор — значение элемента и узел, который будет отображаться перед значением (например, изображение). Пример:

```
TreeItem<String> treeItem = new TreeItem<String>("Пункт 1");
TreeItem<String> treeItem2 = new TreeItem<String>("Пункт 2",
 new Rectangle(10.0, 10.0, Color.RED));
```

Для создания элемента иерархического списка и добавления его в список дочерних элементов удобно применить пользовательский метод `addTreeItem()`:

```
public TreeItem<String> addTreeItem(
 TreeItem<String> parent, String text,
 Node graphic, boolean expanded) {
 TreeItem<String> item = new TreeItem<String>(text);
 if (graphic != null) item.setGraphic(graphic);
 item.setExpanded(expanded);
 if (parent != null) parent.getChildren().add(item);
 return item;
}
```

В параметре `parent` метод `addTreeItem()` принимает ссылку на родительский элемент. Если передать значение `null`, то элемент не будет добавляться в список дочерних элементов. Это удобно для создания корневого элемента. Параметр `text` задает текстовую надпись, а параметр `graphic` — узел, который будет отображаться перед надписью. Если параметр `graphic` имеет значение `null`, то узел добавляться не будет. Если в параметре `expanded` указать значение `true`, то список дочерних элементов будет отображаться, а если `false` — то он будет свернут. В качестве значения метод возвращает ссылку на добавленный или просто созданный элемент. Эту ссылку нужно указать при создании дочерних элементов.

Чтобы было еще удобнее, создадим перегруженные версии метода `addTreeItem()`:

```
public TreeItem<String> addTreeItem(String text) {
 return addTreeItem(null, text, null, true);
}
public TreeItem<String> addTreeItem(
 TreeItem<String> parent, String text) {
 return addTreeItem(parent, text, null, false);
}
public TreeItem<String> addTreeItem(
 TreeItem<String> parent, String text, Node graphic) {
 return addTreeItem(parent, text, graphic, false);
}
public TreeItem<String> addTreeItem(
 TreeItem<String> parent, String text, boolean expanded) {
 return addTreeItem(parent, text, null, expanded);
}
```

Если мы еще дополнительно при создании объектов используем инструкцию `var`, то процесс добавления элементов в иерархический список станет удобным и наглядным:

```
var rootItem = addTreeItem("Корневой элемент");

var treeItem = addTreeItem(rootItem, "Пункт 1", true);
addTreeItem(treeItem, "Пункт 1.1");
addTreeItem(treeItem, "Пункт 1.2");

addTreeItem(rootItem, "Пункт 2", new Rectangle(10.0, 10.0, Color.RED));
addTreeItem(rootItem, "Пункт 3", new Rectangle(10.0, 10.0, Color.GREEN));

var tree = new TreeView<String>(rootItem);
```

**Класс `TreeItem<T>` содержит следующие свойства:**

- ❑ **value** — задает значение элемента. Методы доступа:

```
public final void setValue(T value)
public final T getValue()
public final ObjectProperty<T> valueProperty()
```

- ❑ **graphic** — задает узел (например, изображение), который будет отображаться перед значением. Методы доступа:

```
public final void setGraphic(Node value)
public final Node getGraphic()
public final ObjectProperty<Node> graphicProperty()
```

**Пример:**

```
TreeItem<String> treeItem3 = new TreeItem<String>();
treeItem3.setValue("Пункт 3");
treeItem3.setGraphic(new Rectangle(10.0, 10.0, Color.GREEN));
```

- ❑ **expanded** — если свойство имеет значение `true`, то дочерние элементы (при их наличии) будут отображаться, а если `false` (значение по умолчанию) — скрыты. Методы доступа:

```
public final void setExpanded(boolean value)
public final boolean isExpanded()
public final BooleanProperty expandedProperty()
```

- ❑ **leaf** — содержит значение `true`, если элемент не имеет дочерних элементов, и `false` — в противном случае. Методы доступа:

```
public boolean isLeaf()
public final ReadOnlyBooleanProperty leafProperty()
```

- ❑ **parent** — содержит ссылку на родительский элемент или значение `null`, если нет родительского элемента (например, для корневого элемента). Методы доступа:

```
public final TreeItem<T> getParent()
public final ReadOnlyObjectProperty<TreeItem<T>>
 parentProperty()
```

Класс `TreeItem<T>` содержит следующие методы:

- ❑ `getChildren()` — возвращает ссылку на список дочерних элементов. Если элемент содержит дочерние элементы, то слева от значения в ячейке отобразится треугольник, показывающий, свернут список или отображен. Для изменения состояния достаточно щелкнуть левой кнопкой мыши на элементе. Формат метода:

```
public ObservableList<TreeItem<T>> getChildren()
```

Пример добавления элементов в список:

```
TreeItem<String> treeItem = new TreeItem<String>("Пункт 1");
treeItem.getChildren().add(new TreeItem<String>("Пункт 1.1"));
```

- ❑ `nextSibling()` — возвращает ссылку на следующий элемент, расположенный на том же уровне вложенности, или значение `null`, если такого элемента нет. Форматы метода:

```
public TreeItem<T> nextSibling()
public TreeItem<T> nextSibling(TreeItem<T> afterNode)
```

- ❑ `previousSibling()` — возвращает ссылку на предыдущий элемент, расположенный на том же уровне вложенности, или значение `null`, если такого элемента нет. Форматы метода:

```
public TreeItem<T> previousSibling()
public TreeItem<T> previousSibling(TreeItem<T> beforeNode)
```

Получить информацию об элементе внутри иерархического списка позволяют следующие методы из класса `TreeView<T>`:

- ❑ `getRow()` — возвращает индекс позиции указанного элемента или значение `-1`, если элемент не отображается. Формат метода:

```
public int getRow(TreeItem<T> item)
```

- ❑ `getTreeItem()` — возвращает ссылку на элемент, находящийся в списке на указанной позиции, или значение `null`, если элемента нет. Формат метода:

```
public TreeItem<T> getTreeItem(int row)
```

- ❑ `getTreeItemLevel()` — возвращает уровень вложенности для указанного элемента или значение `-1`. Формат метода:

```
public int getTreeItemLevel(TreeItem<?> node)
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH14_02_TreeView`, приведенном каталоге `CH14_tables` сопровождающего книгу электронного архива (см. *приложение*).

## 14.2.3. Обработка событий

Назначить обработчик для событий изменения элемента позволяет метод `addEventHandler()` из класса `TreeItem<T>`. Формат метода:

```
public <E extends Event> void addEventHandler(
 EventType<E> eventType, EventHandler<E> eventHandler)
```

Для удаления обработчика предназначен метод `removeEventHandler()` из класса `TreeItem<T>`. Формат метода:

```
public <E extends Event> void removeEventHandler(
 EventType<E> eventType, EventHandler<E> eventHandler)
```

**Обратите внимание:** класс `TreeItem<T>` не наследует класс `Node`, а следовательно, не является узлом, и может обрабатывать только следующие события:

- ❑ `treeNotificationEvent()` — базовый тип события, позволяющий перехватывать все приведенные далее события. Формат метода:

```
public static <T> EventType<TreeItem.TreeModificationEvent<T>>
 treeNotificationEvent()
```

- ❑ `valueChangeEvent()` — изменение значения элемента. Формат метода:

```
public static <T> EventType<TreeItem.TreeModificationEvent<T>>
 valueChangeEvent()
```

**Пример назначения обработчика события и получения нового значения:**

```
root.addEventHandler(
 TreeItem.<String>valueChangeEvent(), event -> {
 System.out.println(event.getNewValue());
 System.out.println(event.getTreeItem().getValue());
 });
```

События дочерних элементов всплывают к корневому элементу, поэтому нет необходимости назначать обработчики для каждого элемента списка. Достаточно назначить обработчик для корневого элемента, а ссылку на элемент, являющийся источником события, можно получить с помощью метода `getTreeItem()` через объект события;

- ❑ `graphicChangedEvent()` — изменение узла, отображаемого слева от значения. Формат метода:

```
public static <T> EventType<TreeItem.TreeModificationEvent<T>>
 graphicChangedEvent()
```

- ❑ `expandedItemCountChangeEvent()` — базовый тип события, позволяющий перехватывать все приведенные далее события. Формат метода:

```
public static <T> EventType<TreeItem.TreeModificationEvent<T>>
 expandedItemCountChangeEvent()
```

- ❑ `branchExpandedEvent()` — отображение дочерних элементов. Формат метода:

```
public static <T> EventType<TreeItem.TreeModificationEvent<T>>
 branchExpandedEvent()
```

- ❑ `branchCollapsedEvent()` — сокрытие дочерних элементов. Формат метода:

```
public static <T> EventType<TreeItem.TreeModificationEvent<T>>
 branchCollapsedEvent()
```

- ❑ `childrenModificationEvent()` — добавление или удаление дочерних элементов, а также изменение порядка следования дочерних элементов. Формат метода:

```
public static <T> EventType<TreeItem.TreeModificationEvent<T>>
 childrenModificationEvent()
```

### Пример назначения обработчика и получения значений:

```
root.addEventHandler(
 TreeItem.<String>childrenModificationEvent(), event -> {
 if (event.wasAdded()) {
 System.out.println("Добавлены элементы");
 System.out.println(event.getAddedSize());
 System.out.println(event.getAddedChildren());
 }
 if (event.wasRemoved()) {
 System.out.println("Удалены элементы");
 System.out.println(event.getRemovedSize());
 System.out.println(event.getRemovedChildren());
 }
 if (event.wasPermutated()) {
 System.out.println("Изменился порядок следования");
 System.out.println(event.getTreeItem().getChildren());
 }
 });
```

### Иерархия типов событий выглядит так:

```
treeNotificationEvent()
 valueChangeEvent()
 graphicChangeEvent()
 expandedItemCountChangeEvent()
 branchExpandedEvent()
 branchCollapsedEvent()
 childrenModificationEvent()
```

Внутри обработчика через параметр `event` доступен экземпляр статического класса `TreeItem.TreeModificationEvent<T>`. Иерархия наследования:

`Object` — `java.util.EventObject` — `Event` — `TreeModificationEvent<T>`

Класс `TreeItem.TreeModificationEvent<T>` наследует все методы из базовых классов и добавляет следующие методы:

- ❑ `getTreeItem()` — возвращает ссылку на элемент, который является источником события. Формат метода:

```
public TreeItem<T> getTreeItem()
```

- ❑ `getNewValue()` — возвращает новое значение элемента или значение `null`, если изменения не было. Формат метода:

```
public T getNewValue()
```

- ❑ `wasExpanded()` — возвращает значение `true`, если дочерние элементы были отображены, и `false` — в противном случае. Формат метода:  

```
public boolean wasExpanded()
```
- ❑ `wasCollapsed()` — возвращает значение `true`, если дочерние элементы были скрыты, и `false` — в противном случае. Формат метода:  

```
public boolean wasCollapsed()
```
- ❑ `wasAdded()` — возвращает значение `true`, если элементы были добавлены в список, и `false` — в противном случае. Формат метода:  

```
public boolean wasAdded()
```
- ❑ `getAddedSize()` — возвращает количество добавленных элементов или значение 0, если элементы не добавлялись. Формат метода:  

```
public int getAddedSize()
```
- ❑ `getAddedChildren()` — возвращает список с добавленными элементами или пустой список, если элементы не добавлялись. Формат метода:  

```
public List<? extends TreeItem<T>> getAddedChildren()
```
- ❑ `wasRemoved()` — возвращает значение `true`, если дочерние элементы были удалены, и `false` — в противном случае. Формат метода:  

```
public boolean wasRemoved()
```
- ❑ `getRemovedSize()` — возвращает количество удаленных элементов или значение 0, если элементы не удалялись. Формат метода:  

```
public int getRemovedSize()
```
- ❑ `getRemovedChildren()` — возвращает список с удаленными элементами или пустой список, если элементы не удалялись. Формат метода:  

```
public List<? extends TreeItem<T>> getRemovedChildren()
```
- ❑ `wasPermutated()` — возвращает значение `true`, если изменился порядок следования дочерних элементов внутри списка, но при этом не было добавления новых элементов или удаления существующих элементов, и `false` — в противном случае. Формат метода:  

```
public boolean wasPermutated()
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH14_02_TreeView`, приведенном в каталоге `CH14_tables` сопровождающего книгу электронного архива (см. *приложение*).

## 14.2.4. Управление прокруткой

Прокрутить список так, чтобы элемент с указанным индексом оказался в зоне видимости, позволяет метод `scrollTo()` из класса `TreeView<T>`. Формат метода:

```
public void scrollTo(int index)
```

Прокрутим область к последнему видимому элементу иерархического списка:

```
treeView.scrollTo(treeView.getExpandedItemCount() - 1);
```

Прокрутим область к указанному элементу при условии, что он отображается (если элемент скрыт, то вначале следует отобразить списки всех родительских элементов, иначе метод `getRow()` вернет значение `-1`):

```
int index = treeView.getRow(treeItem2_11);
if (index != -1) treeView.scrollTo(index);
```

Задать обработчик события прокрутки к элементу в результате выполнения метода `scrollTo()` можно с помощью свойства `onScrollTo` из класса `TreeView<T>`. Методы доступа:

```
public void setOnScrollTo(EventHandler<ScrollToEvent<Integer>> value)
public EventHandler<ScrollToEvent<Integer>> getOnScrollTo()
public ObjectProperty<EventHandler<ScrollToEvent<Integer>>>
 onScrollToProperty()
```

Внутри обработчика с помощью метода `getScrollTarget()` можно получить индекс элемента. Формат метода:

```
public T getScrollTarget()
```

Пример создания обработчика:

```
treeView.setOnScrollTo(event -> {
 System.out.println("OnScrollTo " + event.getScrollTarget());
});
```

### **ЭЛЕКТРОННЫЙ АРХИВ**

Полный текст программы содержится в проекте `FX_CH14_02_TreeView`, приведенном в каталоге `CH14_tables` сопровождающего книгу электронного архива (см. приложение).

## **14.2.5. Управление выбором элементов**

Иерархические списки поддерживают выбор как одного элемента (режим `SelectionMode.SINGLE`, режим по умолчанию), так и нескольких элементов (режим `SelectionMode.MULTIPLE`). Для выбора нескольких элементов нужно дополнительно удерживать нажатой клавишу `<Ctrl>` или `<Shift>`. Управлять моделью выбора позволяет свойство `selectionModel` из класса `TreeView<T>`. Методы доступа:

```
import javafx.scene.control.SelectionModel;
public final void setSelectionModel(
 MultipleSelectionModel<TreeItem<T>> value)
public final MultipleSelectionModel<TreeItem<T>> getSelectionModel()
public final ObjectProperty<MultipleSelectionModel<TreeItem<T>>>
 selectionModelProperty()
```

Установим для списка режим выбора нескольких элементов:

```
treeView.getSelectionModel().setSelectionMode(SelectionMode.MULTIPLE);
```

Модель выбора описывается с помощью абстрактного класса `MultipleSelectionModel<TreeItem<T>>`. Все методы этого класса мы уже рассматривали в *разд. 13.9* и *13.3*, поэтому не будем повторяться. Получим индексы и значения всех выбранных элементов после нажатия кнопки:

```
button.setOnAction(event -> {
 ObservableList<Integer> arr =
 treeView.getSelectionModel().getSelectedIndices();
 System.out.println("Индексы выбранных элементов:");
 for (Integer index : arr) {
 System.out.println(index);
 }
 ObservableList<TreeItem<String>> arr2 =
 treeView.getSelectionModel().getSelectedItem();
 System.out.println("Значения выбранных элементов:");
 for (TreeItem<String> item : arr2) {
 System.out.println(item.getValue());
 }
});
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH14_02_TreeView`, приведенном в каталоге `CH14_tables` сопровождающего книгу электронного архива (см. *приложение*).

## 14.2.6. Управление фокусом ввода

Свойство `focusModel` из класса `TreeView<T>` содержит ссылку на объект, с помощью которого можно управлять фокусом ввода. Методы доступа:

```
public final void setFocusModel(FocusModel<TreeItem<T>> value)
public final FocusModel<TreeItem<T>> getFocusModel()
public final ObjectProperty<FocusModel<TreeItem<T>>>
 focusModelProperty()
```

Все методы из класса `FocusModel<T>` мы уже рассматривали в *разд. 13.8.3*, поэтому не будем повторяться. Пример получения индекса строки списка, находящейся в фокусе ввода, а также значения элемента:

```
FocusModel<TreeItem<String>> focusModel = treeView.getFocusModel();
System.out.println("Index " + focusModel.getFocusedIndex());
System.out.println("Item " + focusModel.getFocusedItem());
```

Пример назначения обработчика для отслеживания изменения индекса строки:

```
treeView.getFocusModel().focusedIndexProperty().addListener(
 (obj, oldValue, newValue) -> {
 System.out.println("Index " + newValue);
 });
```



### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH14_02_TreeView`, приведенном в каталоге `CH14_tables` сопровождающего книгу электронного архива (см. приложение).

## 14.2.7. Редактирование элементов

Чтобы пользователь мог редактировать значения элементов иерархического списка, необходимо присвоить свойству `editable` из класса `TreeView<T>` значение `true`. По умолчанию свойство имеет значение `false`. Методы доступа:

```
public final void setEditable(boolean value)
public final boolean isEditable()
public final BooleanProperty editableProperty()
```

Кроме изменения значения свойства `editable`, нужно задать значение свойству `cellFactory`. Можно реализовать свой собственный способ редактирования или воспользоваться готовыми классами из пакета `javafx.scene.control.cell`. Например, классом `TextFieldTreeCell<T>`, который в режиме редактирования отображает внутри ячейки текстовое поле. Из этого класса нам понадобится статический метод `forTreeView()`. Форматы метода:

```
import javafx.scene.control.cell.TextFieldTreeCell;
public static Callback<TreeView<String>, TreeCell<String>> forTreeView()
public static <T> Callback<TreeView<T>, TreeCell<T>> forTreeView(
 StringConverter<T> converter)
```

Первый формат метода работает только с типом `String`, а второй формат позволяет указать способ преобразования значения в строку и строки в значение.

Давайте создадим иерархический список и разрешим редактирование:

```
TreeView<String> treeView = new TreeView<String>(rootItem);
treeView.getSelectionModel().selectFirst();
treeView.setFocusModel().focus(0);
treeView.setEditable(true);
treeView.setCellFactory(TextFieldTreeCell.forTreeView());
```

Получить ссылку на редактируемый элемент позволяет свойство `editingItem` из класса `TreeView<T>`. Если редактируемого элемента нет, то свойство будет иметь значение `null`. Методы доступа:

```
public final TreeItem<T> getEditingItem()
public final ReadOnlyObjectProperty<TreeItem<T>> editingItemProperty()
```

Перейти в режим редактирования из программы позволяет метод `edit()` из класса `TreeView<T>`. Обратите внимание: строка списка должна быть в области видимости, иначе эффекта не будет. Формат метода:

```
public void edit(TreeItem<T> item)
```

Если изменения производились способом, который не отслеживается, то обновить видимую часть списка можно с помощью метода `refresh()`. Метод доступен, начиная с версии 8.60. Формат метода:

```
public void refresh()
```

Класс `TreeView<T>` содержит следующие статические методы, возвращающие типы событий редактирования списка:

- `editAnyEvent()` — любое событие редактирования. Формат метода:

```
public static <T> EventType<TreeView.EditEvent<T>> editAnyEvent()
```

- `editStartEvent()` — событие начала редактирования (пользователь выполнил двойной щелчок левой кнопкой мыши на строке или выделил строку и нажал клавишу `<Enter>`). Формат метода:

```
public static <T> EventType<TreeView.EditEvent<T>>
 editStartEvent()
```

- `editCommitEvent()` — событие успешного окончания редактирования (например, пользователь ввел значение в текстовое поле и нажал клавишу `<Enter>`). Формат метода:

```
public static <T> EventType<TreeView.EditEvent<T>>
 editCommitEvent()
```

- `editCancelEvent()` — событие отказа от редактирования (например, пользователь внутри текстового поля нажал клавишу `<Esc>` или выделил другую строку в режиме редактирования). Формат метода:

```
public static <T> EventType<TreeView.EditEvent<T>>
 editCancelEvent()
```

Пример назначения обработчика и получения значений:

```
treeView.addHandler(TreeView.<String>editCommitEvent(), event -> {
 System.out.println(event.getNewValue());
 System.out.println(event.getOldValue());
 System.out.println(event.getTreeItem());
 System.out.println(event.getSource());
});
```

Внутри обработчика события через параметр `event` доступен экземпляр статического класса `TreeView.EditEvent<T>`, который имеет следующие методы:

- `getNewValue()` — возвращает новое значение при успешном окончании редактирования. Внутри обработчиков других событий метод возвращает значение `null`. Формат метода:

```
public T getNewValue()
```

- `getOldValue()` — возвращает старое значение или значение `null`. Формат метода:

```
public T getOldValue()
```

- `getTreeItem()` — возвращает ссылку на редактируемый элемент списка. Формат метода:

```
public TreeItem<T> getTreeItem()
```

- `getSource()` — возвращает ссылку на объект списка. Формат метода:

```
public TreeView<T> getSource()
```

Управлять обработчиками событий редактирования позволяют также следующие свойства из класса `TreeView<T>`:

- ❑ `onEditStart` — назначает обработчик события начала редактирования. Методы доступа:

```
public final void setOnEditStart(
 EventHandler<TreeView.EditEvent<T>> value)
public final EventHandler<TreeView.EditEvent<T>> getOnEditStart()
public final ObjectProperty<EventHandler<TreeView.EditEvent<T>>>
 onEditStartProperty()
```

- ❑ `onEditCommit` — назначает обработчик события успешного окончания редактирования. Методы доступа:

```
public final void setOnEditCommit(
 EventHandler<TreeView.EditEvent<T>> value)
public final EventHandler<TreeView.EditEvent<T>>
 getOnEditCommit()
public final ObjectProperty<EventHandler<TreeView.EditEvent<T>>>
 onEditCommitProperty()
```

- ❑ `onEditCancel` — назначает обработчик события отказа от редактирования. Методы доступа:

```
public final void setOnEditCancel(
 EventHandler<TreeView.EditEvent<T>> value)
public final EventHandler<TreeView.EditEvent<T>>
 getOnEditCancel()
public final ObjectProperty<EventHandler<TreeView.EditEvent<T>>>
 onEditCancelProperty()
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH14_02_TreeView`, приведенном в каталоге `CH14_tables` сопровождающего книгу электронного архива (см. приложение).

## 14.2.8. Класс `CheckBoxTreeItem<T>`: флажок в качестве элемента списка

Класс `CheckBoxTreeItem<T>` позволяет использовать флажок в качестве элемента иерархического списка (рис. 14.4). Если установить флажок в родительском элементе, то автоматически будут установлены флажки во всех дочерних элементах. Если не установлен флажок в каком-либо дочернем элементе, то флажок в родительском элементе будет находиться в состоянии `indeterminate`. Если сбросить флажок в родительском элементе, то автоматически будут сброшены флажки во всех дочерних элементах. Такое поведение можно использовать для выбора компонентов при установке, а также для управления различными настройками приложения.

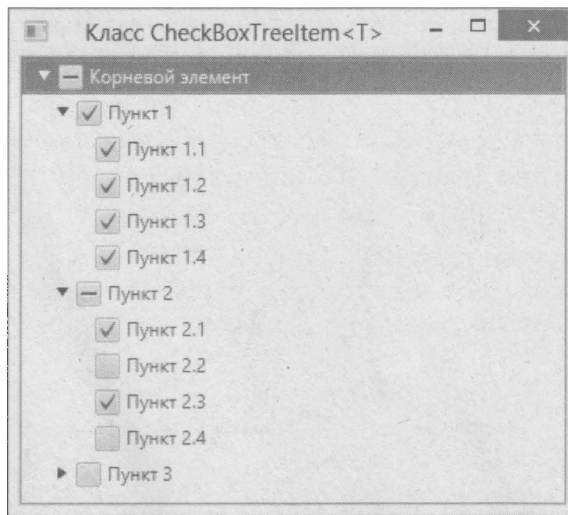


Рис. 14.4. Класс `CheckBoxTreeItem<T>`: флажок в качестве элемента списка

**Инструкция импорта класса `CheckBoxTreeItem<T>`:**

```
// Модуль javafx.controls
import javafx.scene.control.CheckBoxTreeItem;
```

**Иерархия наследования:**

`Object` – `TreeItem<T>` – `CheckBoxTreeItem<T>`

**Конструкторы класса:**

```
CheckBoxTreeItem()
CheckBoxTreeItem(T value)
CheckBoxTreeItem(T value, Node graphic)
CheckBoxTreeItem(T value, Node graphic, boolean selected)
CheckBoxTreeItem(T value, Node graphic, boolean selected,
 boolean independent)
```

**Первый конструктор создает элемент с настройками по умолчанию. Создадим корневой элемент и раскроем список дочерних элементов:**

```
var rootItem = new CheckBoxTreeItem<String>();
rootItem.setValue("Корневой элемент");
rootItem.setExpanded(true);
```

**Второй конструктор позволяет задать текст надписи. Создадим новый элемент и добавим его в список дочерних элементов:**

```
var treeItem = new CheckBoxTreeItem<String>("Пункт 1");
treeItem.setExpanded(true);
rootItem.getChildren().add(treeItem);
```

**В остальных конструкторах параметр `graphic` задает узел, который будет отображаться слева от надписи, а параметр `selected` позволяет установить флажок, пере-**

дав значение `true`. Если в параметре `independent` указать значение `true`, то изменение состояния флажка не будет влиять на состояние флажка родительского элемента, а также на состояния флажков дочерних элементов.

После создания объекта иерархического списка нужно вызвать метод `setCellFactory()` и передать ему объект, возвращаемый статическим методом `forTreeView()` из класса `CheckBoxTreeCell<T>`. Форматы метода:

```
import javafx.scene.control.cell.CheckBoxTreeCell;

public static <T> Callback<TreeView<T>, TreeCell<T>> forTreeView()
public static <T> Callback<TreeView<T>, TreeCell<T>> forTreeView(
 Callback<TreeItem<T>,
 ObservableValue<Boolean>> getSelectedProperty)
public static <T> Callback<TreeView<T>, TreeCell<T>> forTreeView(
 Callback<TreeItem<T>,
 ObservableValue<Boolean>> getSelectedProperty,
 StringConverter<TreeItem<T>> converter)
```

При использовании первого формата метода `forTreeView()` все элементы списка должны быть объектами класса `CheckBoxTreeItem<T>`. Пример создания иерархического списка:

```
var treeView = new TreeView<String>(rootItem);
treeView.getSelectionModel().selectFirst();
treeView.setFocusModel().focus(0);
treeView.setCellFactory(CheckBoxTreeCell.<String>forTreeView());
```

Класс `CheckBoxTreeItem<T>` наследует все свойства и методы из класса `TreeItem<T>` (см. *разд. 14.2.2*) и добавляет следующие свойства:

☐ `selected` — содержит значение `true`, если флажок установлен, или значение `false` — в противном случае. Методы доступа:

```
public final void setSelected(boolean value)
public final boolean isSelected()
public final BooleanProperty selectedProperty()
```

☐ `indeterminate` — содержит значение `true`, если флажок находится в неопределенном состоянии, или значение `false` — в противном случае. Методы доступа:

```
public final void setIndeterminate(boolean value)
public final boolean isIndeterminate()
public final BooleanProperty indeterminateProperty()
```

☐ `independent` — если содержит значение `true`, то изменение состояния флажка не будет влиять на состояние флажка родительского элемента, а также на состояния флажков дочерних элементов. Методы доступа:

```
public final void setIndependent(boolean value)
public final boolean isIndependent()
public final BooleanProperty independentProperty()
```

Метод `checkBoxSelectionChangedEvent()` из класса `CheckBoxTreeItem<T>` возвращает тип события изменения свойств списка. Формат метода:

```
public static <T> EventType<CheckBoxTreeItem.TreeModificationEvent<T>>
 checkBoxSelectionChangedEvent()
```

Внутри обработчика события через параметр `event` доступен экземпляр класса `CheckBoxTreeItem.TreeModificationEvent<T>`, который содержит следующие методы:

```
public CheckBoxTreeItem<T> getTreeItem()
public boolean wasSelectionChanged()
public boolean wasIndeterminateChanged()
```

Метод `getTreeItem()` возвращает ссылку на элемент, состояние которого изменилось. Пример назначения обработчика и получения надписи, а также состояния элемента:

```
rootItem.addEventHandler(
 CheckBoxTreeItem.<String>checkBoxSelectionChangedEvent(), event -> {
 CheckBoxTreeItem<String> item = event.getTreeItem();
 if (item != null) {
 System.out.println("value: " + item.getValue());
 System.out.println("selected: " + item.isSelected());
 System.out.println("indeterminate: " + item.isIndeterminate());
 }
});
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH14_02_TreeView`, приведенном в каталоге `CH14_tables` сопровождающего книгу электронного архива (см. приложение).

## 14.3. Класс *TreeTableView<S>*: иерархический список со столбцами

Класс `TreeTableView<S>` реализует иерархический список со столбцами, как в таблице (рис. 14.5). Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.TreeTableView;
```

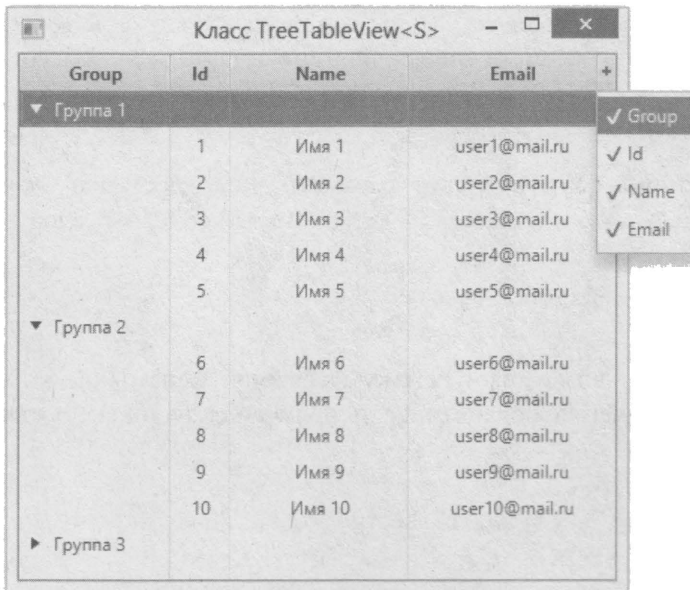
Иерархия наследования:

```
Object — Node — Parent — Region — Control — TreeTableView<S>
```

### 14.3.1. Создание списка и добавление элементов

Конструкторы класса `TreeTableView<S>`:

```
TreeTableView()
TreeTableView(TreeItem<S> root)
```



**Рис. 14.5.** Класс `TreeTableView<S>`: иерархический список со столбцами, как в таблице

Первый конструктор создает пустой список, а второй конструктор позволяет указать ссылку на корневой элемент. Давайте создадим класс `UserGroup`, который будем использовать в дальнейших примерах (листинг 14.2).

**Листинг 14.2.** Содержимое файла `src\application\UserGroup.java`

```
package application;

import javafx.beans.property.SimpleStringProperty;
import javafx.beans.property.StringProperty;

public class UserGroup extends User {

 private StringProperty group;

 public UserGroup(String group) {
 super(-1, "", "");
 this.group = new SimpleStringProperty(this, "group", group);
 }

 public UserGroup(Integer id, String name, String email) {
 super(id, name, email);
 this.group = new SimpleStringProperty(this, "group", "");
 }

 public String getGroup() {
 return group.get();
 }
}
```

```
public void setGroup(String group) {
 this.group.set(group);
}
public StringProperty groupProperty() {
 return group;
}

@Override
public String toString() {
 if (getGroup().isEmpty()) return super.toString();
 return "Group [" + getGroup() + "]";
}
}
```

Класс `UserGroup` наследует класс `User` (см. листинг 14.1) и добавляет одно свойство — `group`, описывающее название группы. Класс содержит два конструктора, которые позволяют создать либо группу, либо пользователя. Первый конструктор создает именованную группу, сохраняя название группы в свойстве `group`. Идентификатор пользователя в этом случае будет равен `-1`, а остальным свойствам из класса `User` присваивается пустая строка. Это позволит вывести в строке списка только название группы, а остальные ячейки оставить пустыми. Второй конструктор создает пользователя, а свойству `group` присваивается пустая строка. Это позволит в столбце выводить только названия групп, а остальные ячейки оставить пустыми.

Теперь создадим корневой элемент с группой. добавим его в иерархический список, раскроем список дочерних элементов и скроем корневой элемент:

```
var rootItem = new TreeItem<UserGroup>(
 new UserGroup("Корневой элемент"));
rootItem.setExpanded(true);

var treeTable = new TreeTableView<UserGroup>>();
treeTable.setRoot(rootItem);
treeTable.setShowRoot(false);
treeTable.getSelectionModel().setSelectionMode(SelectionMode.SINGLE);
treeTable.getSelectionModel().selectFirst();
treeTable.setColumnResizePolicy(TreeTableView.CONSTRAINED_RESIZE_POLICY);
treeTable.setTableMenuButtonVisible(true);
```

Практически все методы, которые мы вызываем в этом примере, должны быть вам уже знакомы. Они используются либо в классе `TreeItem<S>`, либо в классе `TreeView<T>`, либо в классе `TableView<S>`. Метод `setExpanded()` управляет отображением или сокрытием дочерних элементов (мы их показываем). Метод `setRoot()` задает корневой элемент списка. Метод `setShowRoot()` управляет отображением корневого элемента (мы его скрываем). Метод `getSelectionModel()` возвращает модель выбора (мы устанавливаем режим выбора только одного элемента и делаем первый



элемент выбранным). Метод `setColumnResizePolicy()` задает политику изменения ширины столбцов (суммарная ширина всех столбцов будет равна ширине таблицы). Метод `setTableMenuButtonVisible()` отображает в строке заголовка кнопку, при нажатии на которую показывается меню с названиями столбцов. С помощью этого меню можно скрыть или отобразить столбец.

Каждый элемент списка описывается с помощью класса `TreeItem<S>`. Мы уже рассматривали этот класс в *разд. 14.2.2* и *14.2.3*, поэтому не будем повторяться. Давайте создадим две группы и добавим в них несколько пользователей:

```
var group1 = new TreeItem<UserGroup>(new UserGroup("Группа 1"));
group1.setExpanded(true);
for (int i = 1; i < 6; i++) {
 group1.getChildren().add(new TreeItem<UserGroup>(
 new UserGroup(i, "Имя " + i, "user" + i + "@mail.ru")));
}
rootItem.getChildren().add(group1);

var group2 = new TreeItem<UserGroup>(new UserGroup("Группа 2"));
group2.setExpanded(true);
for (int i = 6; i < 11; i++) {
 group2.getChildren().add(new TreeItem<UserGroup>(
 new UserGroup(i, "Имя " + i, "user" + i + "@mail.ru")));
}
rootItem.getChildren().add(group2);
```

Как видно из примера, принцип добавления элементов ничем не отличается от принципа добавления элементов в классе `TreeView<T>` (см. *разд. 14.2.2*). Элемент описывается с помощью класса `TreeItem<S>`, а для прикрепления элемента к группе нужно вызвать метод `getChildren()` и добавить элемент в список. Затем группу добавляем в список дочерних элементов корневого элемента.

Класс `TreeTableView<S>` содержит следующие основные свойства:

❑ `root` — содержит ссылку на корневой элемент или значение `null`. Методы доступа:

```
public final void setRoot(TreeItem<S> value)
public final TreeItem<S> getRoot()
public final ObjectProperty<TreeItem<S>> rootProperty()
```

❑ `showRoot` — если свойство имеет значение `true` (значение по умолчанию), то корневой элемент будет отображаться, а если `false` — то он будет скрыт. Методы доступа:

```
public final void setShowRoot(boolean value)
public final boolean isShowRoot()
public final BooleanProperty showRootProperty()
```

❑ `expandedItemCount` — содержит количество раскрытых элементов. Методы доступа:

```
public final int getExpandedItemCount()
public final ReadOnlyIntegerProperty expandedItemCountProperty()
```

Получить информацию об элементе внутри иерархического списка позволяют следующие методы из класса `TreeTableView<S>`:

- ❑ `getRow()` — возвращает индекс позиции указанного элемента или значение `-1`, если элемент не отображается. Формат метода:

```
public int getRow(TreeItem<S> item)
```

- ❑ `getTreeItem()` — возвращает ссылку на элемент, находящийся в списке на указанной позиции, или значение `null`, если элемента нет. Формат метода:

```
public TreeItem<S> getTreeItem(int row)
```

- ❑ `getTreeItemLevel()` — возвращает уровень вложенности для указанного элемента или значение `-1`. Формат метода:

```
public int getTreeItemLevel(TreeItem<?> node)
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH14_03_TreeTableView`, приведенном в каталоге `CH14_tables` сопровождающего книгу электронного архива (см. *приложение*).

## 14.3.2. Класс `TreeTableColumn<S, T>`: создание столбцов

Просто добавить элементы в компонент недостаточно. Предварительно нужно описать столбцы и назначить правила отображения значений в ячейках столбца. Описание столбцов выполняется с помощью класса `TreeTableColumn<S, T>`. Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.TreeTableColumn;
```

Иерархия наследования:

```
Object — TableColumnBase<TreeItem<S>, T> — TreeTableColumn<S, T>
```

Конструкторы класса:

```
TreeTableColumn()
TreeTableColumn(String text)
```

Параметр `text` задает название столбца, которое отобразится в строке заголовка. Изменить или получить это название в дальнейшем позволяет свойство `text` из класса `TableColumnBase<TreeItem<S>, T>`. Методы доступа:

```
public final void setText(String value)
public final String getText()
public final StringProperty textProperty()
```

Задать правило отображения значений в ячейках столбца позволяет свойство `cellValueFactory` из класса `TreeTableColumn<S, T>`. Методы доступа:

```
import javafx.util.Callback;
import javafx.scene.control.TreeTableColumn.CellDataFeatures;
import javafx.beans.value.ObservableValue;
public final void setCellValueFactory(
 Callback<TreeTableColumn.CellDataFeatures<S, T>,
 ObservableValue<T>> value)
public final Callback<TreeTableColumn.CellDataFeatures<S, T>,
 ObservableValue<T>> getCellValueFactory()
public final ObjectProperty<
 Callback<TreeTableColumn.CellDataFeatures<S, T>,
 ObservableValue<T>>> cellValueFactoryProperty()
```

Первый способ создать необходимый объект заключается в использовании класса `TreeItemPropertyValueFactory<S, T>`. Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.cell.TreeItemPropertyValueFactory;
```

Конструктор класса:

```
TreeItemPropertyValueFactory(String property)
```

Параметр `property` задает название JavaFX-свойства в виде строки. Внутри класса должен быть метод с названием `<property>Property()`, который возвращает ссылку на JavaFX-свойство, — например, `nameProperty()` для свойства `name`. Давайте создадим столбец и свяжем его со свойством `name` из класса `UserGroup`:

```
var columnName = new TreeTableColumn<UserGroup, String>("Name");
columnName.setCellValueFactory(
 new TreeItemPropertyValueFactory<UserGroup, String>("name"));
```

К этому способу можно также обратиться, если по каким-либо причинам мы не хотим или не можем использовать JavaFX-свойства внутри класса. Если в классе существует метод с названием `<property>Property()`, то будет вызываться именно он, иначе выполняется поиск методов с названиями `get<Property>()` или `is<Property>()`. Если методов с такими названиями в классе нет, то генерируется исключение. Следует учитывать, что при использовании обычных типов данных изменение значения поля внутри класса не приведет к автоматическому изменению значения в ячейке компонента. Отслеживать все изменения нужно будет самостоятельно.

Второй способ заключается в реализации интерфейса `Callback<P, R>`. Внутри метода `call()` мы должны вернуть ссылку на JavaFX-свойство, значение которого должно отображаться в ячейке. Так как метод `call()` возвращает объект `ObservableValue<T>`, то будет автоматически добавлен обработчик события изменения значения свойства. Поэтому любое изменение значения свойства внутри класса приведет к изменению значения внутри ячейки таблицы. Пример:

```
var columnName = new TreeTableColumn<UserGroup, String>("Name");
columnName.setCellValueFactory(
 new Callback<CellDataFeatures<UserGroup, String>,
 ObservableValue<String>>() {
 public ObservableValue<String>
 call(CellDataFeatures<UserGroup, String> cellData) {
 return cellData.getValue().getValue().nameProperty();
 }
});
```

Вместо такого громоздкого кода можно использовать лямбда-выражение:

```
var columnName = new TreeTableColumn<UserGroup, String>("Name");
columnName.setCellValueFactory(
 cellData -> cellData.getValue().getValue().nameProperty());
```

Класс `TreeTableColumn.CellDataFeatures<S, T>` содержит следующие методы:

❑ `getValue()` — возвращает ссылку на текущий элемент списка (в нашем случае — на экземпляр класса `TreeItem<UserGroup>`). Формат метода:

```
public TreeItem<S> getValue()
```

❑ `getTreeTableView()` — возвращает ссылку на объект компонента. Формат метода:

```
public TreeTableView<S> getTreeTableView()
```

❑ `getTreeTableColumn()` — возвращает ссылку на объект столбца. Формат метода:

```
public TreeTableColumn<S, T> getTreeTableColumn()
```

При использовании свойств числовых типов (например, `IntegerProperty` и `DoubleProperty`) следует учитывать, что они реализуют интерфейс `ObservableValue<Number>`, а не интерфейсы `ObservableValue<Integer>` и `ObservableValue<Double>`. Поэтому лучше изначально создавать свойство с типом `ObjectProperty<Integer>` или при создании столбца указывать тип `Number` вместо типа `Integer`:

```
var columnId = new TreeTableColumn<UserGroup, Number>("Id");
```

Давайте опишем все столбцы из нашего примера:

```
var columnGroup = new TreeTableColumn<UserGroup, String>("Group");
columnGroup.setCellValueFactory(
 cellData -> cellData.getValue().getValue().groupProperty());
```

```
var columnId = new TreeTableColumn<UserGroup, Number>("Id");
columnId.setCellValueFactory(
 cellData -> cellData.getValue().getValue().idProperty());
```

```
var columnName = new TreeTableColumn<UserGroup, String>("Name");
columnName.setCellValueFactory(
 cellData -> cellData.getValue().getValue().nameProperty());
```

```
var columnEmail = new TreeTableColumn<UserGroup, String>("Email");
columnEmail.setCellValueFactory(
 cellData -> cellData.getValue().getValue().emailProperty());
```

Столбцы мы описали, теперь их нужно добавить в список столбцов компонента `TreeTableView<S>`. Получить ссылку на этот список позволяет метод `getColumns()`. Формат метода:

```
public final ObservableList<TreeTableColumn<S, ?>> getColumns()
```

**Пример добавления столбцов:**

```
treeTable.getColumns().add(columnGroup);
treeTable.getColumns().add(columnId);
treeTable.getColumns().add(columnName);
treeTable.getColumns().add(columnEmail);
```

Свойство `treeColumn` из класса `TreeTableView<S>` задает столбец, внутри которого будет расположена стрелка, позволяющая скрыть или отобразить список дочерних элементов. По умолчанию свойство имеет значение `null`, а стрелка отображается в первом (крайнем левом) столбце. Методы доступа:

```
public final void setTreeColumn(TreeTableColumn<S, ?> value)
public final TreeTableColumn<S, ?> getTreeColumn()
public final ObjectProperty<TreeTableColumn<S, ?>> treeColumnProperty()
```

Добавим стрелку в столбец с названием группы:

```
treeTable.setTreeColumn(columnGroup);
```

### **ЭЛЕКТРОННЫЙ АРХИВ**

Полный текст программы содержится в проекте `FX_CH14_03_TreeTableView`, приведенном в каталоге `CH14_tables` сопровождающего книгу электронного архива (см. *приложение*).

## **14.3.3. Объединение столбцов в группу**

Класс `TreeTableColumn<S, T>` также содержит метод `getColumns()`, но он используется для создания группы столбцов. Формат метода:

```
public final ObservableList<TreeTableColumn<S, ?>> getColumns()
```

**Пример создания группы столбцов:**

```
var treeTableView = new TreeTableView<UserGroup>();
treeTableView.setPlaceholder(new Label("Нет элементов"));
treeTableView.setPrefHeight(150.0);

var group =
 new TreeTableColumn<UserGroup, Integer>("Группа столбцов");
var column1 = new TreeTableColumn<UserGroup, String>("Name");
column1.setPrefWidth(200.0);
var column2 = new TreeTableColumn<UserGroup, String>("Email");
column2.setPrefWidth(200.0);
```

```
group.getColumns().add(column1);
group.getColumns().add(column2);

treeTableView.getColumns().add(group);
```

В этом примере мы изменили значение свойства `placeholder` из класса `TreeTableView<S>`, которое задает узел, отображающийся, если компонент не содержит списка или список пуст. Методы доступа:

```
public final void setPlaceholder(Node value)
public final Node getPlaceholder()
public final ObjectProperty<Node> placeholderProperty()
```

Получить ссылку на объект группы через объект столбца позволяет свойство `parentColumn` из класса `TableColumnBase<TreeItem<S>, T>`. Если столбец не принадлежит группе, то свойство содержит значение `null`. Методы доступа:

```
public final TableColumnBase<TreeItem<S>, ?> getParentColumn()
public final ReadOnlyObjectProperty< TableColumnBase<TreeItem<S>, ?> >
 parentColumnProperty()
```

**Пример:**

```
System.out.println(column1.getParentColumn().getText());
// Группа столбцов
System.out.println(columnGroup.getParentColumn()); // null
```

Для получения ссылки на объект компонента через объект столбца предназначено свойство `treeTableView` из класса `TreeTableColumn<S, T>`. Если столбец не принадлежит компоненту (например, объект столбца создан, но не добавлен в список столбцов), то свойство содержит значение `null`. Методы доступа:

```
public final TreeTableView<S> getTreeTableView()
public final ReadOnlyObjectProperty<TreeTableView<S>>
 treeTableViewProperty()
```

**Пример:**

```
System.out.println(group.getTreeTableView());
// TreeTableView@239b8195[styleClass=tree-table-view]
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH14_03_TreeTableView`, приведенном в каталоге `CH14_tables` сопровождающего книгу электронного архива (см. приложение).

## 14.3.4. Управление размерами столбцов

Управлять шириной столбца позволяют следующие свойства из класса `TableColumnBase<TreeItem<S>, T>`:

□ `width` — содержит текущую ширину столбца. Методы доступа:

```
public final double getWidth()
public final ReadOnlyDoubleProperty widthProperty()
```

❑ **prefWidth** — предпочтительная ширина столбца. Методы доступа:

```
public final void setPrefWidth(double value)
public final double getPrefWidth()
public final DoubleProperty prefWidthProperty()
```

❑ **minWidth** — минимальная ширина столбца. Методы доступа:

```
public final void setMinWidth(double value)
public final double getMinWidth()
public final DoubleProperty minWidthProperty()
```

❑ **maxWidth** — максимальная ширина столбца. Методы доступа:

```
public final void setMaxWidth(double value)
public final double getMaxWidth()
public final DoubleProperty maxWidthProperty()
```

❑ **resizable** — если свойство имеет значение `true`, то ширину столбца можно менять (пользователь может изменить ширину столбца, взявшись мышью за границу в строке заголовка столбцов), а если `false` — то ширину изменить будет нельзя. Методы доступа:

```
public final void setResizable(boolean value)
public final boolean isResizable()
public final BooleanProperty resizableProperty()
```

Укажем ширину столбца с идентификатором:

```
columnId.setMaxWidth(80.0);
columnId.setMinWidth(40.0);
columnId.setPrefWidth(80.0);
```

Управлять размерами позволяют следующие свойства из класса `TreeTableView<S>`:

❑ **columnResizePolicy** — задает политику изменения ширины столбцов. Методы доступа:

```
import javafx.scene.control.TreeTableView.ResizeFeatures;
import javafx.util.Callback;
public final void setColumnResizePolicy(
 Callback<TreeTableView.ResizeFeatures, Boolean> callback)
public final Callback<TreeTableView.ResizeFeatures, Boolean>
 getColumnResizePolicy()
public final ObjectProperty<
 Callback<TreeTableView.ResizeFeatures, Boolean>>
 columnResizePolicyProperty()
```

Можно самостоятельно реализовать интерфейс `Callback` или воспользоваться следующими статическими константами из класса `TreeTableView<S>`:

- `UNCONSTRAINED_RESIZE_POLICY` — после изменения ширины столбца сдвигает все последующие столбцы на это расстояние вправо или влево (значение по умолчанию);

- `CONSTRAINED_RESIZE_POLICY` — при изменении ширины столбца суммарная ширина всех столбцов будет равна ширине компонента.

Пример:

```
treeTable.setColumnResizePolicy(
 TreeTableView.CONSTRAINED_RESIZE_POLICY);
treeTable.setPrefWidth(350.0);
```

- `fixedCellSize` — позволяет задать фиксированную высоту строк. Если свойство имеет отрицательное значение или 0, то фиксированная высота не используется.

Методы доступа:

```
public final void setFixedCellSize(double value)
public final double getFixedCellSize()
public final DoubleProperty fixedCellSizeProperty()
```

Изменить ширину указанного столбца на некоторую величину `delta` позволяет метод `resizeColumn()` из класса `TreeTableView<S>`. Формат метода:

```
public boolean resizeColumn(TreeTableColumn<S, ?> column, double delta)
```

Уменьшим ширину столбца с именем пользователя на 5 пх:

```
treeTable.resizeColumn(columnName, -5.0);
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH14_03_TreeTableView`, приведенном в каталоге `CH14_tables` сопровождающего книгу электронного архива (см. приложение).

## 14.3.5. Изменение видимости столбца

Управлять видимостью столбца на экране позволяет свойство `visible` из класса `TableColumnBase<TreeItem<S>, T>`. Если свойство имеет значение `true`, то столбец отображается на экране, а если `false` — то он будет скрыт. Методы доступа:

```
public final void setVisible(boolean value)
public final boolean isVisible()
public final BooleanProperty visibleProperty()
```

Изменим видимость столбца с идентификатором после нажатия кнопки:

```
button.setOnAction(event -> {
 columnId.setVisible(!columnId.isVisible());
});
```

Если нужно, чтобы пользователь самостоятельно мог скрывать и отображать столбцы, можно присвоить свойству `tableMenuButtonVisible` из класса `TreeTableView<S>` значение `true` (по умолчанию свойство имеет значение `false`). В этом случае в заголовке отобразится кнопка, при нажатии на которую будет выведено меню с названиями столбцов (см. рис. 14.5). Чтобы скрыть столбец, дос-



таточно в этом меню сбросить флажок напротив его названия, а чтобы снова отобразить — установить этот флажок. Методы доступа:

```
public final void setTableMenuButtonVisible(boolean value)
public final boolean isTableMenuButtonVisible()
public final BooleanProperty tableMenuButtonVisibleProperty()
```

**Пример:**

```
treeTable.setTableMenuButtonVisible(true);
```

Получить информацию о видимых столбцах позволяют следующие методы из класса `TreeTableView<S>`:

- ❑ `getVisibleLeafColumns()` — возвращает неизменяемый список, содержащий ссылки на отображаемые столбцы (получить список всех столбцов позволяет метод `getColumns()`). Формат метода:

```
public ObservableList<TreeTableColumn<S, ?>>
 getVisibleLeafColumns()
```

- ❑ `getVisibleLeafColumn()` — возвращает ссылку на столбец, отображаемый в позиции с индексом `column`, или значение `null`. Формат метода:

```
public TreeTableColumn<S, ?> getVisibleLeafColumn(int column)
```

- ❑ `getVisibleLeafIndex()` — возвращает индекс указанного столбца или значение `-1`, если столбец не отображается. Формат метода:

```
public int getVisibleLeafIndex(TreeTableColumn<S, ?> column)
```

## 14.3.6. Управление прокруткой

Управлять прокруткой позволяют следующие свойства из класса `TreeTableView<S>`:

- ❑ `onScrollTo` — задает обработчик события прокрутки к строке в результате выполнения метода `scrollTo()`. Методы доступа:

```
public void setOnScrollTo(
 EventHandler< ScrollToEvent<Integer> > value)
public EventHandler<ScrollToEvent<Integer>> getOnScrollTo()
public ObjectProperty<EventHandler<ScrollToEvent<Integer>>>
 onScrollToProperty()
```

Внутри обработчика с помощью метода `getScrollTarget()` можно получить индекс строки. Формат метода:

```
public T getScrollTarget()
```

**Пример создания обработчика:**

```
treeTable.setOnScrollTo(event -> {
 System.out.println("OnScrollTo " + event.getScrollTarget());
});
```

Формат метода `scrollTo()`:

```
public void scrollTo(int index)
```

Пример:

```
treeTable.scrollTo(0);
```

- `onScrollToColumn` — задает обработчик события прокрутки к столбцу в результате выполнения методов: `scrollToColumn()` и `scrollToColumnIndex()`. Методы доступа:

```
public void setOnScrollToColumn(
 EventHandler<ScrollToEvent<TreeTableColumn<S, ?>>> value)
public EventHandler<ScrollToEvent<TreeTableColumn<S, ?>>>
 getOnScrollToColumn()
public ObjectProperty<
 EventHandler<ScrollToEvent<TreeTableColumn<S, ?>>>>
 onScrollToColumnProperty()
```

Внутри обработчика с помощью метода `getScrollTarget()` можно получить ссылку на столбец. Пример назначения обработчика:

```
treeTable.setOnScrollToColumn(event -> {
 System.out.println("OnScrollToColumn " +
 event.getScrollTarget());
});
```

Форматы методов `scrollToColumn()` и `scrollToColumnIndex()`:

```
public void scrollToColumn(TreeTableColumn<S, ?> column)
public void scrollToColumnIndex(int columnIndex)
```

Методы выполняют прокрутку таким образом, чтобы указанный столбец оказался в зоне видимости. Пример:

```
treeTable.scrollToColumn(columnEmail);
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH14_03_TreeTableView`, приведенном в каталоге `CH14_tables` сопровождающего книгу электронного архива (см. приложение).

## 14.3.7. Класс `TreeTableViewSelectionModel<S>`: управление выбором строк и отдельных ячеек

Компонент `TreeTableView<S>` поддерживает выбор как одной строки (режим `SelectionMode.SINGLE`), так и нескольких строк (режим `SelectionMode.MULTIPLE`). Существует также режим выбора отдельных ячеек. Для выбора нескольких строк (или ячеек) нужно дополнительно удерживать нажатой клавишу `<Ctrl>` или `<Shift>`. Управлять моделью выбора позволяет свойство `selectionModel` из класса `TreeTableView<S>`. Методы доступа:

```

public final void setSelectionModel(
 TreeTableView.TreeTableViewSelectionModel<S> value)
public final TreeTableView.TreeTableViewSelectionModel<S>
 getSelectionModel()
public final ObjectProperty<TreeTableView.TreeTableViewSelectionModel<S>>
 selectionModelProperty()

```

**Установим режим выбора нескольких строк:**

```
treeTable.getSelectionModel().setSelectionMode(SelectionMode.MULTIPLE);
```

**Модель выбора описывается с помощью абстрактного статического класса `TreeTableViewSelectionModel<S>`. Иерархия наследования:**

```

Object — SelectionModel<T> — MultipleSelectionModel<T> —
 TableSelectionModel<TreeItem<S>> —
 TreeTableView.TreeTableViewSelectionModel<S>

```

**Класс `TreeTableViewSelectionModel<S>` наследует и реализует все методы из классов: `SelectionModel<T>` (см. *разд. 13.3*) и `MultipleSelectionModel<T>` (см. *разд. 13.9*). Все эти методы мы уже рассматривали, поэтому не будем повторяться.**

**Выделим первую строку (предыдущий выбор не сбрасывается в режиме `MULTIPLE`):**

```
treeTable.getSelectionModel().selectFirst();
```

**Получим индексы и значения выбранных строк после нажатия кнопки:**

```

button.setOnAction(event -> {
 ObservableList<Integer> arr =
 treeTable.getSelectionModel().getSelectedIndices();
 System.out.println("Индексы выбранных элементов:");
 for (Integer index : arr) {
 System.out.println(index);
 }
 ObservableList<TreeItem<UserGroup>> arr2 =
 treeTable.getSelectionModel().getSelectedItems();
 System.out.println("Значения выбранных элементов:");
 for (TreeItem<UserGroup> item : arr2) {
 System.out.println(item.getValue());
 }
});

```

**Помимо выбора всех ячеек строки, компонент `TreeTableView<S>` поддерживает режим выбора отдельных ячеек. Управлять этим режимом позволяет свойство `cellSelectionEnabled` из класса `TableSelectionModel<TreeItem<S>>`. Если свойство имеет значение `true`, то будет включен режим выбора отдельных ячеек. Значение по умолчанию: `false`. Методы доступа:**

```

public final void setCellSelectionEnabled(boolean value)
public final boolean isCellSelectionEnabled()
public final BooleanProperty cellSelectionEnabledProperty()

```

Пример включения режима выбора отдельных ячеек:

```
treeTable.getSelectionModel().setCellSelectionEnabled(true);
```

Управлять выбором отдельных ячеек позволяют следующие методы:

- ❑ **select()** — выбирает ячейку в строке с индексом `row` из столбца `column`. Предыдущий выбор не сбрасывается. Формат метода:

```
public void select(
 int row, TableColumnBase<TreeItem<S>, ?> column)
```

Пример:

```
treeTable.getSelectionModel().select(0, columnEmail);
```

- ❑ **selectAboveCell()** — выбирает ячейку выше. Предыдущий выбор не сбрасывается. Формат метода:

```
public void selectAboveCell()
```

- ❑ **selectBelowCell()** — выбирает ячейку ниже. Предыдущий выбор не сбрасывается. Формат метода:

```
public void selectBelowCell()
```

- ❑ **selectLeftCell()** — выбирает ячейку слева. Предыдущий выбор не сбрасывается. Формат метода:

```
public void selectLeftCell()
```

- ❑ **selectRightCell()** — выбирает ячейку справа. Предыдущий выбор не сбрасывается. Формат метода:

```
public void selectRightCell()
```

- ❑ **selectRange()** — выбирает диапазон ячеек. Предыдущий выбор не сбрасывается. Формат метода:

```
public void selectRange(
 int minRow, TableColumnBase<TreeItem<S>, ?> minColumn,
 int maxRow, TableColumnBase<TreeItem<S>, ?> maxColumn)
```

Пример:

```
treeTable.getSelectionModel().selectRange(
 0, columnId, 5, columnName);
```

- ❑ **clearAndSelect()** — выбирает ячейку в строке с индексом `row` из столбца `column`, при этом сбрасывая предыдущий выбор. Формат метода:

```
public void clearAndSelect(
 int row, TableColumnBase<TreeItem<S>, ?> column)
```

- ❑ **clearSelection()** — сбрасывает выбор указанной ячейки. Форматы метода:

```
public void clearSelection(
 int row, TableColumnBase<TreeItem<S>, ?> column)
```

- ❑ `isSelected()` — возвращает значение `true`, если указанная ячейка выбрана, и `false` — в противном случае. Формат метода:

```
public boolean isSelected(
 int row, TableColumnBase<TreeItem<S>, ?> column)
```

- ❑ `getSelectedCells()` — возвращает список всех выбранных ячеек. Формат метода:

```
import javafx.scene.control.TreeTablePosition;
public ObservableList<TreeTablePosition<S, ?>> getSelectedCells()
```

**Класс `TreeTablePosition<S, T>` содержит следующие методы:**

- ❑ `getRow()` — возвращает индекс строки. Формат метода:

```
public int getRow()
```

- ❑ `getTreeItem()` — возвращает ссылку на объект строки или значение `null`. Формат метода:

```
public final TreeItem<S> getTreeItem()
```

- ❑ `getColumn()` — возвращает индекс столбца. Формат метода:

```
public int getColumn()
```

- ❑ `getTableColumn()` — возвращает ссылку на объект столбца или значение `null`. Формат метода:

```
public final TableColumn<S, T> getTableColumn()
```

- ❑ `getTreeTableView()` — возвращает ссылку на объект компонента. Формат метода:

```
public final TreeTableView<S> getTreeTableView()
```

**Выведем информацию обо всех выбранных ячейках:**

```
ObservableList<TreeTablePosition<UserGroup, ?>> arr =
 treeTable.getSelectionModel().getSelectedCells();
for (TreeTablePosition<UserGroup, ?> pos : arr) {
 System.out.print(pos.getRow() + " ");
 System.out.print(pos.getColumn() + " ");
 System.out.print(pos.getTableColumn().getText() + " ");
 System.out.print(
 pos.getTableColumn().getCellObservableValue(pos.getRow()).getValue());
 System.out.println(" " +
 pos.getTableColumn().getCellData(pos.getRow()));
}
```

**В этом примере для получения значения выбранной ячейки мы воспользовались следующими методами из класса `TreeTableColumn<S, T>`:**

- ❑ `getCellData()` — возвращает значение по индексу строки или указанному объекту. Форматы метода:

```
public final T getCellData(int index)
public final T getCellData(TreeItem<S> item)
```

- ❑ `getCellObservableValue()` — возвращает объект `ObservableValue<T>` по индексу строки или указанному объекту. Форматы метода:

```
public final ObservableValue<T> getCellObservableValue(int index)
public final ObservableValue<T> getCellObservableValue(
 TreeItem<S> item)
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH14_03_TreeTableView`, приведенном в каталоге `CH14_tables` сопровождающего книгу электронного архива (см. приложение).

## 14.3.8. Класс `TreeTableViewFocusModel<S>`: управление фокусом ввода

Свойство `focusModel` из класса `TreeTableView<S>` содержит ссылку на объект, с помощью которого можно управлять фокусом ввода. Методы доступа:

```
public final void setFocusModel(
 TreeTableView.TreeTableViewFocusModel<S> value)
public final TreeTableView.TreeTableViewFocusModel<S> getFocusModel()
public final ObjectProperty<TreeTableView.TreeTableViewFocusModel<S>>
 focusModelProperty()
```

Иерархия наследования класса `TreeTableViewFocusModel<S>`:

```
Object — FocusModel<T> —
 TableFocusModel<TreeItem<S>, TreeTableColumn<S, ?>> —
 TreeTableView.TreeTableViewFocusModel<S>
```

Класс `TreeTableViewFocusModel<S>` наследует все методы из класса `FocusModel<T>` (см. разд. 13.8.3) и добавляет следующие методы:

- ❑ `focus()` — устанавливает фокус ввода на указанную ячейку. Форматы метода:

```
public void focus(int row, TreeTableColumn<S, ?> column)
public void focus(TreeTablePosition<S, ?> pos)
```

Пример:

```
focusModel.focus(0, columnEmail);
```

- ❑ `focusAboveCell()` — устанавливает фокус на ячейку выше. Формат метода:

```
public void focusAboveCell()
```

- ❑ `focusBelowCell()` — устанавливает фокус на ячейку ниже. Формат метода:

```
public void focusBelowCell()
```

- ❑ `focusLeftCell()` — устанавливает фокус на ячейку слева. Формат метода:

```
public void focusLeftCell()
```

- ❑ `focusRightCell()` — устанавливает фокус на ячейку справа. Формат метода:

```
public void focusRightCell()
```

□ `isFocused()` — возвращает значение `true`, если указанная ячейка находится в фокусе ввода, и `false` — в противном случае. Формат метода:

```
public boolean isFocused(int row, TreeTableColumn<S, ?> column)
```

### ПРИМЕЧАНИЕ

Чтобы иметь возможность устанавливать фокус ввода на отдельную ячейку, нужно чтобы свойство `cellSelectionEnabled` имело значение `true`. В противном случае фокус ввода будет иметь вся строка, а не отдельная ячейка строки.

Отслеживать изменение фокуса ввода отдельной ячейки позволяет свойство `focusedCell`. Методы доступа:

```
public final TreeTablePosition<S, ?> getFocusedCell()
public final ReadOnlyObjectProperty<TreeTablePosition<S, ?>>
 focusedCellProperty()
```

Пример назначения обработчика:

```
treeTable.getFocusModel().focusedCellProperty().addListener(
 (obj, oldValue, newValue) -> {
 if (newValue != null) {
 TreeTableColumn<UserGroup, ?> column = newValue.getTableColumn();
 int row = newValue.getRow();
 if (column == null || row < 0) System.out.println("Cell null");
 else System.out.println("Cell " + column.getCellData(row));
 }
 });
```

Если свойство `cellSelectionEnabled` имеет значение `false`, то можно, например, отслеживать изменение индекса строки:

```
treeTable.getFocusModel().focusedIndexProperty().addListener(
 (obj, oldValue, newValue) -> {
 System.out.println("Index " + newValue);
 });
```

После нажатия кнопки выведем информацию о ячейках или строках, находящихся в фокусе ввода:

```
button.setOnAction(event -> {
 TreeTableViewFocusModel<UserGroup> focusModel =
 treeTable.getFocusModel();
 if (treeTable.getSelectionModel().isCellSelectionEnabled()) {
 System.out.println("Cell " + focusModel.getFocusedCell());
 }
 else {
 System.out.println("Index " + focusModel.getFocusedIndex());
 System.out.println("Item " + focusModel.getFocusedItem());
 }
});
```

**ЭЛЕКТРОННЫЙ АРХИВ**

Полный текст программы содержится в проекте `FX_CH14_03_TreeTableView`, приведенном в каталоге `CH14_tables` сопровождающего книгу электронного архива (см. приложение).

**14.3.9. Изменение свойств ячеек столбца**

Изменить характеристики ячеек столбца позволяет свойство `cellFactory` из класса `TreeTableColumn<S, T>`. Методы доступа:

```
import javafx.util.Callback;
import javafx.scene.control.TreeTableCell;
public final void setCellFactory(
 Callback<TreeTableColumn<S, T>, TreeTableCell<S, T>> value)
public final Callback<TreeTableColumn<S, T>, TreeTableCell<S, T>>
 getCellFactory()
public final ObjectProperty<
 Callback<TreeTableColumn<S, T>, TreeTableCell<S, T>> >
 cellFactoryProperty()
```

Иерархия наследования для класса `TreeTableCell<S, T>`:

```
Object – Node – Parent – Region – Control – Labeled – Cell<T> –
 IndexedCell<T> – TreeTableCell<S, T>
```

В нашем примере в столбце с идентификатором при выводе строки группы отображается значение `-1`. Давайте скроем это значение, а заодно выполним выравнивание идентификаторов по центру:

```
columnId.setCellFactory(
 new Callback<TreeTableColumn<UserGroup, Number>,
 TreeTableCell<UserGroup, Number>>() {
@Override
public TreeTableCell<UserGroup, Number> call(
 TreeTableColumn<UserGroup, Number> param) {
 return new TreeTableCell<UserGroup, Number>() {
 {
 setAlignment(Pos.TOP_CENTER);
 }
@Override
protected void updateItem(Number value, boolean empty) {
 super.updateItem(value, empty);
 if (value == null || empty) {
 setText("");
 }
 else {
 if (value.intValue() < 0) setText("");
 else setText(value.toString());
 }
 }
 }
```



```

 };
};
}
});

```

Точно таким же образом мы можем изменить цвет фона ячеек, задать характеристики шрифта, отобразить какой-либо узел рядом с текстом (или вместо текста), разрешить перенос текста на новую строку (`setWrapText(true)`) и др. Класс `TreeTableCell<S, T>` наследует все методы из класса `Labeled` (см. *разд. 11.3*), поэтому мы можем работать с ячейкой как с текстовой надписью.

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH14_03_TreeTableView`, приведенном в каталоге `CH14_tables` сопровождающего книгу электронного архива (см. *приложение*).

## 14.3.10. Изменение свойств ячеек строки

Изменить характеристики сразу всех ячеек строки позволяет свойство `rowFactory` из класса `TreeTableView<S>`. Методы доступа:

```

import javafx.util.Callback;
import javafx.scene.control.TreeTableRow;
public final void setRowFactory(
 Callback<TreeTableView<S>, TreeTableRow<S>> value)
public final Callback<TreeTableView<S>, TreeTableRow<S>> getRowFactory()
public final ObjectProperty<Callback<TreeTableView<S>, TreeTableRow<S>>>
 rowFactoryProperty()

```

**Иерархия наследования для класса `TreeTableRow<S>`:**

```

Object — Node — Parent — Region — Control — Labeled — Cell<S> —
 IndexedCell<S> — TreeTableRow<S>

```

**Давайте выделим строку, содержащую название группы, серым цветом:**

```

treeTable.setRowFactory(
 new Callback<TreeTableView<UserGroup>, TreeTableRow<UserGroup>>() {
@Override
public TreeTableRow<UserGroup> call(TreeTableView<UserGroup> p) {
 return new TreeTableRow<UserGroup>() {
@Override
protected void updateItem(UserGroup obj, boolean empty) {
 super.updateItem(obj, empty);
 if (obj == null || empty) {
 setStyle("");
 }
 else {
 if (obj.getGroup().isEmpty()) {
 setStyle("");

```

```

 } else {
 setStyle("-fx-background-color: silver;");
 }
 }

}

};

}

});

```

**ЭЛЕКТРОННЫЙ АРХИВ**

Полный текст программы содержится в проекте `FX_CH14_03_TreeTableView`, приведенном в каталоге `CH14_tables` сопровождающего книгу электронного архива (см. приложение).

### 14.3.11. Сортировка элементов

Компонент `TreeTableView<S>` по умолчанию поддерживает сортировку элементов внутри столбца. Для этого достаточно щелкнуть левой кнопкой мыши на заголовке столбца. В результате в заголовке отобразится черный треугольник, показывающий направление сортировки. Чтобы изменить направление сортировки, нужно щелкнуть левой кнопкой мыши на заголовке еще раз. Можно выполнять сортировку сразу по нескольким столбцам, удерживая нажатой клавишу `<Shift>` при щелчках левой кнопкой мыши.

Управлять процессом сортировки позволяют следующие свойства из класса `TreeTableView<S>`:

- onSort — позволяет назначить обработчик события сортировки. Методы доступа:

```
import javafx.scene.control.SortEvent;
public void setOnSort(
 EventHandler<SortEvent<TreeTableView<S>>> value)
public EventHandler<SortEvent<TreeTableView<S>>> getOnSort()
public ObjectProperty<EventHandler<SortEvent<TreeTableView<S>>>>
 onSortProperty()
```

### Пример назначения обработчика:

```
treeTable.setOnSort(event -> {
 System.out.println("Сортировка");
});
```

Можно также воспользоваться следующим кодом:

```
treeTable.addHandler(SortEvent.ANY, event -> {
 System.out.println("SortEvent.ANY");
});
```

- ❑ `sortPolicy` — задает политику сортировки. Методы доступа:

```
public final void setSortPolicy(
 Callback<TreeTableView<S>, Boolean> callback)
```

```
public final Callback<TreeTableView<S>, Boolean> getSortPolicy()
public final ObjectProperty<Callback<TreeTableView<S>, Boolean>>
 sortPolicyProperty()
```

### Запретим сортировку:

```
treeTable.setSortPolicy(table -> false);
```

- **comparator** — содержит способ сравнения значений при сортировке или значение null. Методы доступа:

```
public final Comparator<TreeItem<S>> getComparator()
public final ReadOnlyObjectProperty<Comparator<TreeItem<S>>>
 comparatorProperty()
```

- **sortMode** — задает режим сортировки. В параметре `value` указываются константы: `ALL_DESCENDANTS` (сортировка всех элементов; значение по умолчанию) или `ONLY_FIRST_LEVEL` (сортировка элементов только первого уровня вложенности) из перечисления `TreeSortMode`. Методы доступа:

```
import javafx.scene.control.TreeSortMode;
public final void setSortMode(TreeSortMode value)
public final TreeSortMode getSortMode()
public final ObjectProperty<TreeSortMode> sortModeProperty()
```

### Пример:

```
treeTable.setSortMode(TreeSortMode.ONLY_FIRST_LEVEL);
```

Получить ссылку на список со столбцами, требующими сортировки, позволяет метод `getSortOrder()` из класса `TreeTableView<S>`. Формат метода:

```
public final ObservableList<TreeTableColumn<S, ?>> getSortOrder()
```

В этот список мы можем добавить несколько столбцов, что приведет к сортировке сразу по двум и более столбцам. Вначале сортировка производится по первому столбцу в списке, а в случае равенства значений производится сортировка по следующему столбцу. Вызвать событие сортировки позволяет метод `sort()` из класса `TreeTableView<S>`. Формат метода:

```
public void sort()
```

**Пример запуска сортировки по возрастанию по столбцу Email после нажатия кнопки:**

```
button.setOnAction(event -> {
 columnEmail.setSortType(TreeTableColumn.SortType.ASCENDING);
 treeTable.getSortOrder().clear();
 treeTable.getSortOrder().add(columnEmail);
 treeTable.sort();
});
```

В этом примере с помощью свойства `sortType` из класса `TreeTableColumn<S, T>` мы указали направление сортировки. Методы доступа:

```
public final void setSortType(TreeTableColumn.SortType value)
public final TreeTableColumn.SortType getSortType()
public final ObjectProperty<TreeTableColumn.SortType> sortTypeProperty()
```

Перечисление `TreeTableColumn.SortType` содержит константы: `ASCENDING` (сортировка по возрастанию) и `DESCENDING` (сортировка по убыванию).

Класс `TreeTableColumn<S, T>` наследует следующие свойства из класса `TableColumnBase<TreeItem<S>, T>`:

- ❑ `sortable` — разрешает (значение `true`) или запрещает (значение `false`) сортировку для столбца. Методы доступа:

```
public final void setSortable(boolean value)
public final boolean isSortable()
public final BooleanProperty sortableProperty()
```

- ❑ `comparator` — задает способ сравнения значений столбца при сортировке. Методы доступа:

```
public final void setComparator(Comparator<T> value)
public final Comparator<T> getComparator()
public final ObjectProperty<Comparator<T>> comparatorProperty()
```

- ❑ `sortNode` — позволяет указать узел, который будет отображаться в заголовке столбца вместо значка в виде треугольника, показывающего направление сортировки. Методы доступа:

```
public final void setSortNode(Node value)
public final Node getSortNode()
public final ObjectProperty<Node> sortNodeProperty()
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH14_03_TreeTableView`, приведенном в каталоге `CH14_tables` сопровождающего книгу электронного архива (см. приложение).

## 14.3.12. Редактирование элементов

Чтобы пользователь мог редактировать значения элементов, необходимо присвоить свойству `editable` из класса `TreeTableView<S>` значение `true`. По умолчанию свойство имеет значение `false`. Методы доступа:

```
public final void setEditable(boolean value)
public final boolean isEditable()
public final BooleanProperty editableProperty()
```

Разрешим редактирование:

```
treeTable.setEditable(true);
```

Свойство `editable` есть также в классе `TreeTableColumn<S, T>` (наследуется из класса `TableColumnBase<TreeItem<S>, T>`). По умолчанию свойство имеет значение `true`.

Чтобы запретить редактирование ячеек столбца, нужно присвоить свойству значение `false`. Методы доступа:

```
public final void setEditable(boolean value)
public final boolean isEditable()
public final BooleanProperty editableProperty()
```

Кроме изменения значения свойства `editable`, нужно задать значение свойству `cellFactory`. Можно реализовать свой собственный способ редактирования или воспользоваться готовыми классами из пакета `javafx.scene.control.cell`. Например, классом `TextFieldTreeTableCell<S, T>`, который в режиме редактирования отображает внутри ячейки текстовое поле. Из этого класса нам понадобится статический метод `forTreeTableColumn()`. Форматы метода:

```
import javafx.scene.control.cell.TextFieldTreeTableCell;
public static <S> Callback<TreeTableColumn<S, String>,
 TreeTableCell<S, String>> forTreeTableColumn()
public static <S, T> Callback<TreeTableColumn<S, T>,
 TreeTableCell<S, T>> forTreeTableColumn(
 StringConverter<T> converter)
```

Первый формат метода работает только с типом `String`, а второй — позволяет указать способ преобразования значения в строку и строки в значение.

Запретим редактирование столбцов с группами и идентификаторами, а в ячейках столбцов с именем пользователя и E-mail при редактировании отобразим текстовое поле:

```
columnGroup.setEditable(false);
columnId.setEditable(false);
columnName.setCellFactory(
 TextFieldTreeTableCell.<UserGroup>forTreeTableColumn());
columnEmail.setCellFactory(
 TextFieldTreeTableCell.<UserGroup>forTreeTableColumn());
```

Класс `TreeTableColumn<S, T>` содержит следующие статические методы, возвращающие типы событий редактирования:

❑ `editAnyEvent()` — любое событие редактирования. Формат метода:

```
public static <S, T> EventType<
 TreeTableColumn.CellEditEvent<S, T> > editAnyEvent()
```

❑ `editStartEvent()` — событие начала редактирования (пользователь выполнил двойной щелчок левой кнопкой мыши на ячейке или выделил ячейку и нажал клавишу `<Enter>`). Формат метода:

```
public static <S, T> EventType<
 TreeTableColumn.CellEditEvent<S, T> > editStartEvent()
```

❑ `editCommitEvent()` — событие успешного окончания редактирования (например, пользователь ввел значение в текстовое поле и нажал клавишу `<Enter>`). Формат метода:

```
public static <S, T> EventType<
 TreeTableColumn.CellEditEvent<S, T> > editCommitEvent()
```

- ❑ `editCancelEvent()` — событие отказа от редактирования (например, пользователь внутри текстового поля нажал клавишу <Esc> или выделил другую ячейку в режиме редактирования). Формат метода:

```
public static <S, T> EventType<
 TreeTableColumn.CellEditEvent<S, T> > editCancelEvent()
```

Назначить обработчик для событий редактирования позволяет метод `addEventHandler()` из класса `TableColumnBase<TreeItem<S>, T>`. Формат метода:

```
public <E extends Event> void addEventHandler(
 EventType<E> eventType, EventHandler<E> eventHandler)
```

Для удаления обработчика предназначен метод `removeEventHandler()` из класса `TableColumnBase<TreeItem<S>, T>`. Формат метода:

```
public <E extends Event> void removeEventHandler(
 EventType<E> eventType, EventHandler<E> eventHandler)
```

Пример назначения обработчика и получения значений:

```
columnName.addEventHandler(
 TreeTableColumn.<UserGroup, String>editCommitEvent(), event -> {
 System.out.println(event.getNewValue());
 System.out.println(event.getOldValue());
 System.out.println(event.getRowValue());
 System.out.println(event.getTreeTableView());
 System.out.println(event.getTableColumn());
 System.out.println(event.getTreeTablePosition());
 System.out.println(event.getTreeTablePosition().getRow());
 System.out.println(event.getTreeTablePosition().getColumn());
 System.out.println(event.getTreeTableView().getEditingCell());
 });
```

Класс `TreeTableColumn.CellEditEvent<S, T>` содержит следующие методы:

- ❑ `getNewValue()` — возвращает новое значение при успешном окончании редактирования. Внутри обработчиков других событий метод возвращает значение `null`. Формат метода:

```
public T getNewValue()
```

- ❑ `getOldValue()` — возвращает старое значение или значение `null`. Формат метода:

```
public T getOldValue()
```

- ❑ `getRowValue()` — возвращает ссылку на объект строки с редактируемой ячейкой (в нашем примере `TreeItem<UserGroup>`). Формат метода:

```
public TreeItem<S> getRowValue()
```

- ❑ `getTreeTableView()` — возвращает ссылку на объект компонента. Формат метода:

```
public TreeTableView<S> getTreeTableView()
```

- ❑ `getTableColumn()` — возвращает ссылку на объект столбца. Формат метода:

```
public TreeTableColumn<S, T> getTableColumn()
```

- ❑ `getTreeTablePosition()` — возвращает ссылку на объект, с помощью которого можно получить информацию о положении редактируемой ячейки, — например, индексы строки и столбца. Формат метода:

```
public TreeTablePosition<S, T> getTreeTablePosition()
```

Получить информацию о положении редактируемой ячейки можно также с помощью свойства `editingCell` из класса `TreeTableView<S>`. Методы доступа:

```
public final TreeTablePosition<S, ?> getEditingCell()
public final ReadOnlyObjectProperty<TreeTablePosition<S, ?>>
 editingCellProperty()
```

Управлять обработчиками событий редактирования позволяют также следующие свойства из класса `TreeTableColumn<S, T>`:

- ❑ `onEditStart` — назначает обработчик события начала редактирования. Методы доступа:

```
public final void setOnEditStart(
 EventHandler<TreeTableColumn.CellEditEvent<S, T>> value)
public final EventHandler<TreeTableColumn.CellEditEvent<S, T>>
 getOnEditStart()
public final ObjectProperty<
 EventHandler<TreeTableColumn.CellEditEvent<S, T>>>
 onEditStartProperty()
```

- ❑ `onEditCommit` — назначает обработчик события успешного окончания редактирования. Методы доступа:

```
public final void setOnEditCommit(
 EventHandler<TreeTableColumn.CellEditEvent<S, T>> value)
public final EventHandler<TreeTableColumn.CellEditEvent<S, T>>
 getOnEditCommit()
public final ObjectProperty<
 EventHandler<TreeTableColumn.CellEditEvent<S, T>>>
 onEditCommitProperty()
```

**Обратите внимание:** если вы назначаете обработчик через свойство `onEditCommit`, то переопределяете обработчик по умолчанию, который изменяет значение элемента. Следовательно, нужно будет самостоятельно изменить значение, например, после проверки условия. Если надо просто отследить событие, а не переопределять обработчик по умолчанию, то можно для назначения обработчика воспользоваться методом `addEventHandler()` из класса `TableColumnBase<TreeItem<S>, T>`. Давайте назначим обработчик для столбца с именем пользователя и внутри обработчика проверим допустимость нового значения и либо откажемся от изменения значения (например, если строка описывает группу), либо изменим значение (например, если строка описывает пользователя):

```
columnName.setOnEditCommit(event -> {
 if (event.getNewValue() == null) treeTable.refresh();
 else {
 if (event.getRowValue().getValue().getGroup().isEmpty()) {
 event.getRowValue().getValue().setName(
 event.getNewValue());
 }
 else treeTable.refresh();
 }
});
```

В этом примере мы воспользовались методом `refresh()` из класса `TreeTableView<S>`. Этот метод вызывает обновление видимой части компонента. Метод доступен, начиная с версии 8.60. Формат метода:

```
public void refresh()
```

- ❑ `onEditCancel` — назначает обработчик события отказа от редактирования. Методы доступа:

```
public final void setOnEditCancel(
 EventHandler<TreeTableColumn.CellEditEvent<S, T>> value)
public final EventHandler<TreeTableColumn.CellEditEvent<S, T>>
 getOnEditCancel()
public final ObjectProperty<
 EventHandler<TreeTableColumn.CellEditEvent<S, T>>>
 onEditCancelProperty()
```

Перейти в режим редактирования ячейки из программы позволяет метод `edit()` из класса `TreeTableView<S>`. Обратите внимание: ячейка должна быть в области видимости, иначе эффекта не будет. Формат метода:

```
public void edit(int row, TreeTableColumn<S, ?> column)
```

Пример:

```
treeTable.edit(1, columnEmail);
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH14_03_TreeTableView`, приведенном в каталоге `CH14_tables` сопровождающего книгу электронного архива (см. приложение).

## 14.3.13. Изменение порядка следования столбцов

По умолчанию пользователь может изменить положение столбца, взявшись мышью за заголовок столбца и перетащив его в другое место. Чтобы запретить перетаскивание отдельного столбца, нужно присвоить значение `false` свойству `reorderable` из класса `TableColumnBase<TreeItem<S>, T>`. Свойство доступно, начиная с JavaFX 9. Методы доступа:



```
public final void setReorderable(boolean value)
public final boolean isReorderable()
public final BooleanProperty reorderableProperty()
```

**Запретим перетаскивание столбца с идентификатором:**

```
columnId.setReorderable(false);
```

Чтобы изменить порядок следования столбцов из программы, достаточно изменить порядок следования элементов внутри списка, возвращаемого методом `getColumns()` из класса `TreeTableView<S>`.

### **ЭЛЕКТРОННЫЙ АРХИВ**

Полный текст программы содержится в проекте `FX_CH14_03_TreeTableView`, приведенном в каталоге *CH14\_tables* сопровождающего книгу электронного архива (см. приложение).

## ГЛАВА 15



# Меню и всплывающие окна

Главное меню является основным компонентом пользовательского интерфейса, позволяющим компактно разместить множество команд, объединяя их в логические группы. Меню состоит из горизонтальной панели (реализуемой классом `MenuBar`), на которой расположены отдельные меню (реализуются с помощью класса `Menu`) верхнего уровня. Каждое меню может содержать множество пунктов (реализуемых с помощью класса `MenuItem`), разделители, а также вложенные меню (рис. 15.1). Пункт меню может содержать значок и текст, а также флажок или переключатель.

Помимо главного меню, в приложениях часто используются контекстные меню, которые обычно отображаются при щелчке правой кнопкой мыши в области компонента. Контекстное меню реализуется с помощью класса `ContextMenu` и прикрепляется к компоненту с помощью метода с названием `setContextMenu()` из класса `Control`, который наследуют все компоненты.

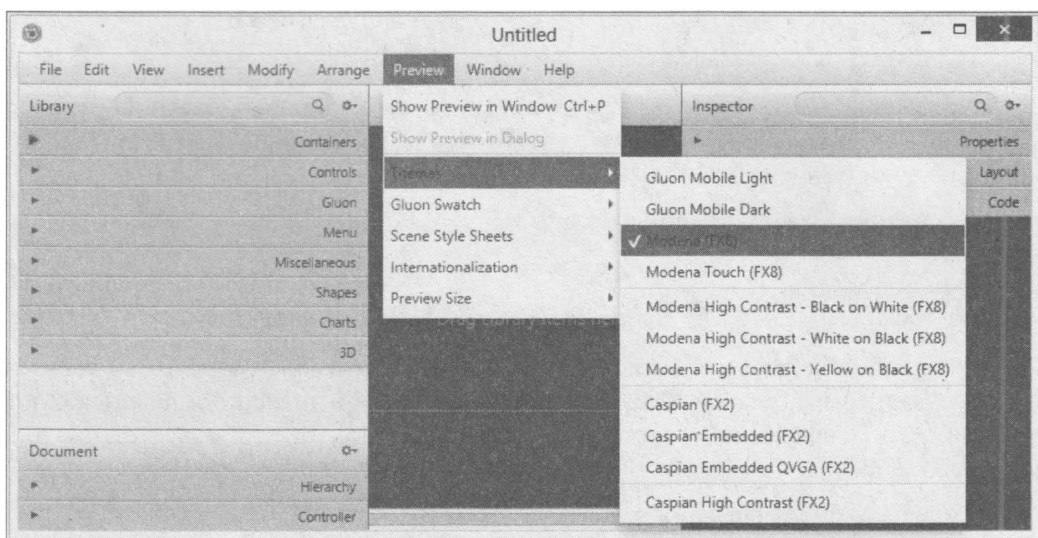


Рис. 15.1. Меню в программе Scene Builder

## 15.1. Класс *MenuBar*: панель меню

Класс `MenuBar` описывает горизонтальную панель меню, которую можно прикрепить к верхней части окна с помощью контейнера `BorderPane` (см. *разд. 3.14*). Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.MenuBar;
```

Иерархия наследования:

Object — Node — Parent — Region — Control — MenuBar

Конструкторы класса:

```
MenuBar()
MenuBar(Menu... menus)
```

Первый конструктор создает пустую панель, а второй конструктор позволяет сразу добавить отдельные меню на панель. Пример:

```
Menu fileMenu = new Menu("_File");
Menu editMenu = new Menu("_Edit");
MenuBar menuBar = new MenuBar(fileMenu, editMenu);
```

Чтобы добавить отдельные меню после создания объекта, нужно воспользоваться методом `getMenus()`, который возвращает список всех меню, прикрепленных к панели. Формат метода:

```
public final ObservableList<Menu> getMenus()
```

Пример:

```
Menu fileMenu = new Menu("_File");
Menu editMenu = new Menu("_Edit");
MenuBar menuBar = new MenuBar();
menuBar.getMenus().addAll(fileMenu, editMenu);
```

После создания панели мы можем прикрепить ее к верхней части контейнера `BorderPane`, а остальные компоненты разместить в центральной части, добавив их, например, в контейнер `VBox`:

```
BorderPane pane = new BorderPane();
pane.setTop(menuBar);
VBox vbox = new VBox(10.0);
pane.setCenter(vbox);
```

Класс `MenuBar` содержит свойство `useSystemMenuBar`, которое разрешает (значение `true`) или запрещает (значение `false`) использование панели в системном меню, при условии, что операционная система поддерживает такую возможность. Методы доступа:

```
public final void setUseSystemMenuBar(boolean value)
public final boolean isUseSystemMenuBar()
public final BooleanProperty useSystemMenuBarProperty()
```

**ЭЛЕКТРОННЫЙ АРХИВ**

Полный текст программы содержится в проекте `FX_CH15_01_MenuBar`, приведенном в каталоге `CH15_menu` сопровождающего книгу электронного архива (см. *приложение*).

## 15.2. Класс *Menu*: отдельное или вложенное меню

Класс `Menu` реализует отдельное меню на панели `MenuBar`, а также вложенное меню.

Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.Menu;
```

Иерархия наследования:

```
Object – MenuItem – Menu
```

Конструкторы класса:

```
Menu()
Menu(String text)
Menu(String text, Node graphic)
Menu(String text, Node graphic, MenuItem... items)
```

Первый конструктор создает отдельное меню с настройками по умолчанию:

```
Menu helpMenu = new Menu();
helpMenu.setText("_Help");
```

Второй конструктор позволяет указать текст надписи:

```
Menu fileMenu = new Menu("_File");
```

Внутри текста символ `_` (подчеркивание), указанный перед буквой, задает комбинацию клавиш быстрого доступа. В этом случае символ, перед которым указан символ подчеркивания, будет подчеркнут, что является подсказкой пользователю. При одновременном нажатии клавиши `<Alt>` и подчеркнутого символа (в нашем примере `<Alt>+<F>`) меню будет отображено.

В третьем и четвертом конструкторах параметр `graphic` задает узел, который будет отображаться перед надписью (например, изображение):

```
Menu editMenu = new Menu("_Edit",
 new ImageView(createImage(10, 10, Color.BLUE)));
```

Четвертый конструктор позволяет сразу добавить пункты в меню:

```
Menu runMenu = new Menu("_Run", null,
 new MenuItem("Run"), new MenuItem("Debug"));
```

Чтобы добавить пункты в меню после создания объекта, нужно воспользоваться методом `getItems()`, который возвращает список всех пунктов. Формат метода:

```
public final ObservableList<MenuItem> getItems()
```

**Пример:**

```
Menu fileMenu = new Menu("_File");
MenuItem openMenuItem = new MenuItem("Open...");
fileMenu.getItems().addAll(openMenuItem);
```

**Класс Menu наследует все свойства из класса MenuItem (см. разд. 15.3.1) и добавляет следующие свойства:**

- **showing** — содержит значение `true`, если меню отображается на экране, и `false` — если скрыто. Методы доступа:

```
public final boolean isShowing()
public final ReadOnlyBooleanProperty showingProperty()
```

**Пример назначения обработчика:**

```
fileMenu.showingProperty().addListener(
 (obj, oldValue, newValue) -> {
 System.out.println("showing " + newValue);
 });
```

- **onShowing** — задает обработчик события, возникающего перед отображением меню. Методы доступа:

```
public final void setOnShowing(EventHandler<Event> value)
public final EventHandler<Event> getOnShowing()
public final ObjectProperty<EventHandler<Event>>
 onShowingProperty()
```

**Пример назначения обработчика:**

```
fileMenu.setOnShowing(event -> {
 System.out.println("OnShowing");
});
```

- **onShown** — задает обработчик события отображения меню. Методы доступа:

```
public final void setOnShown(EventHandler<Event> value)
public final EventHandler<Event> getOnShown()
public final ObjectProperty<EventHandler<Event>>
 onShownProperty()
```

- **onHiding** — задает обработчик события, возникающего перед сокрытием меню. Методы доступа:

```
public final void setOnHiding(EventHandler<Event> value)
public final EventHandler<Event> getOnHiding()
public final ObjectProperty<EventHandler<Event>>
 onHidingProperty()
```

- **onHidden** — задает обработчик события сокрытия меню. Методы доступа:

```
public final void setOnHidden(EventHandler<Event> value)
public final EventHandler<Event> getOnHidden()
```

```
public final ObjectProperty<EventHandler<Event>>
 onHiddenProperty()
```

Назначить обработчик можно также с помощью метода `addEventHandler()`. Формат метода:

```
public <E extends Event> void addEventHandler(
 EventType<E> eventType, EventHandler<E> eventHandler)
```

В этом случае в параметре `eventType` указываются следующие константы:

```
public static final EventType<Event> ON_SHOWING
public static final EventType<Event> ON_SHOWN
public static final EventType<Event> ON_HIDING
public static final EventType<Event> ON_HIDDEN
```

Пример назначения обработчика для события отображения меню:

```
editMenu.addEventHandler(Menu.ON_SHOWN, event -> {
 System.out.println("ON_SHOWN");
});
```

Для удаления обработчика предназначен метод `removeEventHandler()`. Формат метода:

```
public <E extends Event> void removeEventHandler(
 EventType<E> eventType, EventHandler<E> eventHandler)
```

Отобразить меню из программы позволяет метод `show()`, а для сокрытия меню можно воспользоваться методом `hide()`. Меню автоматически скрывается при выборе пункта или при потере фокуса ввода. Форматы методов:

```
public void show()
public void hide()
```

При нажатии клавиши `<S>` отобразим меню, а при нажатии клавиши `<H>` — скроем:

```
window.addEventHandler(KeyEvent.KEY_PRESSED, event -> {
 if (event.getCode() == KeyCode.S) {
 this.editMenu.show();
 }
 else if (event.getCode() == KeyCode.H) {
 this.editMenu.hide();
 }
});
```

Чтобы иметь возможность получить ссылку на меню из любого места программы, нужно при создании объекта меню сохранить ссылку в поле класса. Пример объявления поля:

```
private Menu editMenu;
```

Класс `Menu` наследует класс `MenuItem` (см. *разд. 15.3.1*), а, следовательно, может исполнять роль пункта меню. Это обстоятельство позволяет создавать вложенные меню. Создадим меню со следующей иерархией:

```

Window
 New window
 Zoom
 Zoom In
 Zoom Out

```

В этой схеме пункт **Zoom** является вложенным меню. Пример реализации:

```

Menu windowMenu = new Menu("_Window");
MenuItem newWindowMenuItem = new MenuItem("New window");
Menu zoomMenu = new Menu("_Zoom");
windowMenu.getItems().addAll(newWindowMenuItem, zoomMenu);

MenuItem zoomInMenuItem = new MenuItem("Zoom In");
MenuItem zoomOutMenuItem = new MenuItem("Zoom Out");
zoomMenu.getItems().addAll(zoomInMenuItem, zoomOutMenuItem);

```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH15_01_MenuBar`, приведенном в каталоге *CH15\_меню* сопровождающего книгу электронного архива (см. *приложение*).

## 15.3. Пункты меню

После создания меню в него можно добавить пункты, которые реализуются с помощью классов `MenuItem` (обычный пункт), `CheckMenuItem` (пункт с флажком), `RadioMenuItem` (пункт с переключателем), `SeparatorMenuItem` (горизонтальная линия) или `CustomMenuItem` (произвольный пункт). Давайте рассмотрим эти классы.

### 15.3.1. Класс *MenuItem*: пункт меню

Класс `MenuItem` реализует пункт меню. Инструкция импорта:

```

// Модуль javafx.controls
import javafx.scene.control.MenuItem;

```

Иерархия наследования:

```
Object — MenuItem
```

Конструкторы класса:

```

MenuItem()
MenuItem(String text)
MenuItem(String text, Node graphic)

```

Первый конструктор создает объект с настройками по умолчанию:

```

MenuItem cutMenuItem = new MenuItem();
cutMenuItem.setText("Cut");
cutMenuItem.setGraphic(
 new ImageView(createImage(10, 10, Color.RED)));

```

Второй конструктор позволяет задать текст надписи:

```
MenuItem copyMenuItem = new MenuItem("Copy");
```

В первом параметре третьего конструктора указывается текст надписи, а во втором параметре — узел, который будет отображаться слева от надписи (например, значок):

```
MenuItem pasteMenuItem = new MenuItem("Paste",
 new ImageView(createImage(10, 10, Color.BLUE)));
```

После создания пунктов их нужно добавить в список дочерних элементов меню:

```
editMenu.getItems().addAll(cutMenuItem, copyMenuItem, pasteMenuItem);
```

Класс `MenuItem` содержит следующие основные свойства:

- ❑ `text` — содержит текст надписи. Внутри текста символ `_` (подчеркивание), указанный перед буквой, задает комбинацию клавиш быстрого доступа. В этом случае символ, перед которым указан символ подчеркивания, будет подчеркнут, что является подсказкой пользователю. При одновременном нажатии клавиши `<Alt>` и подчеркнутого символа пункт будет выбран. Нажатие комбинации клавиш быстрого доступа сработает только в том случае, если меню, в котором находится пункт, является активным. Методы доступа:

```
public final void setText(String value)
public final String getText()
public final StringProperty textProperty()
```

- ❑ `graphic` — задает узел (например, изображение), который будет отображаться слева от надписи. Значение по умолчанию: `null`. Методы доступа:

```
public final void setGraphic(Node value)
public final Node getGraphic()
public final ObjectProperty<Node> graphicProperty()
```

- ❑ `id` — задает идентификатор пункта или меню. Методы доступа:

```
public final void setId(String value)
public final String getId()
public final StringProperty idProperty()
```

- ❑ `style` — задает стиль. Методы доступа:

```
public final void setStyle(String value)
public final String getStyle()
public final StringProperty styleProperty()
```

Сделаем фон пункта серым:

```
aboutMenuItem.setStyle("-fx-background-color: silver;");
```

- ❑ `disable` — управляет доступностью пункта или меню. Если указать значение `true`, то пункт или меню станут недоступными. Методы доступа:



```
public final void setDisable(boolean value)
public final boolean isDisable()
public final BooleanProperty disableProperty()
```

- ❑ **visible** — управляет видимостью пункта или меню. Если указать значение `false`, то пункт или меню будут скрыты. Методы доступа:

```
public final void setVisible(boolean value)
public final boolean isVisible()
public final BooleanProperty visibleProperty()
```

- ❑ **parentMenu** — содержит ссылку на меню, внутри которого расположен пункт, или значение `null`. Методы доступа:

```
public final Menu getParentMenu()
public final ReadOnlyObjectProperty<Menu> parentMenuProperty()
```

- ❑ **parentPopup** — содержит ссылку на объект `ContextMenu`, внутри которого расположен пункт или меню. Методы доступа:

```
public final ContextMenu getParentPopup()
public final ReadOnlyObjectProperty<ContextMenu>
 parentPopupProperty()
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH15_01_MenuBar`, приведенном в каталоге `CH15_menu` сопровождающего книгу электронного архива (см. приложение).

## 15.3.2. Обработка выбора пункта меню

Назначить обработчик события выбора пункта меню позволяет свойство `onAction` из класса `MenuItem`. Методы доступа:

```
public final void setOnAction(EventHandler<ActionEvent> value)
public final EventHandler<ActionEvent> getOnAction()
public final ObjectProperty<EventHandler<ActionEvent>> onActionProperty()
```

**Пример назначения обработчика с помощью лямбда-выражения:**

```
cutMenuItem.setOnAction(event -> {
 System.out.println("Выбран пункт Cut");
});
```

Если обработчик назначить через объект меню, то можно будет обрабатывать выбор любого пункта из этого меню. Внутри обработчика через объект события мы можем определить источник события, вызвав метод `getTarget()`. Чтобы различать пункты, лучше назначить им идентификаторы с помощью метода `setId()`:

```
cutMenuItem.setId("Cut");
copyMenuItem.setId("Copy");
pasteMenuItem.setId("Paste");
```

Можно также сохранить пользовательские данные в объекте пункта меню с помощью метода `setUserData()`. Получить эти данные позволяет метод `getUserData()`.  
**Форматы методов:**

```
public void setUserData(Object value)
public Object getUserData()
```

**Назначим обработчик для меню, который будет срабатывать при выборе любого пункта внутри меню:**

```
editMenu.setOnAction(event -> {
 if (event.getTarget() instanceof MenuItem) {
 MenuItem item = (MenuItem) event.getTarget();
 switch (item.getId()) {
 case "Cut":
 System.out.println("Выбран пункт Cut");
 break;
 case "Copy":
 System.out.println("Выбран пункт Copy");
 break;
 case "Paste":
 System.out.println("Выбран пункт Paste");
 break;
 default:
 System.out.println("Не удалось определить");
 }
 }
});
```

Очень часто пункты из меню дублируются в других местах — например, в виде кнопок на панели инструментов. Поэтому лучше внутри класса создать метод, который будет обрабатывать одновременно и событие выбора пункта в меню, и событие нажатия кнопки на панели инструментов. Пример реализации метода:

```
private void onActionAbout(ActionEvent event) {
 System.out.println("Выбран пункт About");
 System.out.println("Источник события " + event.getTarget());
}
```

**При назначении обработчика методу `setOnAction()` нужно передать ссылку на метод `onActionAbout()`:**

```
aboutMenuItem.setOnAction(this::onActionAbout);
button.setOnAction(this::onActionAbout);
```

**Назначить обработчик выбора пункта меню можно также с помощью метода `addEventHandler()`. Формат метода:**

```
public <E extends Event> void addEventHandler(
 EventType<E> eventType, EventHandler<E> eventHandler)
```

В этом случае в параметре `eventType` указывается тип `ActionEvent.ACTION`:

```
editMenu.addEventHandler(ActionEvent.ACTION, event -> {
 System.out.println("ActionEvent.ACTION");
 System.out.println("Источник события " + event.getTarget());
});
```

Для удаления обработчика предназначен метод `removeEventHandler()`. Формат метода:

```
public <E extends Event> void removeEventHandler(
 EventType<E> eventType, EventHandler<E> eventHandler)
```

Сгенерировать событие выбора пункта меню из программы позволяет метод `fire()`. Формат метода:

```
public void fire()
```

Сгенерируем событие выбора первого пункта из меню при нажатии кнопки:

```
button5.setOnAction(event -> {
 this.editMenu.getItems().get(0).fire();
});
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH15_01_MenuBar`, приведенном в каталоге `CH15_меню` сопровождающего книгу электронного архива (см. приложение).

## 15.3.3. Назначение клавиш быстрого доступа

Клавиши быстрого доступа (иногда их также называют «горячими» клавишами) позволяют выбрать пункт меню с помощью нажатия специальной клавиши (например, `<Alt>` или `<Ctrl>`) и какой-либо дополнительной клавиши.

Внутри текста пункта меню символ `_` (подчеркивание), указанный перед буквой, задает комбинацию клавиш быстрого доступа. В этом случае символ, перед которым указан символ подчеркивания, будет подчеркнут, что является подсказкой пользователю. При одновременном нажатии клавиши `<Alt>` и подчеркнутого символа меню будет отображено, а пункт выбран (сработает только в том случае, если меню, в котором находится пункт, является активным).

Пример назначения клавиши быстрого доступа (при нажатии комбинации клавиш `<Alt>+<F>` меню будет отображено):

```
Menu fileMenu = new Menu("_File");
```

Управлять разбором текста надписи пункта меню позволяет свойство `mnemonicParsing` из класса `MenuItem`. Значение `true` (используется по умолчанию) разрешает разбор текста надписи, а значение `false` — запрещает. Методы доступа:

```
public final void setMnemonicParsing(boolean value)
public final boolean isMnemonicParsing()
public final BooleanProperty mnemonicParsingProperty()
```

Получить перечень комбинаций клавиш быстрого доступа верхнего уровня позволяет метод `getMnemonics()` из класса `Scene`. Формат метода:

```
import javafx.collections.ObservableList;
import javafx.collections.ObservableMap;
public ObservableMap<KeyCombination, ObservableList<Mnemonic>>
 getMnemonics()
```

**Пример:**

```
ObservableMap<KeyCombination, ObservableList<Mnemonic>> map =
 scene.getMnemonics();
for (KeyCombination key: map.keySet()) {
 System.out.println(key.toString() + ":");
 for (Mnemonic m: map.get(key)) {
 System.out.println("---" + m.getNode().getClass());
 }
}
```

В классе `MenuItem` существует свойство `accelerator`, позволяющее управлять клавишами-ускорителями. Методы доступа:

```
public final void setAccelerator(KeyCombination value)
public final KeyCombination getAccelerator()
public final ObjectProperty<KeyCombination> acceleratorProperty()
```

Метод `setAccelerator()` позволяет назначить комбинацию клавиш для пункта меню. Пример назначения комбинации клавиш-ускорителей:

```
MenuItem openMenuItem = new MenuItem("Open...");
openMenuItem.setAccelerator(new KeyCodeCombination(KeyCode.O,
 KeyCombination.CONTROL_DOWN));
openMenuItem.setOnAction(event -> {
 System.out.println("Выбран пункт Open");
});
```

Назначенная комбинация в виде текста отобразится справа от надписи, и пользователь узнает о ее существовании, отобразив меню. При нажатии комбинации клавиш `<Ctrl>+<O>` пункт будет выбран вне зависимости от того, активно меню, в котором расположен пункт, или нет. Иными словами, клавиши-ускорители назначаются глобально, тогда как клавиши быстрого доступа будут назначены глобально только для меню верхнего уровня. Чтобы сработали клавиши быстрого доступа для пункта меню, нужно сначала отобразить меню.

Обработать событие нажатия комбинации клавиш-ускорителей позволяет свойство `onMenuValidation` из класса `MenuItem`. Методы доступа:

```
public final void setOnMenuValidation(EventHandler<Event> value)
public final EventHandler<Event> getOnMenuValidation()
public final ObjectProperty<EventHandler<Event>>
 onMenuValidationProperty()
```

Пример назначения обработчика:

```
openMenuItem.setOnMenuValidation(event -> {
 System.out.println("OnMenuValidation");
});
```

При использовании методов `addEventHandler()` и `removeEventHandler()` тип события задается с помощью статической константы `MENU_VALIDATION_EVENT`:

```
public static final EventType<Event> MENU_VALIDATION_EVENT
```

Пример:

```
openMenuItem.addEventHandler(MenuItem.MENU_VALIDATION_EVENT,
 event -> {
 System.out.println("OnMenuValidation");
 });
```

Получить перечень всех комбинаций клавиш-ускорителей позволяет метод `getAccelerators()` из класса `Scene`. Формат метода:

```
public ObservableMap<KeyCombination, Runnable> getAccelerators()
```

Пример:

```
ObservableMap<KeyCombination, Runnable> map2 = scene.getAccelerators();
for (KeyCombination key: map2.keySet()) {
 System.out.println(key.toString());
}
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH15_01_MenuBar`, приведенном в каталоге `CH15_меню` сопровождающего книгу электронного архива (см. приложение).

## 15.3.4. Класс *CheckMenuItem*: пункт меню с флажком

Класс `CheckMenuItem` реализует пункт меню с флажком. При выборе пункта меню флажок устанавливается или сбрасывается (текущее состояние сменяется противоположным). Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.CheckMenuItem;
```

Иерархия наследования:

```
Object — MenuItem — CheckMenuItem
```

Конструкторы класса:

```
CheckMenuItem()
CheckMenuItem(String text)
CheckMenuItem(String text, Node graphic)
```

Класс `CheckMenuItem` наследует все свойства и методы из класса `MenuItem` (см. разд. 15.3.1) и добавляет свойство `selected`, которое содержит текущее состоя-

ние флажка: `true` — если флажок установлен, и `false` — если сброшен. Методы доступа:

```
public final void setSelected(boolean value)
public final boolean isSelected()
public final BooleanProperty selectedProperty()
```

Пример создания пункта меню с флажком и назначения обработчиков:

```
Menu paneMenu = new Menu("_Панели");
CheckMenuItem paneMenuItem = new CheckMenuItem("Панель _меню");
paneMenuItem.setSelected(false);
paneMenuItem.setOnAction(event -> {
 System.out.print("Выбран пункт Панель меню ");
 System.out.println(paneMenuItem.isSelected());
});
paneMenuItem.selectedProperty().addListener(
 (obj, oldValue, newValue) -> {
 System.out.println("Панель меню " + newValue);
 });
paneMenu.getItems().add(paneMenuItem);
```

Пример двунаправленного связывания свойств (установка флажка в меню приведет к установке связанного флажка и наоборот):

```
CheckBox checkBox = new CheckBox("Панель меню");
checkBox.selectedProperty().bindBidirectional(
 paneMenuItem.selectedProperty());
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH15_01_MenuBar`, приведенном в каталоге `CH15_menu` сопровождающего книгу электронного архива (см. *приложение*).

## 15.3.5. Класс *RadioMenuItem*: пункт меню с переключателем

Класс `RadioMenuItem` реализует пункт меню с переключателем. Обычно переключатели организуются в группы. Внутри группы может быть включен только один переключатель. При попытке включить другой переключатель, ранее включенный переключатель автоматически отключается. Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.RadioMenuItem;
```

Иерархия наследования:

`Object` — `MenuItem` — `RadioMenuItem`

Конструкторы класса:

```
RadioMenuItem()
RadioMenuItem(String text)
RadioMenuItem(String text, Node graphic)
```

Класс `RadioMenuItem` наследует все свойства и методы из класса `MenuItem` (см. *разд. 15.3.1*) и добавляет следующие свойства:

- ❑ `selected` — содержит значение `true`, если переключатель включен, и `false` — если выключен. Методы доступа:

```
public final void setSelected(boolean value)
public final boolean isSelected()
public final BooleanProperty selectedProperty()
```

- ❑ `toggleGroup` — содержит ссылку на группу или значение `null`. Группа переключателей реализуется с помощью класса `ToggleGroup` (см. *разд. 11.8.3*). Методы доступа:

```
public final void setToggleGroup(ToggleGroup value)
public final ToggleGroup getToggleGroup()
public final ObjectProperty<ToggleGroup> toggleGroupProperty()
```

Пример создания двух пунктов с переключателями, добавления их в группу и назначения обработчиков события выбора пункта меню:

```
RadioMenuItem leftPaneMenuItem = new RadioMenuItem("Слева");
leftPaneMenuItem.setSelected(true);
leftPaneMenuItem.setUserData("leftPane");
leftPaneMenuItem.setOnAction(event -> {
 System.out.println("Выбран пункт Слева");
});
```

```
RadioMenuItem rightPaneMenuItem = new RadioMenuItem("Справа");
rightPaneMenuItem.setUserData("rightPane");
rightPaneMenuItem.setOnAction(event -> {
 System.out.println("Выбран пункт Справа");
});
```

```
ToggleGroup group = new ToggleGroup();
leftPaneMenuItem.setToggleGroup(group);
rightPaneMenuItem.setToggleGroup(group);
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH15_01_MenuBar`, приведенном в каталоге `CH15_меню` сопровождающего книгу электронного архива (см. *приложение*).

## 15.3.6. Класс `SeparatorMenuItem`: разделитель пунктов меню

Класс `SeparatorMenuItem` реализует горизонтальную линию, с помощью которой можно визуально разделить пункты меню на группы. Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.SeparatorMenuItem;
```

Иерархия наследования:

```
Object — MenuItem — CustomMenuItem — SeparatorMenuItem
```

Конструктор класса:

```
SeparatorMenuItem()
```

Использовать класс `SeparatorMenuItem` очень просто — достаточно создать экземпляр класса и добавить его в список меню как обычный пункт:

```
paneMenu.getItems().add(new SeparatorMenuItem());
```

### 15.3.7. Класс *CustomMenuItem*: произвольный пункт меню

Класс `CustomMenuItem` позволяет создать пункт меню с произвольным узлом. Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.CustomMenuItem;
```

Иерархия наследования:

```
Object — MenuItem — CustomMenuItem
```

Конструкторы класса:

```
CustomMenuItem()
CustomMenuItem(Node content)
CustomMenuItem(Node content, boolean hideOnClick)
```

Класс `CustomMenuItem` наследует все свойства и методы из класса `MenuItem` (см. *разд. 15.3.1*) и добавляет следующие свойства:

□ `content` — задает узел, отображаемый в пункте меню. Методы доступа:

```
public final void setContent(Node value)
public final Node getContent()
public final ObjectProperty<Node> contentProperty()
```

□ `hideOnClick` — если свойство имеет значение `false`, то выбор пункта не будет приводить к автоматическому сокрытию меню, что позволит взаимодействовать с узлом. Значение по умолчанию: `true`. Методы доступа:

```
public final void setHideOnClick(boolean value)
public final boolean isHideOnClick()
public final BooleanProperty hideOnClickProperty()
```

Добавим шкалу с ползунком в меню:

```
Slider slider = new Slider(0.0, 100.0, 50.0);
slider.valueProperty().addListener((obj, oldValue, newValue) -> {
 System.out.println("slider value = " + newValue);
});
```



```
CustomMenuItem customMenuItem = new CustomMenuItem();
customMenuItem.setContent(slider);
customMenuItem.setHideOnClick(false);

paneMenu.getItems().add(customMenuItem);
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH15_01_MenuBar`, приведенном в каталоге `CH15_menu` сопровождающего книгу электронного архива (см. приложение).

## 15.4. Класс *MenuBar*: кнопка вызова меню

Класс `MenuBar` реализует кнопку, при нажатии на которую отображается всплывающее меню (рис. 15.2). Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.MenuButton;
```

Иерархия наследования:

`Object` — `Node` — `Parent` — `Region` — `Control` — `Labeled` — `ButtonBase` — `MenuBar`

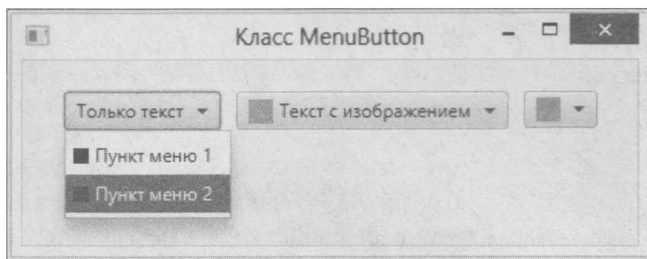


Рис. 15.2. Класс `MenuBar`: кнопка вызова меню

Конструкторы класса:

```
MenuBar()
MenuBar(String text)
MenuBar(String text, Node graphic)
MenuBar(String text, Node graphic, MenuItem... items)
```

В параметре `text` указывается текст надписи, в параметре `graphic` — узел, который будет отображаться слева от надписи (например, изображение), а в параметре `items` — пункты меню через запятую. Добавить пункты меню после создания объекта позволяет метод `getItems()`. Формат метода:

```
public final ObservableList<MenuItem> getItems()
```

Пример:

```
MenuItem redMenuItem = new MenuItem("Красный");
redMenuItem.setOnAction(event -> {
 System.out.println("Выбран пункт Красный");
});
```

```
MenuItem blueMenuItem = new MenuItem("Синий");
blueMenuItem.setOnAction(event -> {
 System.out.println("Выбран пункт Синий");
});

MenuButton menuButton = new MenuButton("Цвет");
menuButton.getItems().addAll(redMenuItem, blueMenuItem);
```

**Класс MenuButton наследует все свойства и методы из базовых классов и добавляет следующие свойства:**

- ❑ **popupSide** — задает местоположение всплывающего меню относительно кнопки. В качестве значения указываются константы: TOP (сверху), BOTTOM (снизу, значение по умолчанию), LEFT (слева) или RIGHT (справа) из перечисления Side. Если места с указанной стороны недостаточно, то значение свойства может быть проигнорировано. Методы доступа:

```
import javafx.geometry.Side;
public final void setPopupSide(Side value)
public final Side getPopupSide()
public final ObjectProperty<Side> popupSideProperty()
```

**Пример:**

```
menuButton.setPopupSide(Side.LEFT);
```

- ❑ **showing** — содержит значение true, если всплывающее меню отображается на экране, и false — если скрыто. Методы доступа:

```
public final boolean isShowing()
public final ReadOnlyBooleanProperty showingProperty()
```

**Пример назначения обработчика:**

```
menuButton.showingProperty().addListener(
 (obj, oldValue, newValue) -> {
 System.out.println("showing " + newValue);
 });
```

- ❑ **onShowing** — задает обработчик события, возникающего перед отображением всплывающего меню. Свойство доступно, начиная с JavaFX 10. Методы доступа:

```
public final void setOnShowing(EventHandler<Event> value)
public final EventHandler<Event> getOnShowing()
public final ObjectProperty<EventHandler<Event>>
 onShowingProperty()
```

**Пример назначения обработчика:**

```
menuButton.setOnShowing(event -> {
 System.out.println("OnShowing");
});
```

- ❑ `onShown` — задает обработчик события отображения всплывающего меню. Свойство доступно, начиная с JavaFX 10. Методы доступа:

```
public final void setOnShown(EventHandler<Event> value)
public final EventHandler<Event> getOnShown()
public final ObjectProperty<EventHandler<Event>>
 onShownProperty()
```

- ❑ `onHiding` — задает обработчик события, возникающего перед сокрытием всплывающего меню. Свойство доступно, начиная с JavaFX 10. Методы доступа:

```
public final void setOnHiding(EventHandler<Event> value)
public final EventHandler<Event> getOnHiding()
public final ObjectProperty<EventHandler<Event>>
 onHidingProperty()
```

- ❑ `onHidden` — задает обработчик события сокрытия всплывающего меню. Свойство доступно, начиная с JavaFX 10. Методы доступа:

```
public final void setOnHidden(EventHandler<Event> value)
public final EventHandler<Event> getOnHidden()
public final ObjectProperty<EventHandler<Event>>
 onHiddenProperty()
```

Назначить обработчик можно также с помощью метода `addEventHandler()`. В этом случае в параметре `eventType` указываются следующие константы (доступны, начиная с версии 8.60):

```
public static final EventType<Event> ON_SHOWING
public static final EventType<Event> ON_SHOWN
public static final EventType<Event> ON_HIDING
public static final EventType<Event> ON_HIDDEN
```

**Пример назначения обработчика для события отображения меню:**

```
menuButton.addEventHandler(MenuButton.ON_SHOWN, event -> {
 System.out.println("ON_SHOWN");
});
```

Отобразить всплывающее меню из программы позволяет метод `show()`, а для сокрытия меню можно воспользоваться методом `hide()`. Меню автоматически скрывается при выборе пункта или при потере фокуса ввода. Форматы методов:

```
public void show()
public void hide()
```

При нажатии клавиши `<S>` отобразим меню, а при нажатии клавиши `<H>` — скроем:

```
window.addEventHandler(KeyEvent.KEY_PRESSED, event -> {
 if (event.getCode() == KeyCode.S) {
 menuButton.show();
 }
})
```

```
else if (event.getCode() == KeyCode.H) {
 menuButton.hide();
}
});
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH15_04_MenuButton`, приведенном в каталоге *CH15\_меню* сопровождающего книгу электронного архива (см. приложение).

## 15.5. Класс *SplitMenuButton*: кнопка с меню

Класс `SplitMenuButton` позволяет создать комбинацию обычной кнопки и кнопки вызова всплывающего меню (рис. 15.3). Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.SplitMenuButton;
```

Иерархия наследования:

```
Object – Node – Parent – Region – Control – Labeled – ButtonBase –
 MenuButton – SplitMenuButton
```

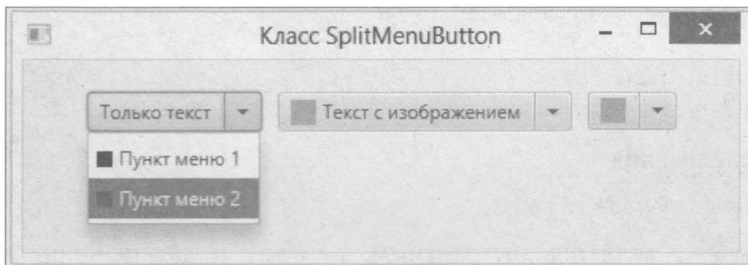


Рис. 15.3. Класс `SplitMenuButton`: кнопка с меню

Конструкторы класса:

```
SplitMenuButton()
SplitMenuButton(MenuItem... items)
```

Класс `SplitMenuButton` наследует все свойства и методы из класса `MenuButton` (см. разд. 15.4), а также из других базовых классов.

Пример создания объекта и назначения обработчиков выбора пунктов меню:

```
MenuItem redMenuItem = new MenuItem("Красный");
redMenuItem.setOnAction(event -> {
 System.out.println("Выбран пункт Красный");
});
MenuItem blueMenuItem = new MenuItem("Синий");
blueMenuItem.setOnAction(event -> {
 System.out.println("Выбран пункт Синий");
});
```

```
SplitMenuButton menuButton = new SplitMenuButton();
menuButton.setText("Текст на кнопке");
menuButton.getItems().addAll(redMenuItem, blueMenuItem);
```

Обработать нажатие кнопки можно так же, как и при использовании обычной кнопки:

```
menuButton.setOnAction(event -> {
 System.out.println("Нажата кнопка");
});
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH15_04_MenuButton`, приведенном в каталоге *CH15\_menu* сопровождающего книгу электронного архива (см. приложение).

## 15.6. Класс *PopupWindow*: всплывающее окно

Абстрактный класс `PopupWindow` описывает окно без заголовка и рамки, которое используется для отображения различных всплывающих окон, — например, для вывода контекстного меню или окна с текстом подсказки. Такое окно не отображается на панели задач в Windows и всегда привязано к родительскому окну. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.stage.PopupWindow;
```

Иерархия наследования:

`Object` — `Window` — `PopupWindow`

Класс `PopupWindow` является абстрактным, поэтому в дальнейших примерах мы будем пользоваться классом `Tooltip` (см. *разд. 15.8*), реализующим всплывающее окно с текстом подсказки пользователю. Пример создания объекта:

```
Tooltip tooltip = new Tooltip("Текст подсказки");
```

### 15.6.1. Отображение и сокрытие всплывающего окна

Показать всплывающее окно на экране позволяет метод `show()`. Форматы метода:

```
public void show(Window owner)
public void show(Window ownerWindow, double anchorX, double anchorY)
public void show(Node ownerNode, double anchorX, double anchorY)
```

В первом формате указывается ссылка на родительское окно (значение `null` не допустимо). Окно выводится по центру экрана. Пример:

```
tooltip.show(window);
```

Второй формат позволяет дополнительно указать координаты местоположения всплывающего окна относительно левого верхнего угла экрана:

```
tooltip.show(window, window.getX() + 100.0, window.getY() + 100.0);
```

Третий формат в первом параметре принимает ссылку на узел, родительское окно которого будет использоваться в качестве родителя, а в последующих параметрах — координаты местоположения всплывающего окна относительно левого верхнего угла экрана:

```
tooltip.show(button, window.getX() + 100.0, window.getY() + 100.0);
```

Получить ссылку на родителя всплывающего окна позволяют следующие свойства из класса `PopupWindow`:

- ❑ `ownerWindow` — содержит ссылку на родительское окно. Методы доступа:

```
public final Window getOwnerWindow()
public final ReadOnlyObjectProperty<Window> ownerWindowProperty()
```

- ❑ `ownerNode` — содержит ссылку на родительский узел или значение `null`. Методы доступа:

```
public final Node getOwnerNode()
public final ReadOnlyObjectProperty<Node> ownerNodeProperty()
```

- ❑ `scene` — содержит ссылку на объект сцены. Методы доступа:

```
public final Scene getScene()
public final ReadOnlyObjectProperty<Scene> sceneProperty()
```

Чтобы скрыть всплывающее окно, следует воспользоваться методом `hide()`. Формат метода:

```
public void hide()
```

Управлять сокрытием всплывающего окна позволяют следующие свойства из класса `PopupWindow`:

- ❑ `autoHide` — если свойство имеет значение `true`, то при потере фокуса всплывающее окно будет автоматически скрыто. Если при отображении окна указывался родительский узел, то окно не будет закрыто при получении фокуса ввода этим узлом. Методы доступа:

```
public final void setAutoHide(boolean value)
public final boolean isAutoHide()
public final BooleanProperty autoHideProperty()
```

- ❑ `onAutoHide` — задает обработчик автоматического сокрытия всплывающего окна. Значение по умолчанию: `null`. Методы доступа:

```
public final void setOnAutoHide(EventHandler<Event> value)
public final EventHandler<Event> getOnAutoHide()
public final ObjectProperty<EventHandler<Event>>
 onAutoHideProperty()
```

- ❑ `hideOnEscape` — если свойство имеет значение `true`, то нажатие клавиши `<Esc>` приведет к сокрытию всплывающего окна. Методы доступа:

```
public final void setHideOnEscape(boolean value)
public final boolean isHideOnEscape()
public final BooleanProperty hideOnEscapeProperty()
```

- ❑ `consumeAutoHidingEvents` — если свойство имеет значение `true`, то будет установлен флаг прерывания распространения события, которое вызвало автоматическое сокрытие всплывающего окна. Например, если мы назначим обработчик нажатия кнопки для окна верхнего уровня, то при сокрытии всплывающего окна с помощью клавиши `<Esc>` обработчик вызван не будет. Методы доступа:

```
public final void setConsumeAutoHidingEvents(boolean value)
public final boolean getConsumeAutoHidingEvents()
public final BooleanProperty consumeAutoHidingEventsProperty()
```

Обработать события отображения и сокрытия всплывающего окна можно с помощью свойств: `onShowing` (событие `WINDOW_SHOWING`), `onShown` (событие `WINDOW_SHOWN`), `onHiding` (событие `WINDOW_HIDING`), `onHidden` (событие `WINDOW_HIDDEN`), `showing` (изменение видимости) и `focused` (изменение фокуса ввода) из класса `Window`. Мы уже рассматривали эти свойства в *разд. 4.7.1*, применительно к окну верхнего уровня. Пример:

```
tooltip.setOnShown(event -> {
 System.out.println("setOnShown - " + event.getEventType());
});
tooltip.addEventHandler(WindowEvent.WINDOW_HIDDEN, event -> {
 System.out.println("WINDOW_HIDDEN - " + event.getEventType());
});
tooltip.showingProperty().addListener((obj, oldValue, newValue) -> {
 System.out.println("Состояние окна изменилось: " + newValue);
});
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH15_06_PopupWindow`, приведенном в каталоге *CH15\_теги* сопровождающего книгу электронного архива (см. *приложение*).

## 15.6.2. Изменение и получение размеров всплывающего окна

Управлять размерами всплывающего окна позволяют следующие свойства из класса `Window`:

- ❑ `width` — ширина окна. Методы доступа:

```
public final void setWidth(double value)
public final double getWidth()
public final ReadOnlyDoubleProperty widthProperty()
```

- ❑ `height` — высота окна. Методы доступа:

```
public final void setHeight(double value)
public final double getHeight()
public final ReadOnlyDoubleProperty heightProperty()
```

### 15.6.3. Местоположение всплывающего окна на экране

Местоположение якорной точки всплывающего окна указывается при вызове метода `show()` относительно левого верхнего угла экрана. Управлять местоположением окна позволяют следующие свойства:

- ❑ `x` — положение левого верхнего угла окна на экране по оси `x`. Методы доступа:

```
public final void setX(double value)
public final double getX()
public final ReadOnlyDoubleProperty xProperty()
```

- ❑ `y` — положение левого верхнего угла окна на экране по оси `y`. Методы доступа:

```
public final void setY(double value)
public final double getY()
public final ReadOnlyDoubleProperty yProperty()
```

- ❑ `anchorX` — положение якорной точки на экране по оси `x`. Методы доступа:

```
public final void setAnchorX(double value)
public final double getAnchorX()
public final ReadOnlyDoubleProperty anchorXProperty()
```

- ❑ `anchorY` — положение якорной точки на экране по оси `y`. Методы доступа:

```
public final void setAnchorY(double value)
public final double getAnchorY()
public final ReadOnlyDoubleProperty anchorYProperty()
```

- ❑ `anchorLocation` — задает положение якорной точки внутри окна. В параметре `value` можно указать константы: `WINDOW_TOP_LEFT`, `WINDOW_TOP_RIGHT`, `WINDOW_BOTTOM_LEFT`, `WINDOW_BOTTOM_RIGHT`, `CONTENT_TOP_LEFT`, `CONTENT_TOP_RIGHT`, `CONTENT_BOTTOM_LEFT` или `CONTENT_BOTTOM_RIGHT` из перечисления `AnchorLocation`. Методы доступа:

```
public final void setAnchorLocation(
 PopupWindow.AnchorLocation value)
public final PopupWindow.AnchorLocation getAnchorLocation()
public final ObjectProperty<PopupWindow.AnchorLocation>
 anchorLocationProperty()
```

Пример:

```
tooltip.setAnchorLocation(
 PopupWindow.AnchorLocation.WINDOW_TOP_LEFT);
```

- ❑ `autoFix` — если свойство имеет значение `true`, то положение всплывающего окна будет корректироваться таким образом, чтобы оно не выходило за пределы экрана. Методы доступа:

```
public final void setAutoFix(boolean value)
public final boolean isAutoFix()
public final BooleanProperty autoFixProperty()
```



### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH15_06_PopupWindow`, приведенном в каталоге *CH15\_теги* сопровождающего книгу электронного архива (см. приложение).

## 15.6.4. Класс *Popup*

Класс `Popup` является реализацией абстрактного класса `PopupWindow`. Инструкция импорта:

```
// Модуль javafx.controls
import javafx.stage.Popup;
```

Иерархия наследования:

`Object` — `Window` — `PopupWindow` — `Popup`

Конструктор класса:

```
Popup()
```

Класс `Popup` наследует все свойства и методы из класса `PopupWindow` и дополнительно содержит метод `getContent()`, с помощью которого можно добавить узлы в окно.

Формат метода:

```
public final ObservableList<Node> getContent()
```

Пример создания всплывающего окна с синим фоном и надписью белого цвета:

```
VBox vbox = new VBox();
vbox.setAlignment(Pos.CENTER);
vbox.setPrefSize(200.0, 100.0);
vbox.setStyle("-fx-background-color: blue;");
```

```
Label txt = new Label("Текст внутри окна");
txt.setTextFill(Color.WHITE);
vbox.getChildren().add(txt);
```

```
Popup popup = new Popup();
popup.setAutoHide(true);
popup.setAutoFix(true);
popup.setHideOnEscape(true);
popup.setConsumeAutoHidingEvents(true);
popup.getContent().add(vbox);
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH15_06_PopupWindow`, приведенном в каталоге *CH15\_теги* сопровождающего книгу электронного архива (см. приложение).

## 15.7. Класс *PopupControl*: всплывающее окно

Класс `PopupControl` расширяет класс `PopupWindow` (см. *разд. 15.6*) и добавляет возможность использования стилей. Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.PopupControl;
```

Иерархия наследования:

Object — Window — `PopupWindow` — `PopupControl`

Конструктор класса:

```
PopupControl()
```

Пример создания всплывающего окна с надписью белого цвета на зеленом фоне:

```
PopupControl popupControl = new PopupControl();
popupControl.setAutoHide(true);
popupControl.setAutoFix(true);
popupControl.setHideOnEscape(true);
```

```
VBox box = new VBox();
box.setAlignment(Pos.CENTER);
box.setPrefSize(200.0, 100.0);
box.setStyle("-fx-background-color: green;");
```

```
Label text = new Label("Текст внутри окна");
text.setTextFill(Color.WHITE);
box.getChildren().add(text);
```

```
popupControl.getScene().setRoot(box);
```

Класс `PopupControl` содержит следующие основные свойства:

- ❑ `prefWidth` и `prefHeight` — задают предпочтительную ширину и высоту соответственно. Методы доступа:

```
public final void setPrefWidth(double value)
public final double getPrefWidth()
public final DoubleProperty prefWidthProperty()
public final void setPrefHeight(double value)
public final double getPrefHeight()
public final DoubleProperty prefHeightProperty()
```

Можно также воспользоваться методом `setPrefSize()`:

```
public void setPrefSize(double prefWidth, double prefHeight)
```

- ❑ `minWidth` и `minHeight` — задают минимальную ширину и высоту соответственно. Методы доступа:

```
public final void setMinWidth(double value)
public final double getMinWidth()
```

```
public final DoubleProperty minWidthProperty()
public final void setMinHeight(double value)
public final double getMinHeight()
public final DoubleProperty minHeightProperty()
```

Можно также воспользоваться методом `setMinSize()`:

```
public void setMinSize(double minWidth, double minHeight)
```

- ❑ `maxWidth` и `maxHeight` — задают максимальную ширину и высоту соответственно. Методы доступа:

```
public final void setMaxWidth(double value)
public final double getMaxWidth()
public final DoubleProperty maxWidthProperty()
public final void setMaxHeight(double value)
public final double getMaxHeight()
public final DoubleProperty maxHeightProperty()
```

Можно также воспользоваться методом `setMaxSize()`:

```
public void setMaxSize(double maxWidth, double maxHeight)
```

- ❑ `id` — задает идентификатор окна. Методы доступа:

```
public final void setId(String value)
public final String getId()
public final StringProperty idProperty()
```

- ❑ `style` — задает стиль. Методы доступа:

```
public final void setStyle(String value)
public final String getStyle()
public final StringProperty styleProperty()
```

Сделаем фон окна со всплывающей подсказкой зеленым:

```
tooltip.setStyle("-fx-background-color: green;");
```

При указании размеров в качестве значения можно передать вещественное число или одну из статических констант из класса `PopupControl`:

- ❑ `USE_COMPUTED_SIZE` — указывает, что значение не задано. Формат:

```
public static final double USE_COMPUTED_SIZE
```

Пример вывода значения константы:

```
System.out.println(PopupControl.USE_COMPUTED_SIZE); // -1.0
```

- ❑ `USE_PREF_SIZE` — указывает, что должны использоваться предпочтительные размеры. Применяется только при указании минимального или максимального размера. Формат:

```
public static final double USE_PREF_SIZE
```

Пример вывода значения константы:

```
System.out.println(PopupControl.USE_PREF_SIZE); // -Infinity
```

**ЭЛЕКТРОННЫЙ АРХИВ**

Полный текст программы содержится в проекте `FX_CH15_06_PopupWindow`, приведенном в каталоге *CH15\_теги* сопровождающего книгу электронного архива (см. приложение).

## 15.8. Класс *Tooltip*: всплывающие подсказки

При работе с программой у пользователя могут возникать вопросы, для чего предназначен тот или иной компонент. Обычно для информирования пользователя используются надписи, расположенные рядом с компонентом. Но часто место в окне либо ограничено, либо вывод таких надписей испортит весь дизайн окна. В таких случаях принято выводить текст подсказки в отдельном окне без рамки (рис. 15.4) при наведении указателя мыши на компонент.

Реализовать такую подсказку внутри всплывающего окна позволяет класс `Tooltip`. При наведении указателя мыши на компонент через некоторый промежуток времени окно отобразится рядом с указателем мыши. При выведении указателя мыши из области компонента или при потере фокуса ввода всплывающее окно автоматически скрывается. Скрыть окно можно также путем нажатия клавиши `<Esc>`. Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.Tooltip;
```

Иерархия наследования:

`Object` — `Window` — `PopupWindow` — `PopupControl` — `Tooltip`

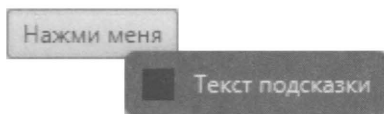


Рис. 15.4. Всплывающее окно с подсказкой и значком рядом с кнопкой

Конструкторы класса:

```
Tooltip()
Tooltip(String text)
```

Прикрепить окно со всплывающей подсказкой к компоненту позволяет свойство `tooltip` из класса `Control`. Методы доступа:

```
public final void setTooltip(Tooltip value)
public final Tooltip getTooltip()
public final ObjectProperty<Tooltip> tooltipProperty()
```

Пример:

```
Label text = new Label("Текст надписи");
text.setTooltip(new Tooltip("Текст подсказки"));
```

Для прикрепления подсказки к узлу можно воспользоваться статическим методом `install()` из класса `Tooltip`, а для удаления привязки — методом `uninstall()`. Форматы методов:

```
public static void install(Node node, Tooltip t)
public static void uninstall(Node node, Tooltip t)
```

**Пример:**

```
Button btn = new Button("Удалить подсказку");
Tooltip t = new Tooltip("Текст подсказки");
Tooltip.install(btn, t);
btn.setOnAction(event -> {
 Tooltip.uninstall(btn, t);
 btn.setDisable(true);
});
```

**Класс `Tooltip` содержит свойства:** `text`, `graphic`, `font`, `wrapText`, `graphicTextGap`, `textOverrun`, `contentDisplay` и `textAlignment`, которые аналогичны одноименным свойствам из класса `Labeled`. Класс `Labeled` мы уже рассматривали в *разд. 11.3*, поэтому не будем повторяться. Пример использования свойств:

```
Tooltip tooltip = new Tooltip();
tooltip.setText("Текст подсказки");
tooltip.setGraphic(new ImageView(createImage(16, 16, Color.BLUE)));
tooltip.setFont(Font.font(12.0));
tooltip.setWrapText(false);
tooltip.setGraphicTextGap(10.0);
tooltip.setTextOverrun(OverrunStyle.ELLIPSIS);
tooltip.setContentDisplay(ContentDisplay.LEFT);
tooltip.setTextAlignment(TextAlignment.LEFT);
tooltip.setStyle("-fx-background-color: green;");
```

```
Button button = new Button("Нажми меня");
button.setTooltip(tooltip);
```

**Класс `Tooltip` наследует все свойства и методы из базовых классов (см. *разд. 15.6* и *15.7*) и добавляет следующие свойства:**

- `activated` — содержит значение `true` при вхождении указателя мыши в область компонента. При отображении подсказки свойству присваивается значение `false`. Промежуток времени между этими двумя событиями задается с помощью свойства `showDelay`. Если пользователь навел указатель мыши и сразу вывел его из области компонента, то подсказка не отобразится, и свойство будет иметь значение `true`. Методы доступа:

```
public final boolean isActivated()
public final ReadOnlyBooleanProperty activatedProperty()
```

**Пример назначения обработчика:**

```
tooltip.activatedProperty().addListener(
 (obj, oldValue, newValue) -> {
 System.out.println("activated: " + newValue);
 });
```

- **showDelay** — задает задержку перед отображением подсказки после вхождения указателя мыши в область компонента. Значение по умолчанию: 1000 миллисекунд. Свойство доступно, начиная с JavaFX 9. Методы доступа:

```
import javafx.util.Duration;
public final void setShowDelay(Duration showDelay)
public final Duration getShowDelay()
public final ObjectProperty<Duration> showDelayProperty()
```

**Пример:**

```
tooltip.setShowDelay(Duration.millis(1000.0));
```

- **showDuration** — задает максимальную продолжительность показа окна с подсказкой, пока указатель мыши находится в области компонента. Если указатель мыши покинет область компонента, то окно может закрыться раньше с задержкой **hideDelay**. Значение по умолчанию: 5000 миллисекунд. Свойство доступно, начиная с JavaFX 9. Методы доступа:

```
public final void setShowDuration(Duration showDuration)
public final Duration getShowDuration()
public final ObjectProperty<Duration> showDurationProperty()
```

**Пример:**

```
tooltip.setShowDuration(Duration.millis(5000.0));
```

- **hideDelay** — задает задержку между выведением указателя мыши из области компонента и сокрытием окна с подсказкой. Значение по умолчанию: 200 миллисекунд. Свойство доступно, начиная с JavaFX 9. Методы доступа:

```
public final void setHideDelay(Duration hideDelay)
public final Duration getHideDelay()
public final ObjectProperty<Duration> hideDelayProperty()
```

**Пример:**

```
tooltip.setHideDelay(Duration.millis(500.0));
```

**ЭЛЕКТРОННЫЙ АРХИВ**

Полный текст программы содержится в проекте `FX_CH15_06_PopupWindow`, приведенном в каталоге `CH15_теги` сопровождающего книгу электронного архива (см. приложение).

## 15.9. Класс *ContextMenu*: контекстное меню

Класс `ContextMenu` реализует контекстное меню, отображаемое при щелчке правой кнопкой мыши в области компонента. Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.ContextMenu;
```

Иерархия наследования:

Object — Window — `PopupWindow` — `PopupControl` — `ContextMenu`

Конструкторы класса:

```
ContextMenu()
ContextMenu(MenuItem... items)
```

Для прикрепления контекстного меню к компоненту используется свойство `contextMenu` из класса `Control`. Методы доступа:

```
public final void setContextMenu(ContextMenu value)
public final ContextMenu getContextMenu()
public final ObjectProperty<ContextMenu> contextMenuProperty()
```

Пример:

```
Label label = new Label("Надпись");
label.setContextMenu(new ContextMenu(new MenuItem("Пункт 1"),
 new MenuItem("Пункт 2")));
```

Для добавления пунктов меню после создания объекта контекстного меню следует воспользоваться методом `getItems()`. Формат метода:

```
public final ObservableList<MenuItem> getItems()
```

Пример создания контекстного меню и назначения обработчиков:

```
MenuItem menuItem = new MenuItem("Пункт меню 1");
MenuItem menuItem2 = new MenuItem("Пункт меню 2");
menuItem.setOnAction(event -> {
 label.setText("Выбран пункт меню 1");
});
menuItem2.setOnAction(event -> {
 label.setText("Выбран пункт меню 2");
});
ContextMenu menu = new ContextMenu();
menu.getItems().addAll(menuItem, menuItem2);
```

```
Label label = new Label("Надпись");
label.setContextMenu(menu);
```

Класс `ContextMenu` наследует все свойства и методы из базовых классов (см. *разд. 15.6* и *15.7*) и добавляет свойство `onAction`, которое позволяет назначить обработчик выбора пункта из контекстного меню. Методы доступа:

```
public final void setOnAction(EventHandler<ActionEvent> value)
public final EventHandler<ActionEvent> getOnAction()
public final ObjectProperty<EventHandler<ActionEvent>> onActionProperty()
```

Получить ссылку на выбранный пункт можно с помощью метода `getTarget()`:

```
menu.setOnAction(event -> {
 System.out.println("setOnAction() " + event.getTarget());
});
```

Класс `ContextMenu` содержит также методы `show()` и `hide()`, которые позволяют отобразить и скрыть контекстное меню соответственно. Форматы методов:

```
public void show(Node anchor, double screenX, double screenY)
public void show(Node anchor, Side side, double dx, double dy)
public void hide()
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH15_06_PopupWindow`, приведенном в каталоге `CH15_меню` сопровождающего книгу электронного архива (см. приложение).

## 15.10. Класс *ContextMenuEvent*: событие вызова контекстного меню

Когда пользователь запрашивает контекстное меню с помощью щелчка правой кнопкой мыши или нажатия комбинации клавиш, генерируется событие `CONTEXT_MENU_REQUESTED` из класса `ContextMenuEvent`. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.scene.input.ContextMenuEvent;
```

Иерархия наследования:

```
Object — EventObject — Event — InputEvent — ContextMenuEvent
```

Пример назначения обработчика:

```
label.addEventHandler(ContextMenuEvent.CONTEXT_MENU_REQUESTED, event -> {
 System.out.println("ContextMenuEvent");
});
```

Внутри обработчика через объект события доступны следующие основные методы:

- `getX()`, `getY()` и `getZ()` — возвращают координаты указателя относительно узла. Форматы методов:

```
public final double getX()
public final double getY()
public final double getZ()
```

- `getSceneX()` и `getSceneY()` — возвращают координаты указателя относительно сцены. Форматы методов:



```
public final double getSceneX()
public final double getSceneY()
```

- ❑ `getScreenX()` и `getScreenY()` — возвращают координаты указателя относительно левого верхнего угла экрана. Форматы методов:

```
public final double getScreenX()
public final double getScreenY()
```

- ❑ `isKeyboardTrigger()` — возвращает значение `true`, если нажата комбинация клавиш на клавиатуре, и значение `false` — если выполнен щелчок правой кнопкой мыши. Формат метода:

```
public boolean isKeyboardTrigger()
```

- ❑ `getPickResult()` — возвращает объект класса `PickResult`. Формат метода:

```
import javafx.scene.input.PickResult;
public final PickResult getPickResult()
```

### **ЭЛЕКТРОННЫЙ АРХИВ**

Полный текст программы содержится в проекте `FX_CH15_06_PopupWindow`, приведенном в каталоге *CH15\_теги* сопровождающего книгу электронного архива (см. приложение).

# ГЛАВА 16



## Диалоговые окна

Диалоговые окна предназначены для информирования пользователя, а также для получения данных от пользователя. В большинстве случаев диалоговые окна являются модальными (т. е. блокирующими все окна приложения или только родительское окно) и отображаются на непродолжительный промежуток времени.

Начиная с версии 8.40, в состав библиотеки JavaFX входит класс `Dialog<R>`, с помощью которого реализуются различные диалоговые окна. Класс `Dialog<R>` наследуют другие классы, которые реализуют готовые диалоговые окна. Например, класс `Alert` предоставляет готовые диалоговые окна для вывода сообщений, класс `TextInputDialog` — для ввода данных в текстовое поле, а класс `ChoiceDialog<T>` позволяет выбрать значение из раскрывающегося списка.

Кроме того, доступны классы `DirectoryChooser` и `FileChooser`, реализующие окна для выбора папки и файла соответственно. Эти классы не наследуют класс `Dialog<R>`.

### 16.1. Класс *Dialog<R>*

Класс `Dialog<R>` позволяет создать пользовательское диалоговое окно. По умолчанию окно выводится с рамкой и заголовком, в котором расположены меню и кнопка **Заккрыть**. Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.Dialog;
```

Инструкция импорта:

```
Object — Dialog<R>
```

#### 16.1.1. Создание и отображение диалогового окна

Конструктор класса `Dialog<R>`:

```
Dialog()
```

Создадим модальное диалоговое окно (рис. 16.1) с сообщением и кнопкой **ОК** внутри обработчика нажатия кнопки (листинг 16.1).

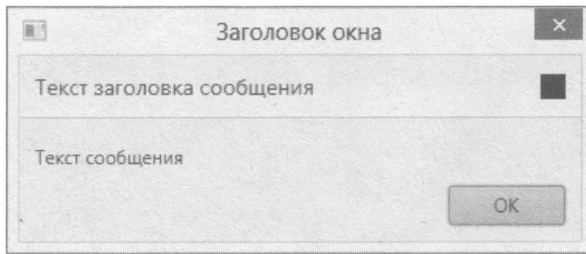


Рис. 16.1. Результат выполнения программы из листинга 16.1

#### Листинг 16.1. Диалоговое окно с сообщением и кнопкой ОК

```
Button button = new Button("Открыть окно");
button.setOnAction(event -> {
 Dialog<ButtonType> dialog = new Dialog<ButtonType>();
 dialog.setTitle("Заголовок окна");
 dialog.setHeaderText("Текст заголовка сообщения");
 dialog.setGraphic(new ImageView(createImage(16, 16, Color.BLUE)));
 dialog.setContentText("Текст сообщения");
 dialog.getDialogPane().getButtonTypes().add(ButtonType.OK);
 dialog.show();
});
```

Рассмотрим основные свойства из класса `Dialog<R>`:

- ❑ **title** — задает текст, выводимый в заголовке диалогового окна. Методы доступа:
 

```
public final void setTitle(String title)
public final String getTitle()
public final StringProperty titleProperty()
```
- ❑ **headerText** — задает текст заголовка сообщения. Если свойство имеет значение `null`, то заголовок сообщения не отображается. Методы доступа:
 

```
public final void setHeaderText(String headerText)
public final String getHeaderText()
public final StringProperty headerTextProperty()
```
- ❑ **contentText** — задает текст сообщения. Методы доступа:
 

```
public final void setContentText(String contentText)
public final String getContentText()
public final StringProperty contentTextProperty()
```
- ❑ **graphic** — позволяет отобразить узел справа от текста заголовка сообщения или слева от текста сообщения, если заголовок сообщения не отображается. Методы доступа:
 

```
public final void setGraphic(Node graphic)
public final Node getGraphic()
public final ObjectProperty<Node> graphicProperty()
```

Основные методы класса `Dialog<R>`:

- ❑ `show()` — отображает диалоговое окно на экране без ожидания действия пользователя (неблокирующий диалог). Используется, когда не нужно определять, какая кнопка нажата пользователем, или не надо получать данные из диалогового окна. Формат метода:

```
public final void show()
```

- ❑ `showAndWait()` — отображает диалоговое окно и ожидает действия пользователя (блокирующий диалог). Используется, когда нужно получить данные из диалогового окна (см. *разд. 16.1.6*). Формат метода:

```
import java.util.Optional;
public final Optional<R> showAndWait()
```

- ❑ `hide()` и `close()` — скрывают диалоговое окно. Форматы методов:

```
public final void hide()
public final void close()
```

- ❑ `initModality()` — указывает, является ли диалоговое окно модальным (блокирующим другие окна). Формат метода:

```
import javafx.stage.Modality;
public final void initModality(Modality modality)
```

В качестве параметра могут быть указаны следующие константы из перечисления `Modality`:

- `NONE` — окно не является модальным;
- `WINDOW_MODAL` — окно блокирует только родительские окна в пределах иерархии. При использовании этого значения необходимо указать ссылку на родительское окно с помощью метода `initOwner()`;
- `APPLICATION_MODAL` — окно блокирует все окна приложения (значение по умолчанию);

- ❑ `getModality()` — позволяет узнать, является ли окно модальным. Формат метода:

```
public final Modality getModality()
```

- ❑ `initOwner()` — позволяет указать ссылку на родительское окно. Формат метода:

```
public final void initOwner(Window window)
```

- ❑ `getOwner()` — возвращает ссылку на родительское окно или значение `null`. Формат метода:

```
public final Window getOwner()
```

- ❑ `initStyle()` — задает стиль окна (см. *разд. 2.2*). Формат метода:

```
public final void initStyle(StageStyle style)
```

**ЭЛЕКТРОННЫЙ АРХИВ**

Полный текст программы содержится в проекте `FX_CH16_01_Dialog`, приведенном в каталоге *CH16\_dialog* сопровождающего книгу электронного архива (см. приложение).

## 16.1.2. Размеры и местоположение диалогового окна

Управлять размерами и местоположением диалогового окна позволяют следующие свойства из класса `Dialog<R>`:

- ❑ `x` — местоположение окна по оси `x`. Методы доступа:

```
public final void setX(double x)
public final double getX()
public final ReadOnlyDoubleProperty xProperty()
```

- ❑ `y` — местоположение окна по оси `y`. Методы доступа:

```
public final void setY(double y)
public final double getY()
public final ReadOnlyDoubleProperty yProperty()
```

- ❑ `width` — ширина окна. Методы доступа:

```
public final void setWidth(double width)
public final double getWidth()
public final ReadOnlyDoubleProperty widthProperty()
```

- ❑ `height` — высота окна. Методы доступа:

```
public final void setHeight(double height)
public final double getHeight()
public final ReadOnlyDoubleProperty heightProperty()
```

- ❑ `resizable` — если свойство имеет значение `true`, то пользователь сможет изменять размеры диалогового окна, а если `false` (значение по умолчанию) — то нет. Методы доступа:

```
public final void setResizable(boolean resizable)
public final boolean isResizable()
public final BooleanProperty resizableProperty()
```

**ЭЛЕКТРОННЫЙ АРХИВ**

Полный текст программы содержится в проекте `FX_CH16_01_Dialog`, приведенном в каталоге *CH16\_dialog* сопровождающего книгу электронного архива (см. приложение).

## 16.1.3. Обработка событий

Назначить обработчики событий можно с помощью следующих свойств:

- ❑ `showing` — содержит значение `true`, если окно отображается на экране, и `false` — если скрыто. Методы доступа:

```
public final boolean isShowing()
public final ReadOnlyBooleanProperty showingProperty()
```

**Пример назначения обработчика:**

```
dialog.showingProperty().addListener(
 (obj, oldValue, newValue) -> {
 System.out.println("showing " + newValue);
 });
```

- ❑ **onShowing** — задает обработчик события, возникающего перед отображением окна. Методы доступа:

```
public final void setOnShowing(EventHandler<DialogEvent> value)
public final EventHandler<DialogEvent> getOnShowing()
public final ObjectProperty<EventHandler<DialogEvent>>
 onShowingProperty()
```

**Пример назначения обработчика:**

```
dialog.setOnShowing(e -> {
 System.out.println("OnShowing");
});
```

- ❑ **onShown** — задает обработчик события отображения окна. Методы доступа:

```
public final void setOnShown(EventHandler<DialogEvent> value)
public final EventHandler<DialogEvent> getOnShown()
public final ObjectProperty<EventHandler<DialogEvent>>
 onShownProperty()
```

- ❑ **onHiding** — задает обработчик события, возникающего перед сокрытием окна. Методы доступа:

```
public final void setOnHiding(EventHandler<DialogEvent> value)
public final EventHandler<DialogEvent> getOnHiding()
public final ObjectProperty<EventHandler<DialogEvent>>
 onHidingProperty()
```

- ❑ **onHidden** — задает обработчик события сокрытия окна. Методы доступа:

```
public final void setOnHidden(EventHandler<DialogEvent> value)
public final EventHandler<DialogEvent> getOnHidden()
public final ObjectProperty<EventHandler<DialogEvent>>
 onHiddenProperty()
```

Пользователь может закрыть диалоговое окно с помощью нажатия кнопки **Закрыть** в заголовке окна. Если необходимо перехватить и обработать закрытие окна этим способом, то внутри программы следует назначить обработчик события с помощью свойства `onCloseRequest`. Методы доступа:

```
public final void setOnCloseRequest(EventHandler<DialogEvent> value)
public final EventHandler<DialogEvent> getOnCloseRequest()
public final ObjectProperty<EventHandler<DialogEvent>>
 onCloseRequestProperty()
```

Если внутри обработчика через объект события вызвать метод `consume()`, то окно закрыто не будет. Пример:

```
dialog.setOnCloseRequest(e -> {
 Alert alert = new Alert(AlertType.CONFIRMATION,
 "Вы действительно хотите закрыть окно?");
 alert.setTitle("Закрытие окна");
 alert.setHeaderText(null);
 Optional<ButtonType> result = alert.showAndWait();
 if (result.isPresent() && result.get() == ButtonType.OK) {
 System.out.println("Окно будет закрыто");
 }
 else {
 System.out.println("Окно не будет закрыто");
 e.consume();
 }
});
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH16_01_Dialog`, приведенном в каталоге `CH16_dialog` сопровождающего книгу электронного архива (см. приложение).

## 16.1.4. Класс *DialogPane*

Содержимое диалогового окна располагается внутри контейнера, реализуемого с помощью класса `DialogPane`. Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.DialogPane;
```

Иерархия наследования:

Object — Node — Parent — Region — Pane — DialogPane

Конструктор класса:

```
DialogPane()
```

Получить ссылку на объект контейнера позволяет свойство `dialogPane` из класса `Dialog<R>`. Методы доступа:

```
public final void setDialogPane(DialogPane value)
public final DialogPane getDialogPane()
public final ObjectProperty<DialogPane> dialogPaneProperty()
```

Пример использования класса `DialogPane` приведен в листинге 16.2.

### Листинг 16.2. Класс `DialogPane`

```
Dialog<ButtonType> dialog = new Dialog<ButtonType>();
dialog.setTitle("Заголовок окна");

DialogPane dialogPane = dialog.getDialogPane();
dialogPane.setHeaderText("Текст заголовка сообщения");
```

```
dialogPane.setGraphic(new ImageView(createImage(16, 16, Color.BLUE)));
dialogPane.setContentText("Текст сообщения");
dialogPane.setExpanded(false);
dialogPane.setExpandableContent(new Label("Скрытый узел"));
dialogPane.getButtonTypes().add(ButtonType.OK);
```

```
dialog.show();
```

Класс `DialogPane` содержит следующие свойства:

❑ **header** — задает узел, отображаемый в области заголовка. Методы доступа:

```
public final void setHeader(Node header)
public final Node getHeader()
public final ObjectProperty<Node> headerProperty()
```

❑ **headerText** — задает текст заголовка сообщения. Если свойство имеет значение `null`, то заголовок сообщения не отображается. Если указано значение для свойства **header**, то свойство **headerText** не используется. Методы доступа:

```
public final void setHeaderText(String headerText)
public final String getHeaderText()
public final StringProperty headerTextProperty()
```

❑ **content** — задает узел, отображаемый в области содержимого. Методы доступа:

```
public final void setContent(Node content)
public final Node getContent()
public final ObjectProperty<Node> contentProperty()
```

❑ **contentText** — задает текст сообщения. Если указано значение для свойства **content**, то свойство **contentText** не используется. Методы доступа:

```
public final void setContentText(String contentText)
public final String getContentText()
public final StringProperty contentTextProperty()
```

❑ **graphic** — позволяет отобразить узел справа от текста заголовка сообщения или слева от текста сообщения, если заголовок сообщения не отображается. Методы доступа:

```
public final void setGraphic(Node graphic)
public final Node getGraphic()
public final ObjectProperty<Node> graphicProperty()
```

❑ **expandableContent** — задает узел, отображаемый в расширяемой области. Если свойство имеет значение, то автоматически отображается кнопка **Show Details**, с помощью которой можно отобразить расширяемую область на экране. Методы доступа:

```
public final void setExpandableContent(Node content)
public final Node getExpandableContent()
public final ObjectProperty<Node> expandableContentProperty()
```



- `expanded` — если свойство имеет значение `true`, то расширяемая область будет показана, а если `false` — то скрыта. Методы доступа:

```
public final void setExpanded(boolean value)
public final boolean isExpanded()
public final BooleanProperty expandedProperty()
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH16_01_Dialog`, приведенном в каталоге `CH16_dialog` сопровождающего книгу электронного архива (см. приложение).

## 16.1.5. Класс *ButtonType*: кнопки внутри диалогового окна

Для добавления кнопок в диалоговое окно следует воспользоваться методом `getButtonTypes()` из класса `DialogPane`. Формат метода:

```
public final ObservableList<ButtonType> getButtonTypes()
```

Пример добавления кнопок **ОК** и **Cancel**:

```
dialog.getDialogPane().getButtonTypes().addAll(
 ButtonType.OK, ButtonType.CANCEL);
```

Если требуется изменить свойства кнопки, то нужно получить ссылку на нее с помощью метода `lookupButton()` из класса `DialogPane`. Формат метода:

```
public final Node lookupButton(ButtonType buttonType)
```

Сделаем кнопку **ОК** неактивной:

```
Node node = dialog.getDialogPane().lookupButton(ButtonType.OK);
if (node != null) node.setDisable(true);
```

Создать кнопку для диалогового окна позволяет класс `ButtonType`. Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.ButtonType;
```

Конструкторы класса:

```
ButtonType(String text)
ButtonType(String text, ButtonData buttonData)
```

Параметр `text` задает текст, отображаемый на кнопке, а параметр `buttonData` — предназначение кнопки. В первом конструкторе параметр `buttonData` имеет значение `ButtonData.OTHER`. Пример создания и добавления кнопок:

```
dialog.getDialogPane().getButtonTypes().addAll(
 new ButtonType("ОК", ButtonData.OK_DONE),
 new ButtonType("Отмена", ButtonData.CANCEL_CLOSE));
```

В качестве значения параметра `buttonData` можно указать следующие константы из перечисления `ButtonBar.ButtonData`: `YES`, `NO`, `OK_DONE`, `CANCEL_CLOSE`, `FINISH`, `APPLY`, `BACK_PREVIOUS`, `NEXT_FORWARD`, `LEFT`, `RIGHT`, `SMALL_GAP`, `BIG_GAP`, `HELP`, `HELP_2` или `OTHER`.

Получить текст, отображаемый на кнопке, и предназначение кнопки позволяют методы `getText()` и `getButtonData()` соответственно. Форматы методов:

```
public final String getText()
public final ButtonData getButtonData()
```

Для создания стандартных кнопок можно также воспользоваться следующими статическими константами из класса `ButtonType`:

```
public static final ButtonType OK
public static final ButtonType CANCEL
public static final ButtonType CLOSE
public static final ButtonType YES
public static final ButtonType NO
public static final ButtonType APPLY
public static final ButtonType FINISH
public static final ButtonType PREVIOUS
public static final ButtonType NEXT
```

Пример добавления кнопки **ОК**:

```
dialog.getDialogPane().getButtonTypes().add(ButtonType.OK);
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH16_01_Dialog`, приведенном в каталоге `CH16_dialog` сопровождающего книгу электронного архива (см. приложение).

## 16.1.6. Получение результата

После сокрытия диалогового окна часто нужно определить, какую кнопку нажал пользователь, и получить данные из диалогового окна. При вызове метода `showAndWait()` диалоговое окно отображается, и после нажатия кнопки пользователем метод возвращает объект класса `Optional<R>`, внутри которого содержится результат. Формат метода:

```
import java.util.Optional;
public final Optional<R> showAndWait()
```

Давайте создадим диалоговое окно с кнопками **ОК** и **Отмена** и определим, какую кнопку нажал пользователь (листинг 16.3).

### Листинг 16.3. Диалоговое окно с сообщением и кнопками ОК и Отмена

```
Button button = new Button("Открыть окно с кнопками ОК и Отмена");
button.setOnAction(event -> {
 Dialog<ButtonType> dialog = new Dialog<ButtonType>();
 dialog.setTitle("Заголовок окна");
 dialog.setHeaderText("Текст заголовка сообщения");
 dialog.setContentText("Текст сообщения");
 dialog.getDialogPane().getButtonTypes().addAll(
 new ButtonType("ОК", ButtonData.OK_DONE),
 new ButtonType("Отмена", ButtonData.CANCEL_CLOSE));
});
```

```
Optional<ButtonType> result = dialog.showAndWait();
if (result.isPresent()) {
 if (result.orElseThrow().getButtonData() == ButtonData.OK_DONE) {
 System.out.println("Нажата кнопка ОК");
 }
 else if (result.orElseThrow().getButtonData() ==
 ButtonData.CANCEL_CLOSE) {
 System.out.println("Нажата кнопка Отмена или кнопка Закрыть");
 }
}
});
```

Для получения результата можно воспользоваться следующими свойствами из класса `Dialog<R>`:

- ❑ `result` — содержит результат после обработки. Методы доступа:

```
public final void setResult(R value)
public final R getResult()
public final ObjectProperty<R> resultProperty()
```

Пример назначения обработчика:

```
dialog.resultProperty().addListener(
 (obj, oldValue, newValue) -> {
 System.out.println("result " + newValue);
 });
```

- ❑ `resultConverter` — позволяет назначить конвертер для предварительной обработки результата. Методы доступа:

```
public final void setResultConverter(
 Callback<ButtonType, R> value)
public final Callback<ButtonType, R> getResultConverter()
public final ObjectProperty<Callback<ButtonType, R>>
 resultConverterProperty()
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH16_01_Dialog`, приведенном в каталоге `CH16_dialog` сопровождающего книгу электронного архива (см. приложение).

## 16.2. Класс *Alert*: окно с сообщением

Класс `Alert` наследует класс `Dialog<R>` (см. *разд. 16.1*) и реализует модальное диалоговое окно с сообщениями различных типов. Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.Alert;
```

Инструкция импорта:

```
Object — Dialog<ButtonType> — Alert
```

## Конструкторы класса:

```
Alert (Alert.AlertType alertType)
Alert (Alert.AlertType alertType, String contentText,
 ButtonType... buttons)
```

Параметр `contentText` задает текст сообщения, а с помощью параметра `buttons` можно добавить кнопки в диалоговое окно. В параметре `alertType` указывается тип диалогового окна в виде следующих констант из перечисления `Alert.AlertType`:

- ❑ `INFORMATION` — окно с информационным сообщением (рис. 16.2). Пример:

```
Alert alert = new Alert (AlertType.INFORMATION,
 "Текст сообщения");
alert.setTitle("Заголовок окна");
alert.setHeaderText("Окно INFORMATION");
alert.show();
```

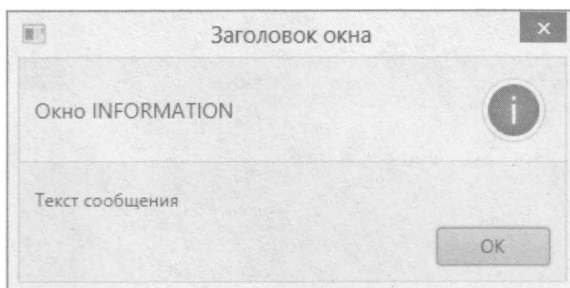


Рис. 16.2. Окно с информационным сообщением

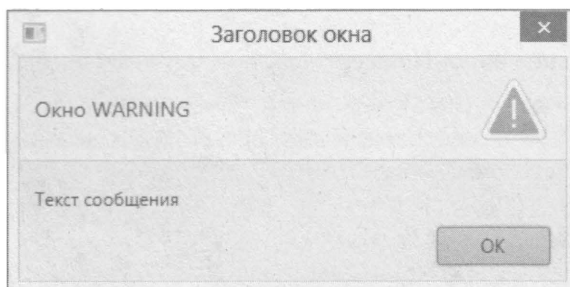


Рис. 16.3. Окно с предупреждающим сообщением

- ❑ `WARNING` — окно с предупреждающим сообщением (рис. 16.3);
- ❑ `ERROR` — окно с сообщением об ошибке (рис. 16.4);
- ❑ `CONFIRMATION` — окно с запросом подтверждения (рис. 16.5). Пример:

```
Alert alert = new Alert (AlertType.CONFIRMATION, "Текст вопроса");
alert.setTitle("Заголовок окна");
alert.setHeaderText("Окно CONFIRMATION");
Optional<ButtonType> result = alert.showAndWait();
```

```

if (result.isPresent() && result.get() == ButtonType.OK) {
 System.out.println("Нажата кнопка OK");
}
else System.out.println("Нажата кнопка Cancel");

```

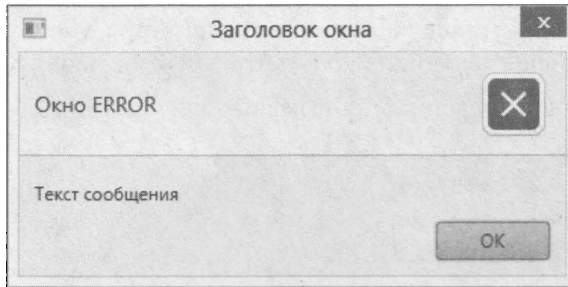


Рис. 16.4. Окно с сообщением об ошибке

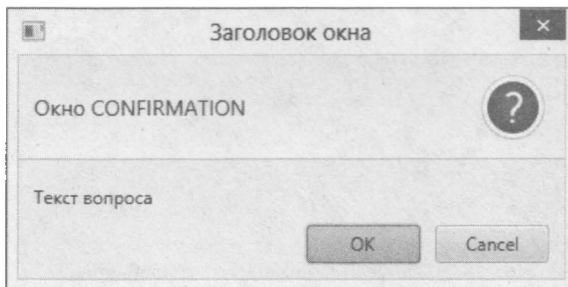


Рис. 16.5. Окно с запросом подтверждения действия

#### □ NONE — произвольное окно. Пример:

```

Alert alert = new Alert(AlertType.NONE, "Текст вопроса",
 ButtonType.OK, ButtonType.CANCEL);
alert.setTitle("Заголовок окна");
alert.setHeaderText("Окно NONE");
Optional<ButtonType> result = alert.showAndWait();
if (result.isPresent() && result.get() == ButtonType.OK) {
 System.out.println("Нажата кнопка OK");
}
else System.out.println("Нажата кнопка Cancel");

```

Управлять типом окна позволяет свойство `alertType`. Методы доступа:

```

public final void setAlertType(Alert.AlertType alertType)
public final Alert.AlertType getAlertType()
public final ObjectProperty<Alert.AlertType> alertTypeProperty()

```

Добавить кнопки можно с помощью метода `getButtonTypes()`. Формат метода:

```

public final ObservableList<ButtonType> getButtonTypes()

```

**ЭЛЕКТРОННЫЙ АРХИВ**

Полный текст программы содержится в проекте `FX_CH16_01_Dialog`, приведенном в каталоге `CH16_dialog` сопровождающего книгу электронного архива (см. приложение).

## 16.3. Класс *TextInputDialog*: окно с текстовым полем

Класс `TextInputDialog` наследует класс `Dialog<R>` (см. разд. 16.1) и реализует модальное диалоговое окно с текстовым полем (рис. 16.6). Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.control.TextInputDialog;
```

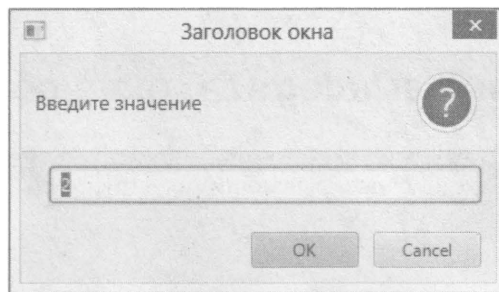


Рис. 16.6. Диалоговое окно с текстовым полем

Иерархия наследования:

`Object` — `Dialog<String>` — `TextInputDialog`

Конструкторы класса:

```
TextInputDialog()
TextInputDialog(String defaultValue)
```

В параметре `defaultValue` можно указать текст, отображаемый в текстовом поле по умолчанию. Получить значение по умолчанию позволяет метод `getDefaultValue()`.

Формат метода:

```
public final String getDefaultValue()
```

Для получения ссылки на текстовое поле предназначен метод `getEditor()`. Формат метода:

```
public final TextField getEditor()
```

Пример создания диалогового окна и получения результата приведен в листинге 16.4.

### Листинг 16.4. Диалоговое окно с текстовым полем

```
Button button = new Button("Открыть окно");
button.setOnAction(event -> {
 TextInputDialog dialog = new TextInputDialog("2");
```

```

dialog.setTitle("Заголовок окна");
dialog.setHeaderText("Введите значение");
dialog.getEditor().setPrefWidth(300.0);
Optional<String> result = dialog.showAndWait();
if (result.isPresent()) {
 System.out.println(result.get());
}
else System.out.println("Нажата кнопка Cancel");
});

```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH16_01_Dialog`, приведенном в каталоге `CH16_dialog` сопровождающего книгу электронного архива (см. *приложение*).

## 16.4. Класс *ChoiceDialog<T>*: окно со списком

Класс `ChoiceDialog<T>` наследует класс `Dialog<R>` (см. *разд. 16.1*) и реализует модальное диалоговое окно с раскрывающимся списком (рис. 16.7). Инструкция импорта:

```

// Модуль javafx.controls
import javafx.scene.control.ChoiceDialog;

```

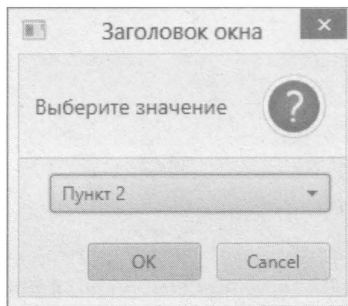


Рис. 16.7. Диалоговое окно со списком

### Иерархия наследования:

Object — Dialog<T> — ChoiceDialog<T>

### Конструкторы класса:

```

ChoiceDialog()
ChoiceDialog(T defaultChoice, T... choices)
ChoiceDialog(T defaultChoice, Collection<T> choices)

```

Параметр `choices` задает элементы, которые будут отображаться в списке. Добавить элементы после создания объекта позволяет метод `getItems()`. Формат метода:

```

public final ObservableList<T> getItems()

```

Параметр `defaultChoice` задает выбранный элемент списка. Этот элемент должен входить в состав элементов из параметра `choices`. Получить элемент по умолчанию или значение `null` позволяет метод `getDefaultChoice()`. Формат метода:

```
public final T getDefaultChoice()
```

Для управления выбором элемента предназначено свойство `selectedItem`. Методы доступа:

```
public final void setSelectedItem(T item)
public final T getSelectedItem()
public final ReadOnlyObjectProperty<T> selectedItemProperty()
```

Пример создания диалогового окна и получения результата приведен в листинге 16.5.

#### Листинг 16.5. Диалоговое окно со списком

```
Button button = new Button("Открыть окно");
button.setOnAction(event -> {
 ChoiceDialog<String> dialog = new ChoiceDialog<String>(
 "Пункт 2", "Пункт 1", "Пункт 2", "Пункт 3");
 dialog.setTitle("Заголовок окна");
 dialog.setHeaderText("Выберите значение");

 Optional<String> result = dialog.showAndWait();
 if (result.isPresent()) {
 System.out.println(result.get());
 }
 else System.out.println("Нажата кнопка Cancel");
});
```

#### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH16_01_Dialog`, приведенном в каталоге `CH16_dialog` сопровождающего книгу электронного архива (см. приложение).

## 16.5. Класс *DirectoryChooser*: окно для выбора папки

Класс `DirectoryChooser` реализует модальное диалоговое окно для выбора папки (рис. 16.8). Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.stage.DirectoryChooser;
```

Иерархия наследования:

`Object` — `DirectoryChooser`



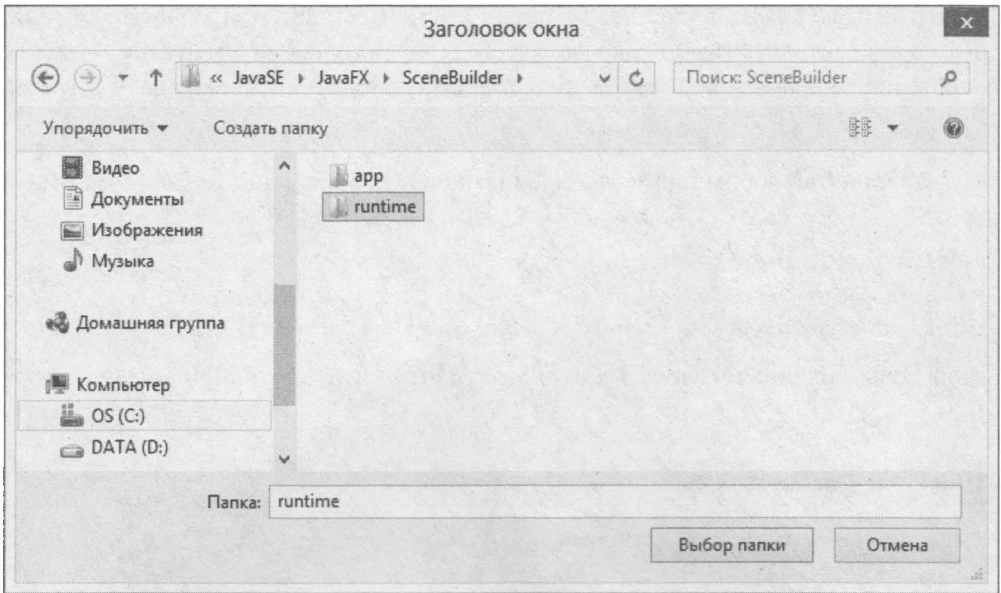


Рис. 16.8. Диалоговое окно для выбора папки

### Конструктор класса:

`DirectoryChooser()`

Для отображения диалогового окна предназначен метод `showDialog()`. После выбора папки пользователем метод вернет результат или значение `null`, если будет нажата кнопка **Отмена** или кнопка **Заккрыть** в заголовке окна. В параметре `ownerWindow` указывается ссылка на родительское окно. Формат метода:

```
public File showDialog(Window ownerWindow)
```

Класс `DirectoryChooser` содержит следующие свойства:

❑ `title` — задает текст, отображаемый в заголовке окна. Методы доступа:

```
public final void setTitle(String value)
public final String getTitle()
public final StringProperty titleProperty()
```

❑ `initialDirectory` — задает начальную папку. Методы доступа:

```
public final void setInitialDirectory(File value)
public final File getInitialDirectory()
public final ObjectProperty<File> initialDirectoryProperty()
```

Пример создания диалогового окна и получения результата приведен в листинге 16.6.

#### Листинг 16.6. Диалоговое окно для выбора папки

```
Button button = new Button("Открыть окно");
button.setOnAction(event -> {
 DirectoryChooser dialog = new DirectoryChooser();
```

```
dialog.setTitle("Заголовок окна");
dialog.setInitialDirectory(
 new File("C:\\JavaSE\\JavaFX\\SceneBuilder"));

File result = dialog.showDialog(window);
if (result != null) System.out.println(result);
else System.out.println("Нажата кнопка Отмена");
});
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH16_01_Dialog`, приведенном в каталоге `CH16_dialog` сопровождающего книгу электронного архива (см. *приложение*).

## 16.6. Класс *FileChooser*: окно для выбора файла

Класс `FileChooser` реализует модальное диалоговое окно для выбора одного или нескольких файлов (рис. 16.9), а также для сохранения файла (рис. 16.10). Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.stage.FileChooser;
```

Иерархия наследования:

Object — `FileChooser`

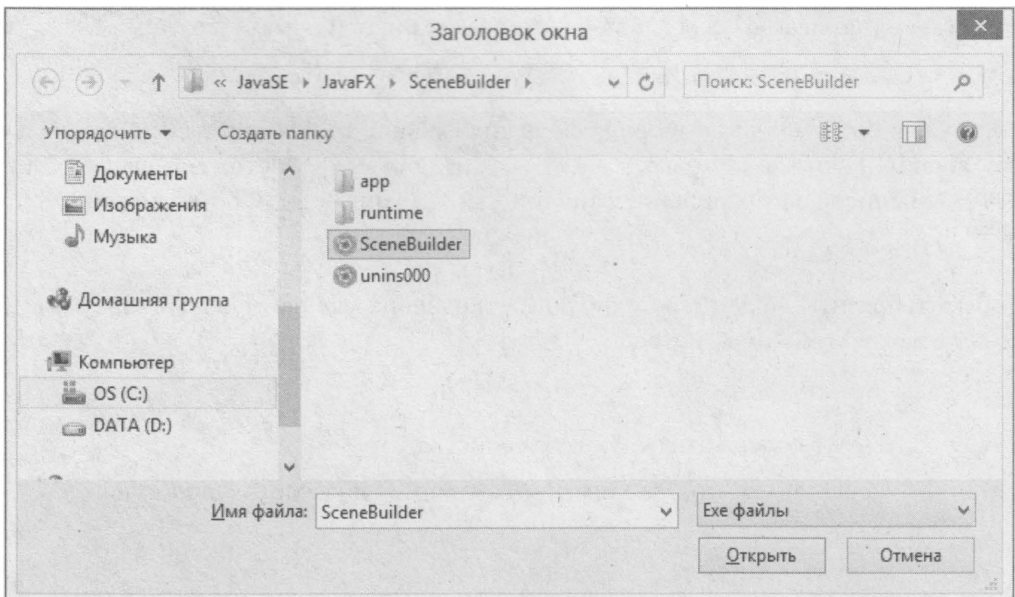


Рис. 16.9. Диалоговое окно для выбора файла

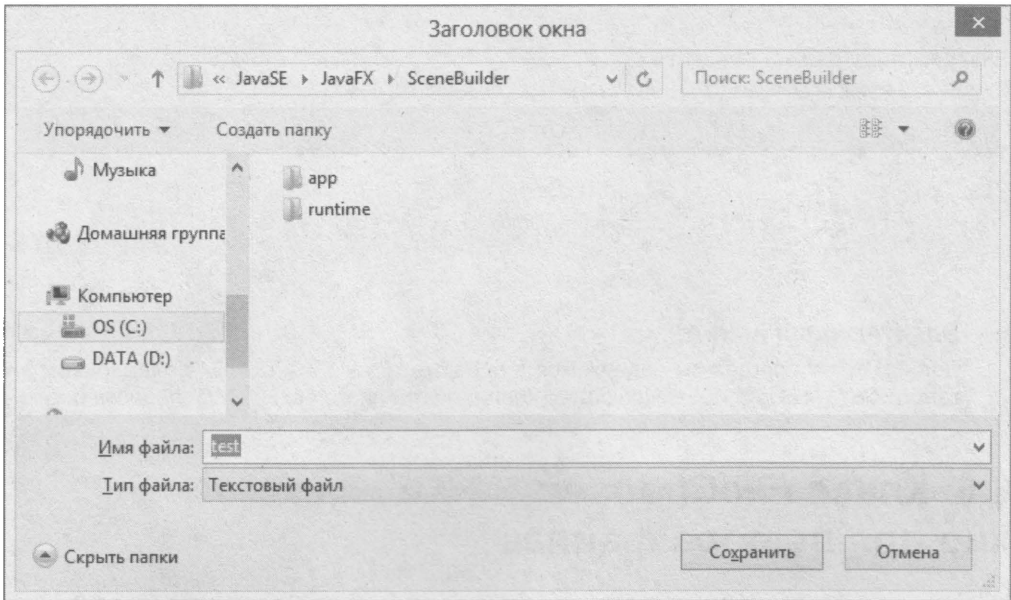


Рис. 16.10. Диалоговое окно для сохранения файла

### Конструктор класса:

```
FileChooser()
```

Для отображения диалогового окна предназначен метод `showOpenDialog()`. После выбора файла пользователем метод вернет результат или значение `null`, если будет нажата кнопка **Отмена** или кнопка **Заккрыть** в заголовке окна. В параметре `ownerWindow` указывается ссылка на родительское окно. Формат метода:

```
public File showOpenDialog(Window ownerWindow)
```

Если нужна возможность выбора нескольких файлов, то для отображения окна следует воспользоваться методом `showOpenMultipleDialog()`. Метод возвращает неизменяемый список с выбранными файлами или значение `null`. Формат метода:

```
public List<File> showOpenMultipleDialog(Window ownerWindow)
```

Чтобы отобразить диалоговое окно для сохранения файла, нужно вызвать метод `showSaveDialog()`. Формат метода:

```
public File showSaveDialog(Window ownerWindow)
```

Класс `FileChooser` содержит следующие свойства:

□ `title` — задает текст, отображаемый в заголовке окна. Методы доступа:

```
public final void setTitle(String value)
public final String getTitle()
public final StringProperty titleProperty()
```

❑ `initialDirectory` — задает начальную папку. Методы доступа:

```
public final void setInitialDirectory(File value)
public final File getInitialDirectory()
public final ObjectProperty<File> initialDirectoryProperty()
```

❑ `initialFileName` — задает имя для сохранения файла. Методы доступа:

```
public final void setInitialFileName(String value)
public final String getInitialFileName()
public final ObjectProperty<String> initialFileNameProperty()
```

❑ `selectedExtensionFilter` — задает фильтр. Методы доступа:

```
public final void setSelectedExtensionFilter(
 FileChooser.ExtensionFilter filter)
public final FileChooser.ExtensionFilter
 getSelectedExtensionFilter()
public final ObjectProperty<FileChooser.ExtensionFilter>
 selectedExtensionFilterProperty()
```

Получить ссылку на все установленные фильтры позволяет метод `getExtensionFilters()`. Формат метода:

```
public ObservableList<FileChooser.ExtensionFilter> getExtensionFilters()
```

Класс `FileChooser.ExtensionFilter` содержит следующие конструкторы:

```
ExtensionFilter(String description, String... extensions)
ExtensionFilter(String description, List<String> extensions)
```

В первом параметре указывается название фильтра, а во втором — значения в формате `*.расширение` через запятую или в виде списка.

Пример создания диалогового окна и получения результата приведен в листинге 16.7.

#### Листинг 16.7. Диалоговое окно для выбора файла

```
Button button = new Button("Выбрать файл");
button.setOnAction(event -> {
 FileChooser dialog = new FileChooser();
 dialog.setTitle("Заголовок окна");
 dialog.setInitialDirectory(
 new File("C:\\JavaSE\\JavaFX\\SceneBuilder"));
 dialog.getExtensionFilters().addAll(
 new FileChooser.ExtensionFilter("Ехе файлы", "*.exe"),
 new FileChooser.ExtensionFilter(
 "Текстовые файлы", "*.txt", "*.doc"),
 new FileChooser.ExtensionFilter("Все файлы", "*.*")
);
});
```

```
File result = dialog.showOpenDialog(window);
if (result != null) System.out.println(result);
else System.out.println("Нажата кнопка Cancel");
});
```

### **ЭЛЕКТРОННЫЙ АРХИВ**

Полный текст программы содержится в проекте `FX_CH16_01_Dialog`, приведенном в каталоге *CH16\_dialog* сопровождающего книгу электронного архива (см. *приложение*).

# ГЛАВА 17



## Графики и диаграммы

Библиотека JavaFX содержит множество классов, реализующих различные виды диаграмм — например: круговую диаграмму, линейный график, гистограмму и т. д. Давайте рассмотрим эти классы подробно.

### **ПРИМЕЧАНИЕ**

При использовании программы Scene Builder все диаграммы можно найти на вкладке **Charts**.

### 17.1. Класс *Chart*

Абстрактный класс `Chart` является базовым для всех классов, реализующих диаграммы. Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.chart.Chart;
```

Иерархия наследования:

Object — Node — Parent — Region — Chart

Класс `Chart` содержит следующие основные свойства:

❑ `title` — текст заголовка. Методы доступа:

```
public final void setTitle(String value)
public final String getTitle()
public final StringProperty titleProperty()
```

❑ `titleSide` — задает местоположение заголовка. Можно указать константы: `TOP` (значение по умолчанию), `BOTTOM`, `LEFT` и `RIGHT` из перечисления `Side`. Методы доступа:

```
import javafx.geometry.Side;
public final void setTitleSide(Side value)
public final Side getTitleSide()
public final ObjectProperty<Side> titleSideProperty()
```

- ❑ `legendSide` — задает сторону диаграммы, с которой будет выводиться блок с именами областей и соответствующими им цветами. Можно указать константы: `TOP`, `BOTTOM` (значение по умолчанию), `LEFT` и `RIGHT` из перечисления `Side`. Методы доступа:

```
import javafx.geometry.Side;
public final void setLegendSide(Side value)
public final Side getLegendSide()
public final ObjectProperty<Side> legendSideProperty()
```

- ❑ `legendVisible` — если свойство имеет значение `false`, то блок с именами областей и соответствующими им цветами отображаться не будет. Методы доступа:

```
public final void setLegendVisible(boolean value)
public final boolean isLegendVisible()
public final BooleanProperty legendVisibleProperty()
```

- ❑ `animated` — если свойство имеет значение `true`, то изменение значений будет выполняться с анимацией, а не сразу. Методы доступа:

```
public final void setAnimated(boolean value)
public final boolean getAnimated()
public final BooleanProperty animatedProperty()
```

## 17.2. Класс *PieChart*: круговая диаграмма

Класс `PieChart` создает круговую диаграмму (рис. 17.1). Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.chart.PieChart;
```

Иерархия наследования:

`Object` — `Node` — `Parent` — `Region` — `Chart` — `PieChart`

Конструкторы класса:

```
PieChart()
PieChart(ObservableList<PieChart.Data> data)
```

Класс `PieChart` наследует все свойства из класса `Chart` (см. *разд. 17.1*) и добавляет следующие свойства:

- ❑ `startAngle` — задает начальный угол для первой области. Методы доступа:

```
public final void setStartAngle(double value)
public final double getStartAngle()
public final DoubleProperty startAngleProperty()
```

- ❑ `clockwise` — если свойство имеет значение `true`, то области будут следовать по часовой стрелке, начиная от угла `startAngle`, а если `false` — то против часовой стрелки. Методы доступа:

```
public final void setClockwise(boolean value)
public final boolean isClockwise()
public final BooleanProperty clockwiseProperty()
```

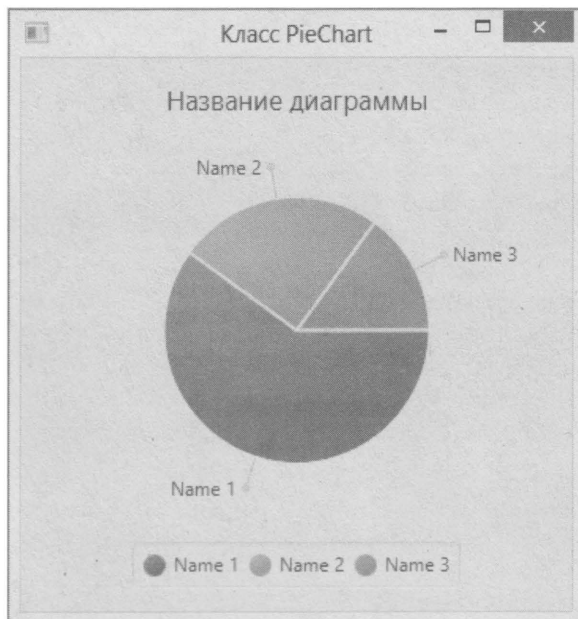


Рис. 17.1. Класс PieChart: круговая диаграмма

- ❑ **labelsVisible** — если свойство имеет значение `true`, то рядом с областями будут отображаться их названия, а если `false` — то нет. Методы доступа:

```
public final void setLabelsVisible(boolean value)
public final boolean getLabelsVisible()
public final BooleanProperty labelsVisibleProperty()
```

- ❑ **labelLineLength** — задает длину линии между областью и ее названием. Методы доступа:

```
public final void setLabelLineLength(double value)
public final double getLabelLineLength()
public final DoubleProperty labelLineLengthProperty()
```

- ❑ **data** — задает список с отображаемыми данными. Любые изменения списка отразятся на диаграмме. Методы доступа:

```
public final void setData(ObservableList<PieChart.Data> value)
public final ObservableList<PieChart.Data> getData()
public final ObjectProperty<ObservableList<PieChart.Data>>
 dataProperty()
```

Для описания области на диаграмме предназначен класс `PieChart.Data`. Конструктор класса:

```
PieChart.Data(String name, double value)
```



В первом параметре указывается название области, а во втором — значение. Пример создания круговой диаграммы приведен в листинге 17.1.

#### Листинг 17.1. Класс `PieChart`

```
ObservableList<PieChart.Data> list =
 FXCollections.<PieChart.Data>observableArrayList();
list.add(new PieChart.Data("Name 1", 60.0));
list.add(new PieChart.Data("Name 2", 25.0));
list.add(new PieChart.Data("Name 3", 15.0));

PieChart chart = new PieChart();
chart.setTitle("Название диаграммы");
chart.setTitleSide(Side.TOP);
chart.setLegendSide(Side.BOTTOM);
chart.setLegendVisible(true);
chart.setAnimated(true);

chart.setData(list);
chart.setStartAngle(0.0);
chart.setClockwise(true);
chart.setLabelsVisible(true);
chart.setLabelLineLength(20.0);

VBox pane = new VBox(10.0);
pane.getChildren().add(chart);
```

#### **ЭЛЕКТРОННЫЙ АРХИВ**

Полный текст программы содержится в проекте `FX_CH17_01_Chart`, приведенном в каталоге `CH17_charts` сопровождающего книгу электронного архива (см. приложение).

## 17.3. Класс `XYChart<X, Y>`

Абстрактный класс `XYChart<X, Y>` является базовым для всех классов диаграмм, использующих две оси (x и y). Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.chart.XYChart;
```

**Иерархия наследования:**

Object — Node — Parent — Region — Chart — `XYChart<X, Y>`

**Конструктор класса:**

```
XYChart(Axis<X> xAxis, Axis<Y> yAxis)
```

### 17.3.1. Класс *Axis<T>*: ось диаграммы

Абстрактный класс *Axis<T>* является базовым классом для всех классов, описывающих ось диаграммы. Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.chart.Axis;
```

Иерархия наследования:

Object — Node — Parent — Region — *Axis<T>*

Класс *Axis<T>* содержит следующие основные свойства:

- ❑ **animated** — если свойство имеет значение `true`, то изменения будут выполняться с анимацией. Методы доступа:

```
public final void setAnimated(boolean value)
public final boolean getAnimated()
public final BooleanProperty animatedProperty()
```

- ❑ **autoRanging** — если свойство имеет значение `true`, то диапазон определяется автоматически в соответствии с отображаемыми данными. Методы доступа:

```
public final void setAutoRanging(boolean value)
public final boolean isAutoRanging()
public final BooleanProperty autoRangingProperty()
```

- ❑ **label** — задает текст, отображаемый рядом с осью. Методы доступа:

```
public final void setLabel(String value)
public final String getLabel()
public final ObjectProperty<String> labelProperty()
```

- ❑ **side** — сторона диаграммы с осью. Методы доступа:

```
import javafx.geometry.Side;
public final void setSide(Side value)
public final Side getSide()
public final ObjectProperty<Side> sideProperty()
```

- ❑ **tickLabelsVisible** — если свойство имеет значение `false`, то надписи рядом с основными рисками отображаться не будут. Методы доступа:

```
public final void setTickLabelsVisible(boolean value)
public final boolean isTickLabelsVisible()
public final BooleanProperty tickLabelsVisibleProperty()
```

- ❑ **tickMarkVisible** — если свойство имеет значение `false`, то основные риски отображаться не будут. Методы доступа:

```
public final void setTickMarkVisible(boolean value)
public final boolean isTickMarkVisible()
public final BooleanProperty tickMarkVisibleProperty()
```

- ❑ `tickLength` — длина основных рисков. Методы доступа:

```
public final void setTickLength(double value)
public final double getTickLength()
public final DoubleProperty tickLengthProperty()
```

- ❑ `tickLabelFill` — цвет надписей рядом с основными рисками. Методы доступа:

```
public final void setTickLabelFill(Paint value)
public final Paint getTickLabelFill()
public final ObjectProperty<Paint> tickLabelFillProperty()
```

**Пример:**

```
NumberAxis xAxis = new NumberAxis();
xAxis.setTickLabelFill(Color.RED);
```

- ❑ `tickLabelFont` — характеристики шрифта надписи, расположенной рядом с основными рисками. Методы доступа:

```
public final void setTickLabelFont(Font value)
public final Font getTickLabelFont()
public final ObjectProperty tickLabelFontProperty()
```

- ❑ `tickLabelGap` — расстояние между надписями и основными рисками. Методы доступа:

```
public final void setTickLabelGap(double value)
public final double getTickLabelGap()
public final DoubleProperty tickLabelGapProperty()
```

- ❑ `tickLabelRotation` — угол, на который должны быть повернуты надписи, расположенные рядом с основными рисками. Методы доступа:

```
public final void setTickLabelRotation(double value)
public final double getTickLabelRotation()
public final DoubleProperty tickLabelRotationProperty()
```

Получить ссылку на неизменяемый список с метками позволяет метод `getTickMarks()`. Формат метода:

```
public ObservableList<Axis.TickMark<T>> getTickMarks()
```

Класс `Axis.TickMark<T>` содержит следующие свойства:

- ❑ `value` — значение. Методы доступа:

```
public final void setValue(T v)
public final T getValue()
public final ObjectExpression<T> valueProperty()
```

- ❑ `position` — позиция на оси. Методы доступа:

```
public final void setPosition(double value)
public final double getPosition()
public final DoubleExpression positionProperty()
```

□ **label** — текст надписи. Методы доступа:

```
public final void setLabel(String value)
public final String getLabel()
public final StringExpression labelProperty()
```

Методы класса `Axis.TickMark<T>`, управляющие отображением текста надписи:

```
public final void setTextVisible(boolean value)
public final boolean isTextVisible()
```

### 17.3.2. Классы *ValueAxis<T>* и *NumberAxis*: ось диаграммы с числовыми значениями

Для создания оси с числовыми значениями используется класс `NumberAxis`. Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.chart.NumberAxis;
```

Иерархия наследования:

`Object` — `Node` — `Parent` — `Region` — `Axis<T>` — `ValueAxis<Number>` — `NumberAxis`

Конструкторы класса:

```
NumberAxis()
NumberAxis(double lowerBound, double upperBound, double tickUnit)
NumberAxis(String axisLabel, double lowerBound, double upperBound,
 double tickUnit)
```

Первый конструктор создает ось с автоматическим определением диапазона значений шкалы. Второй и третий конструкторы в параметре `lowerBound` принимают минимальное значение, в параметре `upperBound` — максимальное значение шкалы, в параметре `tickUnit` — шаг для основных рисок, а в параметре `axisLabel` — текст, отображаемый рядом с осью:

```
NumberAxis yAxis = new NumberAxis("Ось Y", 0.0, 200.0, 50.0);
```

Класс `NumberAxis` наследует все свойства и методы из базовых классов и содержит следующие свойства:

□ **tickUnit** — шаг между основными рисками. Методы доступа:

```
public final void setTickUnit(double value)
public final double getTickUnit()
public final DoubleProperty tickUnitProperty()
```

□ **forceZeroInRange** — если свойство имеет значение `true`, то значение `0` всегда будет включено в видимый диапазон на оси. Методы доступа:

```
public final void setForceZeroInRange(boolean value)
public final boolean isForceZeroInRange()
public final BooleanProperty forceZeroInRangeProperty()
```

**Класс** `NumberAxis` наследует следующие свойства из класса `ValueAxis<Number>`:

```
import javafx.scene.chart.ValueAxis;
```

□ `lowerBound` — минимальное значение на оси. Методы доступа:

```
public final void setLowerBound(double value)
public final double getLowerBound()
public final DoubleProperty lowerBoundProperty()
```

□ `upperBound` — максимальное значение на оси. Методы доступа:

```
public final void setUpperBound(double value)
public final double getUpperBound()
public final DoubleProperty upperBoundProperty()
```

□ `minorTickVisible` — если свойство имеет значение `false`, то промежуточные риски отображаться не будут. Методы доступа:

```
public final void setMinorTickVisible(boolean value)
public final boolean isMinorTickVisible()
public final BooleanProperty minorTickVisibleProperty()
```

□ `minorTickCount` — количество промежуточных рисков. Методы доступа:

```
public final void setMinorTickCount(int value)
public final int getMinorTickCount()
public final IntegerProperty minorTickCountProperty()
```

□ `minorTickLength` — длина промежуточных рисков. Методы доступа:

```
public final void setMinorTickLength(double value)
public final double getMinorTickLength()
public final DoubleProperty minorTickLengthProperty()
```

□ `tickLabelFormatter` — задает способ преобразования значения в текст. Методы доступа:

```
public final void setTickLabelFormatter(StringConverter<T> value)
public final StringConverter<T> getTickLabelFormatter()
public final ObjectProperty<StringConverter<T>>
 tickLabelFormatterProperty()
```

Для форматирования числовых значений можно воспользоваться классом `NumberAxis.DefaultFormatter`. Конструкторы класса:

```
DefaultFormatter(NumberAxis axis)
DefaultFormatter(NumberAxis axis, String prefix, String suffix)
```

В первом параметре указывается ссылка на объект оси с числовыми значениями, во втором — строка, которая будет добавлена перед значением, а в третьем — строка, которая будет добавлена после значения. В параметрах `prefix` и `suffix` можно указать значение `null`.

Выведем значения внутри круглых скобок — например, (10):

```
yAxis.setTickLabelFormatter(
 new NumberAxis.DefaultFormatter(yAxis, "(", ")"));
```

- `scale` — коэффициент масштабирования. Методы доступа:

```
public final double getScale()
public final ReadOnlyDoubleProperty scaleProperty()
```

Пример создания оси и добавления к объекту линейного графика:

```
NumberAxis xAxis = new NumberAxis();
NumberAxis yAxis = new NumberAxis();
LineChart<Number, Number> chart = new LineChart<>(xAxis, yAxis);
chart.setTitle("Название графика");
```

Получить ссылку на оси диаграммы позволяют следующие методы из класса `XYChart<X, Y>`:

- `getXAxis()` — возвращает ссылку на объект горизонтальной оси. Формат метода:  

```
public Axis<X> getXAxis()
```
- `getYAxis()` — возвращает ссылку на объект вертикальной оси. Формат метода:  

```
public Axis<Y> getYAxis()
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH17_01_Chart`, приведенном в каталоге `CH17_charts` сопровождающего книгу электронного архива (см. *приложение*).

## 17.3.3. Свойства класса `XYChart<X, Y>`

Класс `XYChart<X, Y>` содержит следующие свойства:

- `horizontalGridLinesVisible` — если свойство имеет значение `true`, то будут нарисованы горизонтальные линии сетки, а если `false` — то нет. Методы доступа:  

```
public final void setHorizontalGridLinesVisible(boolean value)
public final boolean isHorizontalGridLinesVisible()
public final BooleanProperty horizontalGridLinesVisibleProperty()
```
- `verticalGridLinesVisible` — если свойство имеет значение `true`, то будут нарисованы вертикальные линии сетки, а если `false` — то нет. Методы доступа:  

```
public final void setVerticalGridLinesVisible(boolean value)
public final boolean getVerticalGridLinesVisible()
public final BooleanProperty verticalGridLinesVisibleProperty()
```
- `horizontalZeroLineVisible` — если свойство имеет значение `true`, то при наличии положительных и отрицательных значений будет проведена дополнительная горизонтальная осевая линия для нулевого значения. Методы доступа:  

```
public final void setHorizontalZeroLineVisible(boolean value)
public final boolean isHorizontalZeroLineVisible()
public final BooleanProperty horizontalZeroLineVisibleProperty()
```
- `verticalZeroLineVisible` — если свойство имеет значение `true`, то при наличии положительных и отрицательных значений будет проведена дополнительная вертикальная осевая линия для нулевого значения. Методы доступа:

```
public final void setVerticalZeroLineVisible(boolean value)
public final boolean isVerticalZeroLineVisible()
public final BooleanProperty verticalZeroLineVisibleProperty()
```

- ❑ **alternativeRowFillVisible** — если свойство имеет значение `true`, то альтернативные горизонтальные строки будут иметь заливку. Методы доступа:

```
public final void setAlternativeRowFillVisible(boolean value)
public final boolean isAlternativeRowFillVisible()
public final BooleanProperty alternativeRowFillVisibleProperty()
```

- ❑ **alternativeColumnFillVisible** — если свойство имеет значение `true`, то альтернативные вертикальные столбцы будут иметь заливку. Методы доступа:

```
public final void setAlternativeColumnFillVisible(boolean value)
public final boolean isAlternativeColumnFillVisible()
public final BooleanProperty
 alternativeColumnFillVisibleProperty()
```

- ❑ **data** — список с сериями данных для диаграммы (см. *разд. 17.3.4*). Методы доступа:

```
public final void setData(
 ObservableList<XYChart.Series<X, Y>> value)
public final ObservableList<XYChart.Series<X, Y>> getData()
public final ObjectProperty<ObservableList<XYChart.Series<X, Y>>>
 dataProperty()
```

### 17.3.4. Класс *XYChart.Series<X, Y>*: серия данных

Класс *XYChart.Series<X, Y>* описывает именованную серию данных. Конструкторы класса:

```
Series()
Series(ObservableList<Data<X, Y>> data)
Series(String name, ObservableList<Data<X, Y>> data)
```

Параметр `name` задает название серии данных, а параметр `data` — список с данными.

Класс *XYChart.Series<X, Y>* содержит следующие свойства:

- ❑ **name** — название серии данных. Методы доступа:

```
public final void setName(String value)
public final String getName()
public final StringProperty nameProperty()
```

- ❑ **data** — список с данными. Методы доступа:

```
public final void setData(
 ObservableList<XYChart.Data<X, Y>> value)
public final ObservableList<XYChart.Data<X, Y>> getData()
public final ObjectProperty<ObservableList<XYChart.Data<X, Y>>>
 dataProperty()
```

- ❑ `node` — ссылка на узел (может быть установлена только после добавления списка данных в объект графика) или значение `null`. Методы доступа:

```
public final void setNode(Node value)
public final Node getNode()
public final ObjectProperty<Node> nodeProperty()
```

- ❑ `chart` — содержит ссылку на объект графика или значение `null`. Методы доступа:

```
public final XYChart<X, Y> getChart()
public final ReadOnlyObjectProperty<XYChart<X, Y>>
 chartProperty()
```

Пример:

```
XYChart.Series<Number, Number> series = new XYChart.Series<>();
series.setName("Name 1");
series.getData().add(new XYChart.Data<>(10, 10));
series.getData().add(new XYChart.Data<>(30, 50));
series.getData().add(new XYChart.Data<>(40, 100));
series.getData().add(new XYChart.Data<>(50, 90));
series.getData().add(new XYChart.Data<>(60, 150));
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH17_01_Chart`, приведенном в каталоге `CH17_charts` сопровождающего книгу электронного архива (см. *приложение*).

## 17.3.5. Класс `XYChart.Data<X, Y>`: данные

Класс `XYChart.Data<X, Y>` описывает данные. Конструкторы класса:

```
XYChart.Data()
XYChart.Data(X xValue, Y yValue)
XYChart.Data(X xValue, Y yValue, Object extraValue)
```

Параметр `xValue` задает значение по оси `X`, параметр `yValue` — значение по оси `Y`, а параметр `extraValue` — дополнительные данные — например, радиус пузырька при использовании класса `BubbleChart` (см. *разд. 17.7*). Пример:

```
XYChart.Data<Number, Number> data = new XYChart.Data<>(70, 50);
```

Свойства класса `XYChart.Data<X, Y>`:

- ❑ `XValue` — значение по оси `X`. Методы доступа:

```
public final void setXValue(X value)
public final X getXValue()
public final ObjectProperty<X> XValueProperty()
```

- ❑ `YValue` — значение по оси `Y`. Методы доступа:

```
public final void setYValue(Y value)
public final Y getYValue()
public final ObjectProperty<Y> YValueProperty()
```



❑ `extraValue` — дополнительные данные. Методы доступа:

```
public final void setExtraValue(Object value)
public final Object getExtraValue()
public final ObjectProperty<Object> extraValueProperty()
```

❑ `node` — ссылка на узел, представляющий данные. Методы доступа:

```
public final void setNode(Node value)
public final Node getNode()
public final ObjectProperty<Node> nodeProperty()
```

Пример добавления всплывающей подсказки для первых данных в серии (подсказку можно добавлять только после прикрепления серии данных к объекту диаграммы):

```
// chart.getData().add(series);
Tooltip.install(series.getData().get(0).getNode(),
 new Tooltip("Data 1"));
```

Пример указания собственного узла вместо используемого по умолчанию:

```
XYChart.Data<Number, Number> data = new XYChart.Data<>(70, 50);
data.setNode(new ImageView(createImage(10, 10, Color.RED)));
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH17_01_Chart`, приведенном в каталоге `CH17_charts` сопровождающего книгу электронного архива (см. приложение).

## 17.3.6. Класс *CategoryAxis*: ось диаграммы со строковыми значениями

В предыдущих разделах мы научились создавать оси с числовыми значениями. Однако часто бывает нужно отображать данные на одной оси в виде числа, а на другой — в виде строки (например, график доходов компании по месяцам). Создать ось диаграммы со строковыми значениями позволяет класс `CategoryAxis`. Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.chart.CategoryAxis;
```

Иерархия наследования:

`Object` — `Node` — `Parent` — `Region` — `Axis<String>` — `CategoryAxis`

Конструкторы класса:

```
CategoryAxis()
CategoryAxis(ObservableList<String> categories)
```

В параметре `categories` указывается список всех категорий — например, список с названиями всех месяцев:

```
ObservableList<String> categories =
 FXCollections.<String>observableArrayList();
```

```
categories.addAll("Январь", "Февраль", "Март", "Апрель", "Май",
 "Июнь", "Июль", "Август", "Сентябрь", "Октябрь",
 "Ноябрь", "Декабрь");
CategoryAxis xAxis = new CategoryAxis(categories);
```

Управлять списком категорий после создания объекта позволяют методы: `setCategories()` и `getCategories()`. Форматы методов:

```
public final void setCategories(ObservableList<String> value)
public final ObservableList<String> getCategories()
```

Класс `CategoryAxis` наследует все свойства и методы из класса `Axis<T>` (см. *разд. 17.3.1*) и добавляет следующие основные свойства:

❑ `startMargin` — расстояние между осью и первой категорией. Методы доступа:

```
public final void setStartMargin(double value)
public final double getStartMargin()
public final DoubleProperty startMarginProperty()
```

❑ `endMargin` — расстояние между последней категорией и концом оси. Методы доступа:

```
public final void setEndMargin(double value)
public final double getEndMargin()
public final DoubleProperty endMarginProperty()
```

❑ `gapStartAndEnd` — если свойство имеет значение `true`, то точка данных будет на одной линии с риской категории на оси. Методы доступа:

```
public final void setGapStartAndEnd(boolean value)
public final boolean isGapStartAndEnd()
public final BooleanProperty gapStartAndEndProperty()
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH17_01_Chart`, приведенном в каталоге `CH17_charts` сопровождающего книгу электронного архива (см. *приложение*).

## 17.4. Класс `LineChart<X, Y>`: линейный график

Класс `LineChart<X, Y>` реализует линейный график, на котором точки соединяются прямыми линиями (рис. 17.2). Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.chart.LineChart;
```

Иерархия наследования:

```
Object — Node — Parent — Region — Chart — XYChart<X, Y> — LineChart<X, Y>
```

Конструкторы класса:

```
LineChart(Axis<X> xAxis, Axis<Y> yAxis)
LineChart(Axis<X> xAxis, Axis<Y> yAxis,
 ObservableList<XYChart.Series<X, Y>> data)
```

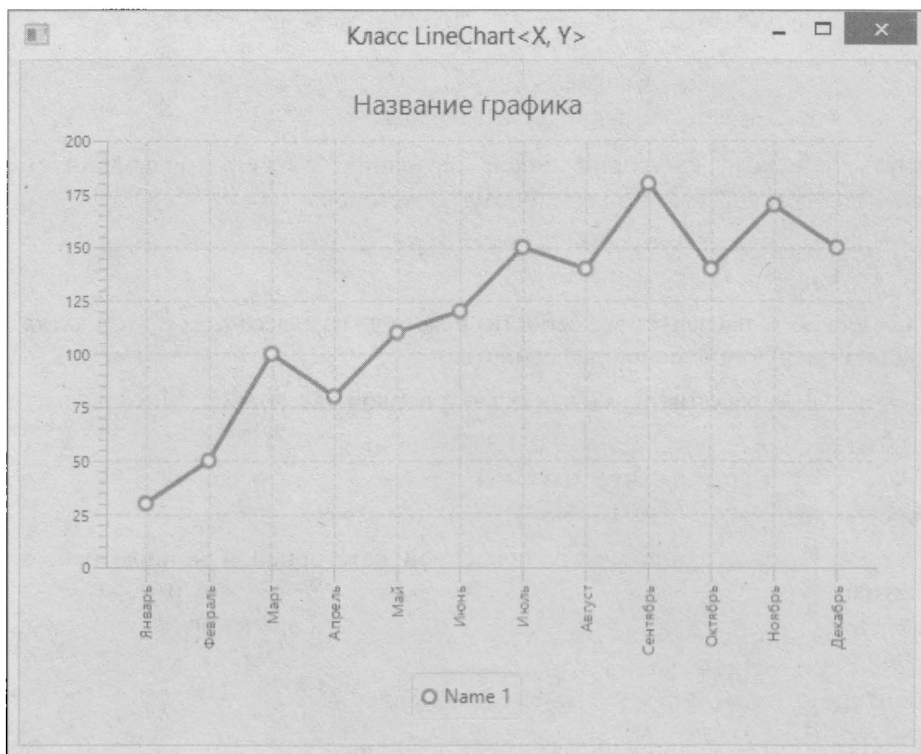


Рис. 17.2. Класс `LineChart<X, Y>`: линейный график

Параметр `xAxis` задает объект для оси X, параметр `yAxis` — объект для оси Y, а параметр `data` — список с отображаемыми данными.

Пример создания линейного графика приведен в листинге 17.2.

#### Листинг 17.2. Класс `LineChart<X, Y>`

```
ObservableList<String> categories =
 FXCollections.<String>observableArrayList();
categories.addAll("Январь", "Февраль", "Март", "Апрель", "Май",
 "Июнь", "Июль", "Август", "Сентябрь", "Октябрь",
 "Ноябрь", "Декабрь");

CategoryAxis xAxis = new CategoryAxis(categories);
xAxis.setTickLabelRotation(-90.0);
NumberAxis yAxis = new NumberAxis();

LineChart<String, Number> chart = new LineChart<>(xAxis, yAxis);
chart.setTitle("Название графика");

XYChart.Series<String, Number> series = new XYChart.Series<>();
series.setName("Name 1");
```

```

series.getData().add(new XYChart.Data<>("Январь", 30));
series.getData().add(new XYChart.Data<>("Февраль", 50));
series.getData().add(new XYChart.Data<>("Март", 100));
series.getData().add(new XYChart.Data<>("Апрель", 80));
series.getData().add(new XYChart.Data<>("Май", 110));
series.getData().add(new XYChart.Data<>("Июнь", 120));
series.getData().add(new XYChart.Data<>("Июль", 150));
series.getData().add(new XYChart.Data<>("Август", 140));
series.getData().add(new XYChart.Data<>("Сентябрь", 180));
series.getData().add(new XYChart.Data<>("Октябрь", 140));
series.getData().add(new XYChart.Data<>("Ноябрь", 170));
series.getData().add(new XYChart.Data<>("Декабрь", 150));
chart.getData().add(series);

```

```

VBox pane = new VBox(10.0);
pane.getChildren().add(chart);

```

Если нужно показывать только линии, то можно отключить отображение точек. Для этого свойству `createSymbols` следует присвоить значение `false`. Методы доступа:

```

public final void setCreateSymbols(boolean value)
public final boolean getCreateSymbols()
public final BooleanProperty createSymbolsProperty()

```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH17_01_Chart`, приведенном в каталоге `CH17_charts` сопровождающего книгу электронного архива (см. *приложение*).

## 17.5. Класс *BarChart<X, Y>*: гистограмма

Класс `BarChart<X, Y>` создает гистограмму (рис. 17.3), в которой количество данных в категории определяет высоту столбца (для вертикальной гистограммы) или его длину (для горизонтальной гистограммы). Инструкция импорта:

```

// Модуль javafx.controls
import javafx.scene.chart.BarChart;

```

### Иерархия наследования:

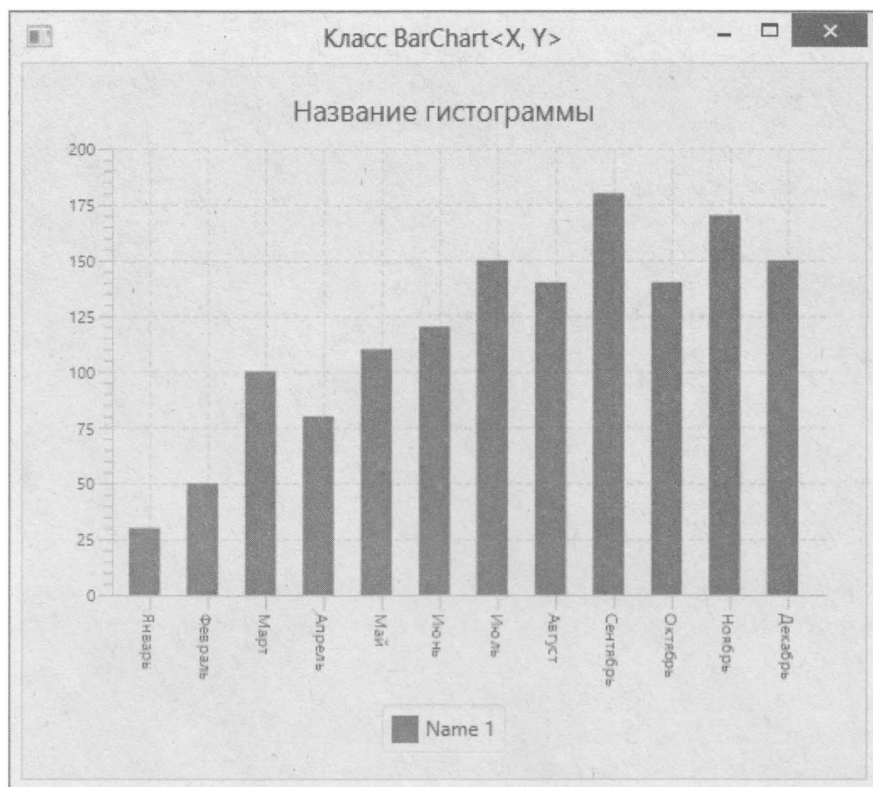
`Object` – `Node` – `Parent` – `Region` – `Chart` – `XYChart<X, Y>` – `BarChart<X, Y>`

### Конструкторы класса:

```

BarChart(Axis<X> xAxis, Axis<Y> yAxis)
BarChart(Axis<X> xAxis, Axis<Y> yAxis,
 ObservableList<XYChart.Series<X, Y>> data)
BarChart(Axis<X> xAxis, Axis<Y> yAxis,
 ObservableList<XYChart.Series<X, Y>> data, double categoryGap)

```



**Рис. 17.3.** Класс `BarChart<X, Y>`: гистограмма с отображением данных разных серий в виде отдельных столбцов

Если в параметре `xAxis` указан объект класса `CategoryAxis`, то в параметре `yAxis` должен быть объект класса `NumberAxis` (вертикальная гистограмма). Если в параметре `xAxis` указан объект класса `NumberAxis`, то в параметре `yAxis` должен быть объект класса `CategoryAxis` (горизонтальная гистограмма). Параметр `data` задает список с отображаемыми данными, а параметр `categoryGap` — расстояние между категориями гистограммы.

Класс `BarChart<X, Y>` наследует все свойства из класса `XYChart<X, Y>` (см. *разд. 17.3.3*) и добавляет следующие свойства:

- ☐ `categoryGap` — расстояние между категориями гистограммы. Методы доступа:
 

```
public final void setCategoryGap(double value)
public final double getCategoryGap()
public final DoubleProperty categoryGapProperty()
```
- ☐ `barGap` — расстояние между столбцами гистограммы внутри одной категории. Методы доступа:
 

```
public final void setBarGap(double value)
public final double getBarGap()
public final DoubleProperty barGapProperty()
```

Пример создания гистограммы приведен в листинге 17.3.

### Листинг 17.3. Класс `BarChart<X, Y>`

```
CategoryAxis xAxis = new CategoryAxis();
NumberAxis yAxis = new NumberAxis();

BarChart<String, Number> chart = new BarChart<>(xAxis, yAxis);
chart.setTitle("Название гистограммы");

XYChart.Series<String, Number> series = new XYChart.Series<>();
series.setName("Name 1");
series.getData().add(new XYChart.Data<>("Январь", 30));
series.getData().add(new XYChart.Data<>("Февраль", 50));
series.getData().add(new XYChart.Data<>("Март", 100));
series.getData().add(new XYChart.Data<>("Апрель", 80));
series.getData().add(new XYChart.Data<>("Май", 110));
series.getData().add(new XYChart.Data<>("Июнь", 120));
series.getData().add(new XYChart.Data<>("Июль", 150));
series.getData().add(new XYChart.Data<>("Август", 140));
series.getData().add(new XYChart.Data<>("Сентябрь", 180));
series.getData().add(new XYChart.Data<>("Октябрь", 140));
series.getData().add(new XYChart.Data<>("Ноябрь", 170));
series.getData().add(new XYChart.Data<>("Декабрь", 150));
chart.getData().add(series);

VBox pane = new VBox(10.0);
pane.getChildren().add(chart);
```

#### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH17_01_Chart`, приведенном в каталоге `CH17_charts` сопровождающего книгу электронного архива (см. приложение).

## 17.6. Класс `StackedBarChart<X, Y>`: гистограмма

Класс `StackedBarChart<X, Y>` создает гистограмму (рис. 17.4), в которой количество данных в категории определяет высоту столбца (для вертикальной гистограммы) или его длину (для горизонтальной гистограммы). При использовании класса `BarChart<X, Y>` данные разных серий в одной категории отображаются в виде отдельных столбцов, тогда как при использовании класса `StackedBarChart<X, Y>` данные разных серий накладываются друг на друга. Итоговая высота столбца будет равна сумме данных разных серий в одной категории. Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.chart.StackedBarChart;
```

## Иерархия наследования:

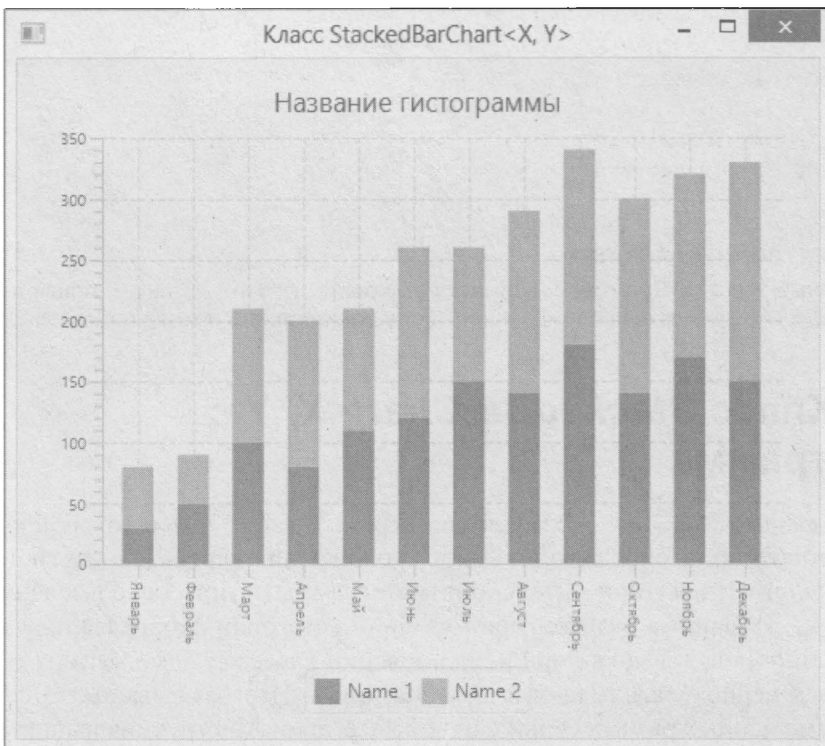
Object — Node — Parent — Region — Chart — XYChart<X, Y> — StackedBarChart<X, Y>

## Конструкторы класса:

```
StackedBarChart(Axis<X> xAxis, Axis<Y> yAxis)
StackedBarChart(Axis<X> xAxis, Axis<Y> yAxis,
 ObservableList<XYChart.Series<X, Y>> data)
StackedBarChart(Axis<X> xAxis, Axis<Y> yAxis,
 ObservableList<XYChart.Series<X, Y>> data,
 double categoryGap)
```

Если в параметре `xAxis` указан объект класса `CategoryAxis`, то в параметре `yAxis` должен быть объект класса `NumberAxis` (вертикальная гистограмма). Если в параметре `xAxis` указан объект класса `NumberAxis`, то в параметре `yAxis` должен быть объект класса `CategoryAxis` (горизонтальная гистограмма). Параметр `data` задает список с отображаемыми данными, а параметр `categoryGap` — расстояние между категориями гистограммы. Управлять этим расстоянием после создания объекта позволяет свойство `categoryGap`. Методы доступа:

```
public final void setCategoryGap(double value)
public final double getCategoryGap()
public final DoubleProperty categoryGapProperty()
```



**Рис. 17.4.** Класс `StackedBarChart<X, Y>`: гистограмма с наложением друг на друга данных разных серий

**ЭЛЕКТРОННЫЙ АРХИВ**

Полный текст программы содержится в проекте `FX_CH17_01_Chart`, приведенном в каталоге `CH17_charts` сопровождающего книгу электронного архива (см. приложение).

## 17.7. Класс *BubbleChart<X, Y>*: пузырьковая диаграмма

Класс `BubbleChart<X, Y>` реализует диаграмму, в которой точки данных отображаются в виде пузырьков (рис. 17.5). Радиус пузырька задается с помощью параметра `extraValue` в конструкторе класса `XYChart.Data<X, Y>`:

```
XYChart.Data(X xValue, Y yValue, Object extraValue)
```

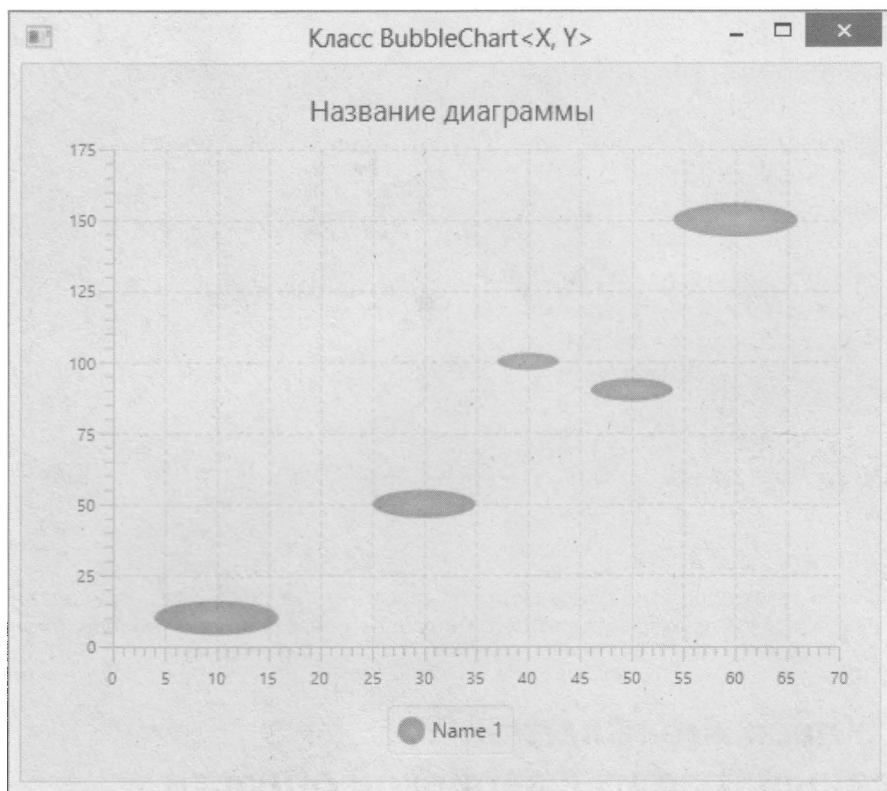


Рис. 17.5. Класс `BubbleChart<X, Y>`: пузырьковая диаграмма

### Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.chart.BubbleChart;
```

### Иерархия наследования:

```
Object — Node — Parent — Region — Chart — XYChart<X, Y> — BubbleChart<X, Y>
```



### Конструкторы класса:

```
BubbleChart(Axis<X> xAxis, Axis<Y> yAxis)
BubbleChart(Axis<X> xAxis, Axis<Y> yAxis,
 ObservableList<XYChart.Series<X, Y>> data)
```

В параметрах `xAxis` и `yAxis` указываются объекты осей с числовыми значениями (тип `NumberAxis`). Параметр `data` задает список с отображаемыми данными.

Пример создания пузырьковой диаграммы приведен в листинге 17.4.

#### Листинг 17.4. Класс `BubbleChart<X, Y>`

```
NumberAxis xAxis = new NumberAxis();
NumberAxis yAxis = new NumberAxis();

BubbleChart<Number, Number> chart = new BubbleChart<>(xAxis, yAxis);
chart.setTitle("Название диаграммы");

XYChart.Series<Number, Number> series = new XYChart.Series<>();
series.setName("Name 1");
series.getData().add(new XYChart.Data<>(10, 10, 6));
series.getData().add(new XYChart.Data<>(30, 50, 5));
series.getData().add(new XYChart.Data<>(40, 100, 3));
series.getData().add(new XYChart.Data<>(50, 90, 4));
series.getData().add(new XYChart.Data<>(60, 150, 6));
chart.getData().add(series);

VBox pane = new VBox(10.0);
pane.getChildren().add(chart);
```

#### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH17_01_Chart`, приведенном в каталоге `CH17_charts` сопровождающего книгу электронного архива (см. приложение).

## 17.8. Класс `AreaChart<X, Y>`: линейный график с заливкой области

Класс `AreaChart<X, Y>` реализует линейный график с заливкой области под линиями (рис. 17.6). Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.chart.AreaChart;
```

#### Иерархия наследования:

```
Object — Node — Parent — Region — Chart — XYChart<X, Y> — AreaChart<X, Y>
```

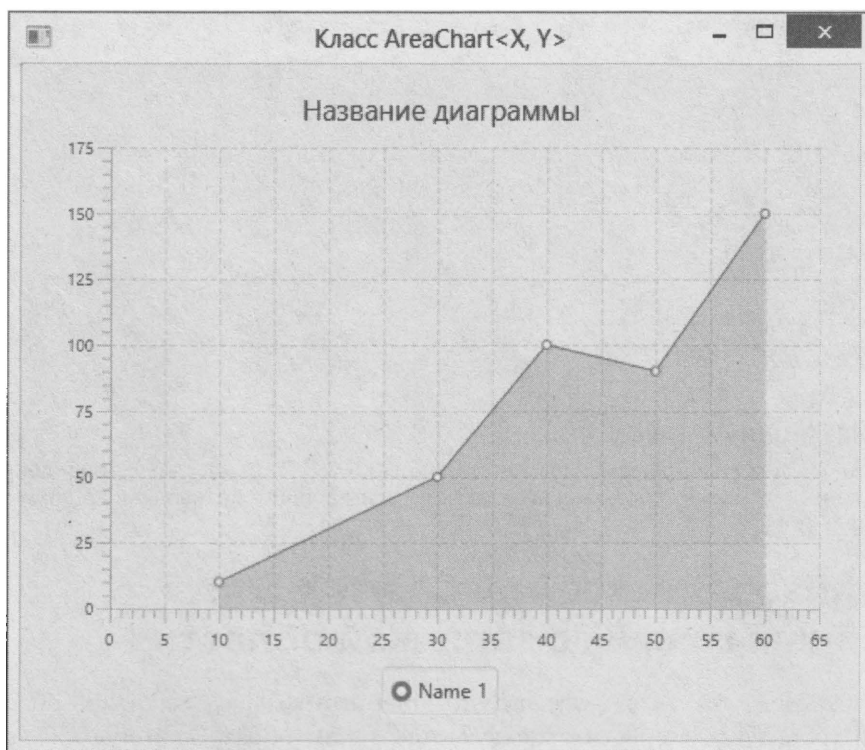


Рис. 17.6. Класс `AreaChart<X, Y>`: линейный график с заливкой области под линией

### Конструкторы класса:

```
AreaChart(Axis<X> xAxis, Axis<Y> yAxis)
AreaChart(Axis<X> xAxis, Axis<Y> yAxis,
 ObservableList<XYChart.Series<X, Y>> data)
```

Если нужно показывать только линии, то можно отключить отображение точек. Для этого свойству `createSymbols` следует присвоить значение `false`. Методы доступа:

```
public final void setCreateSymbols(boolean value)
public final boolean getCreateSymbols()
public final BooleanProperty createSymbolsProperty()
```

Пример создания графика приведен в листинге 17.5.

### Листинг 17.5. Класс `AreaChart<X, Y>`

```
NumberAxis xAxis = new NumberAxis();
NumberAxis yAxis = new NumberAxis();

AreaChart<Number, Number> chart = new AreaChart<>(xAxis, yAxis);
chart.setTitle("Название диаграммы");
```

```

XYChart.Series<Number, Number> series = new XYChart.Series<>();
series.setName("Name 1");
series.getData().add(new XYChart.Data<>(10, 10));
series.getData().add(new XYChart.Data<>(30, 50));
series.getData().add(new XYChart.Data<>(40, 100));
series.getData().add(new XYChart.Data<>(50, 90));
series.getData().add(new XYChart.Data<>(60, 150));
chart.getData().add(series);

```

```

VBox pane = new VBox(10.0);
pane.getChildren().add(chart);

```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH17_01_Chart`, приведенном в каталоге `CH17_charts` сопровождающего книгу электронного архива (см. *приложение*).

## 17.9. Класс *StackedAreaChart<X, Y>*: линейный график с заливкой области

Класс `StackedAreaChart<X, Y>` реализует линейный график с заливкой области под линиями (рис. 17.7). Области складываются таким образом, чтобы каждая серия данных примыкала, но не перекрывала предыдущую серию данных. Инструкция импорта:

```

// Модуль javafx.controls
import javafx.scene.chart.StackedAreaChart;

```

Иерархия наследования:

```

Object — Node — Parent — Region — Chart — XYChart<X, Y> —
 StackedAreaChart<X, Y>

```

Конструкторы класса:

```

StackedAreaChart(Axis<X> xAxis, Axis<Y> yAxis)
StackedAreaChart(Axis<X> xAxis, Axis<Y> yAxis,
 ObservableList<XYChart.Series<X, Y>> data)

```

Если нужно показывать только линии, то можно отключить отображение точек. Для этого свойству `createSymbols` следует присвоить значение `false`. Методы доступа:

```

public final void setCreateSymbols(boolean value)
public final boolean getCreateSymbols()
public final BooleanProperty createSymbolsProperty()

```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH17_01_Chart`, приведенном в каталоге `CH17_charts` сопровождающего книгу электронного архива (см. *приложение*).

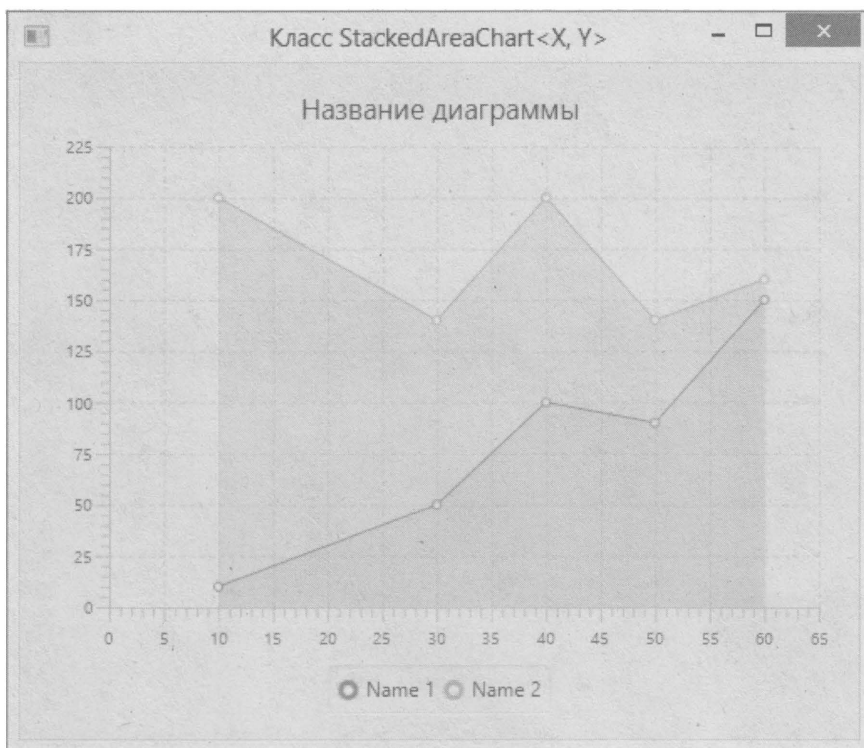


Рис. 17.7. Класс `StackedAreaChart<X, Y>`:  
линейный график с заливкой области под линиями с примыканием областей

## 17.10. Класс `ScatterChart<X, Y>`: диаграмма в виде символов

Класс `ScatterChart<X, Y>` реализует диаграмму, в которой точки данных отображаются в виде различных символов (рис. 17.8). Инструкция импорта:

```
// Модуль javafx.controls
import javafx.scene.chart.ScatterChart;
```

Иерархия наследования:

Object — Node — Parent — Region — Chart — XYChart<X, Y> — ScatterChart<X, Y>

Конструкторы класса:

```
ScatterChart(Axis<X> xAxis, Axis<Y> yAxis)
ScatterChart(Axis<X> xAxis, Axis<Y> yAxis,
 ObservableList<XYChart.Series<X, Y>> data)
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH17_01_Chart`, приведенном в каталоге `CH17_charts` сопровождающего книгу электронного архива (см. приложение).

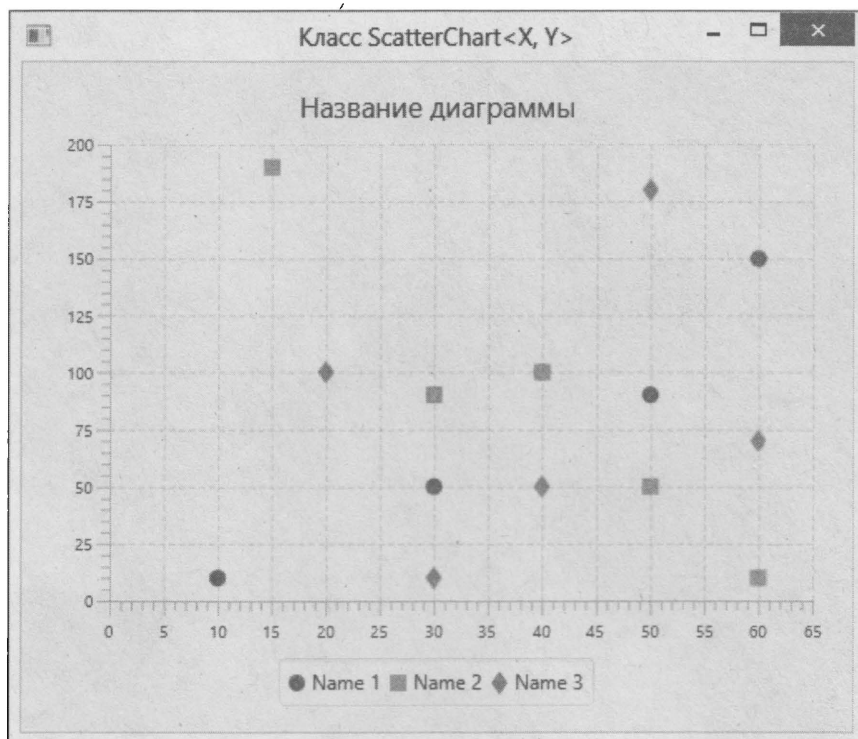


Рис. 17.8. Класс ScatterChart<X, Y>: диаграмма в виде символов

# ГЛАВА 18



## Аудио и видео

Библиотека JavaFX позволяет воспроизводить аудио и видео. Для этого нужно выполнить следующие шаги:

1. Создать объект `Media` с ресурсом.
2. Создать объект `MediaPlayer` и передать ему объект ресурса.
3. Создать объект `MediaView` и передать ему объект плеера (для аудио этот шаг можно пропустить).
4. Добавить объект `MediaView` в граф сцены (для аудио этот шаг можно пропустить).
5. Настроить параметры воспроизведения и вызвать метод `play()` объекта плеера.

Для простоты воспроизведения аудио можно также воспользоваться классом `AudioClip`, но следует помнить, что он всегда загружает ресурс целиком в память, а не читает его по частям в буфер.

### 18.1. Класс *AudioClip*: воспроизведение аудио

Класс `AudioClip` позволяет загружать аудиофайлы по протоколам: HTTP, HTTPS, FILE или JAR и воспроизводить их. Инструкция импорта:

```
// Модуль javafx.media
import javafx.scene.media.AudioClip;
```

Иерархия наследования:

`Object` — `AudioClip`

Конструктор класса:

```
AudioClip(String source)
```

В параметре `source` указывается интернет-адрес (URL) с протоколом (HTTP, HTTPS, FILE или JAR) и полным путем к файлу. Если в процессе загрузки возникли ошибки, то генерируются исключения классов `IllegalArgumentException` или

`MediaException`. Пример загрузки файла с локального диска (файл расположен рядом с каталогом `src`) с обработкой исключений:

```
AudioClip clip;
try {
 URI path = Paths.get("audio.wav").toAbsolutePath().toUri();
 clip = new AudioClip(path.toString());
} catch (Exception e) {
 System.out.println("Не удалось найти или загрузить файл");
 return;
}
```

Методы класса `AudioClip`:

❑ `getSource()` — возвращает интернет-адрес (URL) загруженного файла. Формат метода:

```
public String getSource()
```

❑ `play()` — начинает воспроизведение аудио с начала. Форматы метода:

```
public void play()
public void play(double volume)
public void play(double volume, double balance, double rate,
 double pan, int priority)
```

Начнем воспроизведение с настройками по умолчанию после нажатия кнопки:

```
Button btnPlay = new Button("Воспроизвести");
btnPlay.setOnAction(event -> {
 if (!clip.isPlaying()) clip.play();
});
```

❑ `isPlaying()` — возвращает значение `true`, если аудио воспроизводится, и `false` — в противном случае. Формат метода:

```
public boolean isPlaying()
```

❑ `stop()` — останавливает воспроизведение аудио. Формат метода:

```
public void stop()
```

Остановим воспроизведение после нажатия кнопки:

```
Button btnStop = new Button("Остановить");
btnStop.setOnAction(event -> {
 if (clip.isPlaying()) clip.stop();
});
```

Свойства класса `AudioClip` (настройка свойств должна выполняться до вызова метода `play()`, изменение свойств во время выполнения не отразится на воспроизводимом клипе):

❑ `volume` — громкость. В качестве значения указывается вещественное число от 0.0 (без звука) до 1.0 (полная громкость). Значение по умолчанию: 1.0. Методы доступа:

```
public final void setVolume(double value)
public final double getVolume()
public DoubleProperty volumeProperty()
```

- **balance** — баланс. В качестве значения указывается вещественное число от  $-1.0$  (левый канал) до  $1.0$  (правый канал). Значение по умолчанию:  $0.0$ . Методы доступа:

```
public void setBalance(double balance)
public double getBalance()
public DoubleProperty balanceProperty()
```

- **pan** — «центр» клипа. В отличие от свойства **balance**, выполняется смешивание двух каналов без потери данных. В качестве значения указывается вещественное число от  $-1.0$  (левый канал) до  $1.0$  (правый канал). Значение по умолчанию:  $0.0$ . Методы доступа:

```
public void setPan(double pan)
public double getPan()
public DoubleProperty panProperty()
```

- **rate** — скорость воспроизведения. В качестве значения указывается вещественное число от  $0.125$  до  $8.0$ . Значение по умолчанию:  $1.0$ . Методы доступа:

```
public void setRate(double rate)
public double getRate()
public DoubleProperty rateProperty()
```

- **cycleCount** — количество повторов воспроизведения. Если нужно воспроизводить бесконечное количество раз, то в качестве значения следует указать статическую константу `AudioClip.INDEFINITE`. Значение по умолчанию:  $1$ . Методы доступа:

```
public static final int INDEFINITE = -1;
public void setCycleCount(int count)
public int getCycleCount()
public IntegerProperty cycleCountProperty()
```

- **priority** — приоритет клипа относительно других клипов. Значение по умолчанию:  $0$ . Методы доступа:

```
public void setPriority(int priority)
public int getPriority()
public IntegerProperty priorityProperty()
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH18_01_media`, приведенном в каталоге `CH18_media` сопровождающего книгу электронного архива (см. приложение).



## 18.2. Класс *Media*

Класс `Media` описывает ресурс с аудио и видео, который может быть загружен по протоколам: HTTP, HTTPS, FILE или JAR. Инструкция импорта:

```
// Модуль javafx.media
import javafx.scene.media.Media;
```

Иерархия наследования:

Object — Media

Конструктор класса:

```
Media(String source)
```

В параметре `source` указывается интернет-адрес (URL) с протоколом (HTTP, HTTPS, FILE или JAR) и полным путем к файлу. Если в процессе загрузки возникли ошибки, то генерируются исключения классов `IllegalArgumentException` или `MediaException`. Пример загрузки файла формате MP4 с локального диска (файл расположен рядом с каталогом `src`) с обработкой исключений:

```
Media video;
try {
 URI path = Paths.get("video.mp4").toAbsolutePath().toUri();
 video = new Media(path.toString());
} catch (Exception e) {
 System.out.println("Не удалось найти или загрузить файл");
 return;
}
```

Методы класса `Media`:

❑ `getSource()` — возвращает интернет-адрес (URL) загруженного файла. Формат метода:

```
public String getSource()
```

❑ `getMetadata()` — возвращает словарь с метаданными. Формат метода:

```
public final ObservableMap<String, Object> getMetadata()
```

❑ `getMarkers()` — возвращает словарь с маркерами. Формат метода:

```
public final ObservableMap<String, Duration> getMarkers()
```

❑ `getTracks()` — возвращает список с дорожками. Формат метода:

```
public final ObservableList<Track> getTracks()
```

Свойства класса `Media`:

❑ `width` — ширина в пикселах или значение 0. Методы доступа:

```
public final int getWidth()
public ReadOnlyIntegerProperty widthProperty()
```

- ❑ **height** — высота в пикселах или значение 0. Методы доступа:

```
public final int getHeight()
public ReadOnlyIntegerProperty heightProperty()
```

- ❑ **duration** — продолжительность, или значение UNKNOWN, или значение INDEFINITE. Методы доступа:

```
public final Duration getDuration()
public ReadOnlyObjectProperty<Duration> durationProperty()
```

- ❑ **error** — объект исключения или значение null (при отсутствии ошибок). Методы доступа:

```
public final MediaException getError()
public ReadOnlyObjectProperty<MediaException> errorProperty()
```

- ❑ **onError** — позволяет назначить обработчик для события изменения свойства error. Методы доступа:

```
public final void setOnError(Runnable value)
public final Runnable getOnError()
public ObjectProperty<Runnable> onErrorProperty()
```

Пример назначения обработчика:

```
video.setOnError(() -> {
 System.out.println("OnError");
});
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH18_01_media`, приведенном в каталоге `CH18_media` сопровождающего книгу электронного архива (см. *приложение*).

## 18.3. Класс *MediaPlayer*

Класс `MediaPlayer` позволяет воспроизводить аудио и видео, которые представлены с помощью класса `Media`. Инструкция импорта:

```
// Модуль javafx.media
import javafx.scene.media.MediaPlayer;
```

Иерархия наследования:

```
Object — Node — MediaPlayer
```

Конструктор класса:

```
MediaPlayer(Media media)
```

Если параметр `media` имеет значение `null`, то генерируется исключение `NullPointerException`. Если загрузка выполняется синхронно, то при возникновении ошибки генерируется исключение `MediaException`, а если асинхронно, то свойству `error` будет присвоен объект исключения и вызван обработчик, назначенный с помощью свойства `onError`. Пример:

```
// Media video;
MediaPlayer playerVideo = new MediaPlayer(video);
```

### 18.3.1. Обработка изменения статуса и ошибок

Так как загрузка может производиться как синхронно, так и асинхронно, после создания объекта плеера нужно дождаться статуса `READY`, который говорит, что плеер готов к работе. Отследить изменение статуса позволяет свойство `status`. Методы доступа:

```
public final MediaPlayer.Status getStatus()
public ReadOnlyObjectProperty<MediaPlayer.Status> statusProperty()
```

Перечисление `MediaPlayer.Status` содержит следующие константы:

- ☐ `UNKNOWN` — состояние плеера после создания;
- ☐ `READY` — плеер готов к работе;
- ☐ `PAUSED` — воспроизведение временно приостановлено;
- ☐ `PLAYING` — идет воспроизведение;
- ☐ `STOPPED` — воспроизведение остановлено;
- ☐ `STALLED` — в буфере недостаточно данных для продолжения воспроизведения;
- ☐ `HALTED` — произошла критическая ошибка, и плеер больше использовать нельзя;
- ☐ `DISPOSED` — все ресурсы освобождены, и плеер больше использовать нельзя.

Запустим воспроизведение видео после установки статуса `READY`:

```
playerVideo.statusProperty().addListener((obj, oldValue, newValue) -> {
 if (newValue == MediaPlayer.Status.READY) {
 playerVideo.play();
 }
});
```

Назначить обработчики позволяют следующие свойства:

- ☐ `onReady` — обработчик для статуса `READY`. Методы доступа:

```
public final void setOnReady(Runnable value)
public final Runnable getOnReady()
public ObjectProperty<Runnable> onReadyProperty()
```

- ☐ `onPaused` — обработчик для статуса `PAUSED`. Методы доступа:

```
public final void setOnPaused(Runnable value)
public final Runnable getOnPaused()
public ObjectProperty<Runnable> onPausedProperty()
```

- ☐ `onPlaying` — обработчик для статуса `PLAYING`. Методы доступа:

```
public final void setOnPlaying(Runnable value)
public final Runnable getOnPlaying()
public ObjectProperty<Runnable> onPlayingProperty()
```

- ❑ `onStopped` — обработчик для статуса `STOPPED`. Методы доступа:

```
public final void setOnStopped(Runnable value)
public final Runnable getOnStopped()
public ObjectProperty<Runnable> onStoppedProperty()
```

- ❑ `onStalled` — обработчик для статуса `STALLED`. Методы доступа:

```
public final void setOnStalled(Runnable value)
public final Runnable getOnStalled()
public ObjectProperty<Runnable> onStalledProperty()
```

- ❑ `onHalted` — обработчик для статуса `HALTED`. Методы доступа:

```
public final void setOnHalted(Runnable value)
public final Runnable getOnHalted()
public ObjectProperty<Runnable> onHaltedProperty()
```

- ❑ `error` — содержит объект исключения или значение `null` (при отсутствии ошибок). Методы доступа:

```
public final MediaException getError()
public ReadOnlyObjectProperty<MediaException> errorProperty()
```

- ❑ `onError` — обработчик изменения значения свойства `error`. Методы доступа:

```
public final void setOnError(Runnable value)
public final Runnable getOnError()
public ObjectProperty<Runnable> onErrorProperty()
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH18_01_media`, приведенном в каталоге `CH18_media` сопровождающего книгу электронного архива (см. приложение).

## 18.3.2. Управление воспроизведением

Управлять воспроизведением позволяют следующие методы из класса `MediaPlayer`:

- ❑ `play()` — запускает воспроизведение. Если воспроизведение было временно приостановлено, то оно начнется с того же самого места, в противном случае — со значения свойства `startTime`. Формат метода:

```
public void play()
```

- ❑ `pause()` — временно приостанавливает воспроизведение. Формат метода:

```
public void pause()
```

- ❑ `stop()` — останавливает воспроизведение. Формат метода:

```
public void stop()
```

- ❑ `seek()` — перемещает указатель текущей позиции в указанную позицию. Формат метода:

```
public void seek(Duration seekTime)
```

- ❑ `dispose()` — освобождает все ресурсы и устанавливает для плеера статус `DISPOSED`. Формат метода:

```
public void dispose()
```

#### Свойства класса `MediaPlayer`:

- ❑ `autoPlay` — если свойство имеет значение `true`, то воспроизведение начнется сразу, как только плеер будет готов к работе. Методы доступа:

```
public final void setAutoPlay(boolean value)
public final boolean isAutoPlay()
public BooleanProperty autoPlayProperty()
```

- ❑ `startTime` — начальное время, с которого начинается воспроизведение. Значение по умолчанию: `Duration.ZERO`. Методы доступа:

```
public final void setStartTime(Duration value)
public final Duration getStartTime()
public ObjectProperty<Duration> startTimeProperty()
```

- ❑ `stopTime` — конечное время. Методы доступа:

```
public final void setStopTime(Duration value)
public final Duration getStopTime()
public ObjectProperty<Duration> stopTimeProperty()
```

- ❑ `onEndOfMedia` — задает обработчик для события достижения текущей позиции значения свойства `stopTime`. Методы доступа:

```
public final void setOnEndOfMedia(Runnable value)
public final Runnable getOnEndOfMedia()
public ObjectProperty<Runnable> onEndOfMediaProperty()
```

- ❑ `onRepeat` — задает обработчик для события достижения текущей позиции значения свойства `stopTime` и последующего повтора воспроизведения. Методы доступа:

```
public final void setOnRepeat(Runnable value)
public final Runnable getOnRepeat()
public ObjectProperty<Runnable> onRepeatProperty()
```

- ❑ `currentTime` — содержит текущую позицию. Методы доступа:

```
public final Duration getCurrentTime()
public ReadOnlyObjectProperty<Duration> currentTimeProperty()
```

- ❑ `currentCount` — содержит количество завершенных циклов воспроизведения. Методы доступа:

```
public final int getCurrentCount()
public ReadOnlyIntegerProperty currentCountProperty()
```

- ❑ `currentRate` — содержит текущую скорость воспроизведения или значение 0. Методы доступа:

```
public final double getCurrentRate()
public ReadOnlyDoubleProperty currentRateProperty()
```

- ❑ **cycleDuration** — продолжительность между **startTime** и **stopTime**. Чтобы получить общую продолжительность, следует воспользоваться свойством **duration** из класса **Media**. Методы доступа:

```
public final Duration getCycleDuration()
public ReadOnlyObjectProperty<Duration> cycleDurationProperty()
```

- ❑ **totalDuration** — общая продолжительность воспроизведения (произведение **cycleCount** и **cycleDuration**). Методы доступа:

```
public final Duration getTotalDuration()
public ReadOnlyObjectProperty<Duration> totalDurationProperty()
```

- ❑ **bufferProgressTime** — буфер воспроизведения. Методы доступа:

```
public final Duration getBufferProgressTime()
public ReadOnlyObjectProperty<Duration>
 bufferProgressTimeProperty()
```

### 18.3.3. Настройки воспроизведения аудио и видео

Управлять настройками воспроизведения аудио и видео позволяют следующие свойства из класса **MediaPlayer**:

- ❑ **volume** — громкость. В качестве значения указывается вещественное число от 0.0 (без звука) до 1.0 (полная громкость). Значение по умолчанию: 1.0. Методы доступа:

```
public final void setVolume(double value)
public final double getVolume()
public DoubleProperty volumeProperty()
```

- ❑ **mute** — если свойство имеет значение **true**, то звук будет выключен (воспроизведение при этом продолжится). Методы доступа:

```
public final void setMute(boolean value)
public final boolean isMute()
public BooleanProperty muteProperty()
```

- ❑ **balance** — баланс. В качестве значения указывается вещественное число от -1.0 (левый канал) до 1.0 (правый канал). Значение по умолчанию: 0.0. Методы доступа:

```
public final void setBalance(double value)
public final double getBalance()
public DoubleProperty balanceProperty()
```

- ❑ **rate** — скорость воспроизведения. В качестве значения указывается вещественное число от 0.0 до 8.0. Значение по умолчанию: 1.0. Методы доступа:

```
public final void setRate(double value)
public final double getRate()
public DoubleProperty rateProperty()
```

- ❑ **cycleCount** — количество повторов воспроизведения. Если нужно воспроизводить бесконечное количество раз, то в качестве значения следует указать статическую константу `MediaPlayer.INDEFINITE`. Значение по умолчанию: 1. Методы доступа:

```
public static final int INDEFINITE = -1;
public final void setCycleCount(int value)
public final int getCycleCount()
public IntegerProperty cycleCountProperty()
```

### ПРИМЕЧАНИЕ

Для работы с аудио существуют свойства: `audioSpectrumInterval`, `audioSpectrumNumBands`, `audioSpectrumThreshold` и `audioSpectrumListener`, а также метод `getAudioEqualizer()`. Подробную информацию смотрите в документации.

## 18.4. Класс *MediaView*

Класс `MediaView` описывает узел, позволяющий встроить плеер в граф сцены. Инструкция импорта:

```
// Модуль javafx.media
import javafx.scene.media.MediaView;
```

Иерархия наследования:

```
Object — Node — MediaView
```

Конструкторы класса:

```
MediaView()
MediaView(MediaPlayer mediaPlayer)
```

Первый конструктор создает объект с настройками по умолчанию, а второй позволяет сразу указать объект плеера. Пример отображения видео:

```
Media video;
try {
 URI path = Paths.get("video.mp4").toAbsolutePath().toUri();
 video = new Media(path.toString());
} catch (Exception e) {
 System.out.println("Не удалось найти или загрузить файл");
 return;
}
```

```
MediaPlayer playerVideo = new MediaPlayer(video);
playerVideo.setAutoplay(true);
```

```
MediaView viewVideo = new MediaView(playerVideo);
viewVideo.setFitWidth(500.0);
```

```
VBox pane = new VBox(10.0);
pane.getChildren().add(viewVideo);
```

**Класс `MediaView` содержит следующие свойства:**

❑ **`mediaPlayer`** — содержит объект плеера. Методы доступа:

```
public final void setMediaPlayer(MediaPlayer value)
public final MediaPlayer getMediaPlayer()
public final ObjectProperty<MediaPlayer> mediaPlayerProperty()
```

❑ **`x`** — координата левого верхнего угла по оси `x`. Методы доступа:

```
public final void setX(double value)
public final double getX()
public final DoubleProperty xProperty()
```

❑ **`y`** — координата левого верхнего угла по оси `y`. Методы доступа:

```
public final void setY(double value)
public final double getY()
public final DoubleProperty yProperty()
```

❑ **`fitWidth`** — предпочтительная ширина отображения видео. Если значение меньше или равно нулю, то используется исходная ширина видео. Значение по умолчанию: `0.0`. Методы доступа:

```
public final void setFitWidth(double value)
public final double getFitWidth()
public final DoubleProperty fitWidthProperty()
```

❑ **`fitHeight`** — предпочтительная высота отображения видео. Если значение меньше или равно нулю, то используется исходная высота видео. Значение по умолчанию: `0.0`. Методы доступа:

```
public final void setFitHeight(double value)
public final double getFitHeight()
public final DoubleProperty fitHeightProperty()
```

❑ **`preserveRatio`** — если свойство имеет значение `true`, то изменение размеров видео (чтобы вписаться в область размером `fitWidth` на `fitHeight`) будет производиться с сохранением пропорций. Значение по умолчанию: `true`. Методы доступа:

```
public final void setPreserveRatio(boolean value)
public final boolean isPreserveRatio()
public final BooleanProperty preserveRatioProperty()
```

❑ **`smooth`** — если свойство имеет значение `true`, то при изменении размеров видео будет использоваться сглаживание. Методы доступа:



```
public final void setSmooth(boolean value)
public final boolean isSmooth()
public final BooleanProperty smoothProperty()
```

- ❑ **viewport** — позволяет указать, какую область кадра видео показывать. По умолчанию свойство имеет значение `null`, что приводит к отображению всего кадра видео. Методы доступа:

```
public final void setViewport(Rectangle2D value)
public final Rectangle2D getViewport()
public final ObjectProperty<Rectangle2D> viewportProperty()
```

- ❑ **onError** — позволяет назначить обработчик, который будет вызван при ошибке. Методы доступа:

```
public final void setOnError(EventHandler<MediaErrorEvent> value)
public final EventHandler<MediaErrorEvent> getOnError()
public final ObjectProperty<EventHandler<MediaErrorEvent>>
 onErrorProperty()
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH18_01_media`, приведенном в каталоге `CH18_media` сопровождающего книгу электронного архива (см. приложение).

## 18.5. Класс *Track*: дорожка

Класс `Track` является базовым для классов, описывающих дорожки: `AudioTrack` (аудиодорожка), `VideoTrack` (видеодорожка) и `SubtitleTrack` (дорожка с субтитрами). Инструкции импорта:

```
import javafx.scene.media.Track;
import javafx.scene.media.AudioTrack;
import javafx.scene.media.VideoTrack;
import javafx.scene.media.SubtitleTrack;
```

Иерархии наследования:

```
Object — Track — AudioTrack
Object — Track — VideoTrack
Object — Track — SubtitleTrack
```

Один ресурс может содержать сразу несколько дорожек. Получить список со всеми дорожками позволяет метод `getTracks()` из класса `Media`. Формат метода:

```
public final ObservableList<Track> getTracks()
```

Пример получения списка дорожек внутри обработчика статуса готовности плеера:

```
playerVideo.setOnReady(() -> {
 ObservableList<Track> tracks = video.getTracks();
});
```

Класс `Track` содержит следующие методы:

- ❑ `getName()` — возвращает название дорожки или значение `null`. Формат метода:  

```
public final String getName()
```
- ❑ `getTrackID()` — возвращает уникальный идентификатор дорожки. Формат метода:  

```
public final long getTrackID()
```
- ❑ `getLocale()` — возвращает локаль или значение `null`. Формат метода:  

```
public final Locale getLocale()
```
- ❑ `getMetadata()` — возвращает словарь с метаданными. Формат метода:  

```
public final Map<String, Object> getMetadata()
```

Пример метаданных для аудио:

```
{name=audio/mpeg, encoding=AAC, enabled=true}
```

Пример метаданных для видео:

```
{name=video/x-h264, video width=1920, encoding=H264,
video height=1080, enabled=true}
```

Класс `VideoTrack` содержит два дополнительных метода:

- ❑ `getWidth()` — возвращает ширину видео. Формат метода:  

```
public final int getWidth()
```
- ❑ `getHeight()` — возвращает высоту видео. Формат метода:  

```
public final int getHeight()
```

### **ЭЛЕКТРОННЫЙ АРХИВ**

Полный текст программы содержится в проекте `FX_CH18_01_media`, приведенном в каталоге `CH18_media` сопровождающего книгу электронного архива (см. *приложение*).



# ГЛАВА 19



## JavaFX CSS: изменяем вид интерфейса с помощью стилей

Библиотека JavaFX позволяет изменять внешний вид окна, контейнеров и компонентов с помощью *каскадных таблиц стилей* (Cascading Style Sheets, CSS) точно так же, как это делается в Web-программировании. Применение стилей позволяет задавать точные характеристики практически всех элементов пользовательского интерфейса.

### 19.1. Способы встраивания определения стиля

Пример изменения цвета текста надписи:

```
Label label = new Label("Текст надписи");
label.setStyle("-fx-text-fill: #ff0000;");
```

Прежде чем приступить к изучению JavaFX CSS, разберемся с основными понятиями. Значение параметра в методе `setStyle()` (`-fx-text-fill: #ff0000`) называется *определением стиля* или *стилем*. Элемент определения стиля (`-fx-text-fill`) называется *атрибутом*. Названия атрибутов в JavaFX CSS начинаются с префикса `-fx-`. Каждый атрибут имеет *значение* (`#ff0000`), указываемое после двоеточия. Совокупность определений стилей называют *таблицей стилей*.

Задать стиль можно двумя основными способами: встроить определение стиля в узел или вынести таблицу стилей в отдельный файл.

#### 19.1.1. Встраивание определения стиля в узел

Определение стиля встраивается в любой узел с помощью свойства `style` из класса `Node`. Методы доступа:

```
public final void setStyle(String value)
public final String getStyle()
public final StringProperty styleProperty()
```

**Пример изменения цвета текста надписи:**

```
Label label = new Label("Текст надписи");
label.setStyle("-fx-text-fill: #ff0000;");
```

Если определение стиля состоит из нескольких атрибутов, то они указываются через точку с запятой. Укажем цвет текста надписи и размер шрифта:

```
label.setStyle("-fx-text-fill: #ff0000; -fx-font-size: 12pt;");
```

Если какое-либо значение атрибута требует наличия кавычек, то оно указывается в апострофах:

```
label.setStyle("-fx-text-fill: #ff0000; -fx-font-size: 12pt;" +
 "-fx-font-family: 'Times New Roman'");
```

## 19.1.2. Вынесение таблицы стилей в отдельный файл

Таблицу стилей можно вынести в отдельный файл, который обычно имеет расширение `css`. Для подключения этого файла используются следующие свойства и методы:

❑ **метод `getStylesheets()` из класса `Parent`. Формат метода:**

```
public final ObservableList<String> getStylesheets()
```

**Пример (файл `style.css` в каталоге `src\application`):**

```
VBox pane = new VBox(10.0);
pane.getStylesheets().add(getClass().getResource(
 "/application/style.css").toExternalForm());
```

❑ **метод `getStylesheets()` из класса `Scene`. Формат метода:**

```
public final ObservableList<String> getStylesheets()
```

**Пример (файл `style.css` в каталоге `src\application`):**

```
Scene scene = new Scene(pane, 500, 300);
scene.getStylesheets().add(getClass().getResource(
 "/application/style.css").toExternalForm());
```

❑ **свойство `userAgentStylesheet` из классов `Scene` и `SubScene`. Методы доступа:**

```
public final void setUserAgentStylesheet(String url)
public final String getUserAgentStylesheet()
public final ObjectProperty<String> userAgentStylesheetProperty()
```

Подключить таблицу стилей для всего приложения позволяет статический метод `setUserAgentStylesheet()` из класса `Application`, а получить текущее значение или `null` можно с помощью метода `getUserAgentStylesheet()`. Форматы методов:

```
public static void setUserAgentStylesheet(String url)
public static String getUserAgentStylesheet()
```

В параметре `url` можно указать одну из констант из класса `Application`:

```
public static final String STYLESHEET_CASPIAN = "CASPIAN";
public static final String STYLESHEET_MODENA = "MODENA";
```

Стиль MODENA используется в приложении по умолчанию. Пример указания стиля CASPIAN:

```
Application.setUserAgentStylesheet(Application.STYLESHEET_CASPIAN);
```

Атрибуты определения стиля, указанные в отдельном файле, заключаются в фигурные скобки, перед которыми ставится *селектор*. Если атрибутов несколько, то они приводятся через точку с запятой (после последнего определения стиля перед закрывающей фигурной скобкой ставить точку с запятой не обязательно):

```
<Селектор> { <Атрибут 1>: <Значение 1>; ...; <Атрибут N>: <Значение N> }
```

Пример изменения цвета текста надписи и размера шрифта:

```
/* Так можно вставить комментарий */
.label { -fx-text-fill: green; -fx-font-size: 8pt; }
```

В качестве параметра <Селектор> часто используются: идентификатор компонента (формат #Идентификатор), стилевой класс (формат .Класс) или селектор типа (формат Тип). Для прикрепления идентификатора к узлу следует воспользоваться свойством `id` из класса `Node`. Методы доступа:

```
public final void setId(String value)
public final String getId()
public final StringProperty idProperty()
```

Пример прикрепления идентификатора к кнопке:

```
Button button = new Button("Нажми меня");
button.setId("my_button");
```

Пример назначения стиля:

```
#my_button { -fx-text-fill: green; }
```

Для назначения стилевого класса предназначен метод `getStyleClass()` из класса `Node`. Один узел может иметь сразу несколько стилевых классов. Формат метода:

```
public final ObservableList<String> getStyleClass()
```

Пример получения стилевого класса, назначаемого кнопкам по умолчанию:

```
Button button = new Button("Нажми меня");
System.out.println(button.getStyleClass()); // button
```

Теперь добавим свой стилевой класс в список:

```
Button button = new Button("Нажми меня");
button.getStyleClass().add("my_button");
```

Пример назначения стиля:

```
.my_button { -fx-text-fill: green; }
```

Для получения селектора типа предназначен метод `getTypeSelector()` из класса `Node`. Метод возвращает название класса. Формат метода:

```
public String getTypeSelector()
```

Пример получения селектора типа для кнопки:

```
Button button = new Button("Нажми меня");
System.out.println(button.getTypeSelector()); // Button
```

Пример назначения стиля для всех кнопок:

```
Button { -fx-text-fill: green; }
```

### 19.1.3. Приоритет применения стилей

Предположим, что для надписи указан стиль сразу в нескольких местах: в файле `style.css` (листинг 19.1), в файле `style_pane.css` (листинг 19.2), в файле `style_scene.css` (листинг 19.3), а также с помощью метода `setStyle()` (листинг 19.4).

#### Листинг 19.1. Содержимое файла `src\application\style.css`

```
.label { -fx-text-fill: yellow; -fx-font-size: 20pt; }
```

#### Листинг 19.2. Содержимое файла `src\application\style_pane.css`

```
.label { -fx-text-fill: red; -fx-font-size: 32pt; }
```

#### Листинг 19.3. Содержимое файла `src\application\style_scene.css`

```
.label { -fx-text-fill: green; -fx-font-size: 8pt; }
```

#### Листинг 19.4. Фрагмент содержимого файла `src\application\Main.java`

```
Stage window = new Stage();

VBox pane = new VBox(10.0);
pane.getStylesheets().add(getClass().getResource(
 "/application/style_pane.css").toExternalForm());
pane.setAlignment(Pos.CENTER);
pane.setPadding(new Insets(10.0));

Label label = new Label("Текст надписи");
label.setFont(Font.font(16.0));
label.setStyle("-fx-text-fill: blue; -fx-font-size: 5pt;");
pane.getChildren().add(label);

Scene scene = new Scene(pane, 500, 300);
scene.getStylesheets().add(getClass().getResource(
 "/application/style_scene.css").toExternalForm());
scene.setUserAgentStylesheet(getClass().getResource(
 "/application/style.css").toExternalForm());
window.setScene(scene);
window.setTitle("Приоритет применения стилей");
window.show();
```

Какого цвета и размера будет текст надписи? Для ответа на этот вопрос существует приоритет стилей (от низшего до высшего):

- ☐ свойство `userAgentStylesheet` из класса `Scene`;
- ☐ значение, установленное из кода (`label.setFont(Font.font(16.0));`);
- ☐ метод `getStylesheets()` из класса `Scene`;
- ☐ метод `getStylesheets()` из класса `Parent`;
- ☐ свойство `style` из класса `Node` (высший приоритет).

Иными словами, наивысший приоритет имеет встроенный стиль. В нашем примере к надписи будет применено форматирование, указанное в методе `setStyle()`. Именно из-за такой структуры приоритетов таблицы стилей называют *каскадными*.

Кроме того, следует учитывать, что стиль, заданный через идентификатор, будет иметь более высокий приоритет, чем стиль, заданный через класс. Пример:

```
#my_button { -fx-text-fill: red; }
.button { -fx-text-fill: green; }
```

В этом примере текст надписи на кнопке будет красного цвета, а не зеленого.

С помощью свойства `!important` можно изменить приоритет. Для примера изменим содержимое файла `style_pane.css` (листинг 19.2) на:

```
.label {
 -fx-text-fill: red !important;
 -fx-font-size: 32pt !important;
}
```

В результате мы переопределили все стили и надпись будет иметь красный цвет.

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH19_01_CSS`, приведенном в каталоге `CH19_CSS` сопровождающего книгу электронного архива (см. *приложение*).

## 19.2. Указание значений атрибутов

После значения в определении стиля необходимо уточнить, в каких единицах указано это значение. Впрочем, для значения 0 единицы измерения можно не указывать.

### 19.2.1. Числа

Дробная часть вещественного числа указывается через символ «точка»:

```
Label label = new Label("Текст надписи");
label.setStyle("-fx-text-fill: rgba(255, 0, 0, 0.5);");
```

Если перед символом «точка» указан нуль, то его можно опустить:

```
label.setStyle("-fx-text-fill: rgba(255, 0, 0, .5);");
```



## 19.2.2. Размеры

Размеры можно задавать в абсолютных или относительных единицах.

Абсолютные единицы:

- mm — миллиметр;
- cm — сантиметр;
- in — дюйм, 1 in = 2.54 cm;
- pt — пункт, 1 pt = 1/72 in;
- pc — пика, 1 pc = 12 pt.

Относительные единицы:

- px — пиксел (относительно устройства просмотра);
- % — процент;
- em — высота текущего шрифта;
- ex — высота буквы x текущего шрифта.

Пример указания внутреннего отступа для надписи:

```
Label label = new Label("Текст надписи");
label.setStyle("-fx-label-padding: 10px;");
```

## 19.2.3. Цвет

Цвет можно задать одним из следующих способов:

- именем цвета: blue, green и т. д.:  

```
.label { -fx-background-color: red; }
```
- значением вида #RGB, где R — насыщенность красного, G — насыщенность зеленого и B — насыщенность синего в цвете. Значения задаются одинарными шестнадцатеричными числами от 0 до F:

```
.label { -fx-background-color: #F00; }
```

- значением вида #RRGGBB, где RR — насыщенность красного, GG — насыщенность зеленого и BB — насыщенность синего в цвете. В таком формате значения задаются двузначными шестнадцатеричными числами от 00 до FF:

```
.label { -fx-background-color: #FF0000; }
```

- значением вида rgb(R, G, B), где R, G и B — насыщенности красного, зеленого и синего цветов, которые задаются десятичными числами от 0 до 255:

```
.label { -fx-background-color: rgb(255, 0, 0); }
```

- значением вида rgb(R%, G%, B%), где R%, G% и B% — насыщенности красного, зеленого и синего цветов, которые задаются в процентах:

```
.label { -fx-background-color: rgb(100%, 0%, 0%); }
```

- значением вида `rgba(R, G, B, A)`, где `R`, `G` и `B` — насыщенности красного, зеленого и синего цветов, которые задаются десятичными числами от 0 до 255, `A` — задает уровень прозрачности цвета (альфа-канал), представляющий собой значение от 0.0 (цвет полностью прозрачен) до 1.0 (цвет полностью непрозрачен). Пример указания полупрозрачного красного цвета:

```
.label { -fx-background-color: rgba(255, 0, 0, 0.5); }
```

Значения насыщенности красного, зеленого и синего цветов можно также задать в процентах:

```
.label { -fx-background-color: rgba(100%, 0%, 0%, 0.5); }
```

- значением вида `hsb(H, S, B)`, где `H` — оттенок (число от 0 до 359), `S` — насыщенность (проценты от 0 до 100) и `B` — яркость (проценты от 0 до 100):

```
.label { -fx-background-color: hsb(0, 100%, 50%); }
```

- значением вида `hsba(H, S, B, A)`, где `H` — оттенок (число от 0 до 359), `S` — насыщенность (проценты от 0 до 100), `B` — яркость (проценты от 0 до 100) и `A` — уровень прозрачности цвета (альфа-канал), представляющий собой значение от 0.0 (цвет полностью прозрачен) до 1.0 (цвет полностью непрозрачен):

```
.label { -fx-background-color: hsba(0, 100%, 50%, 0.5); }
```

- с помощью функции `derive(<Цвет>, <Яркость>%)`. В параметре `<Яркость>` указывается яркость цвета в процентах от -100 (более темный цвет) до 100 (более светлый цвет):

```
.label { -fx-background-color: derive(red, 50%); }
```

Можно также воспользоваться функциями: `ladder()`, `linear-gradient()` (линейный градиент), `radial-gradient()` (радиальный градиент), `image-pattern()` (заливка текстурой) и `repeating-image-pattern()`. Подробную информацию по этим функциям можно найти в документации. Пример заливки линейным градиентом:

```
.label {
 -fx-background-color: linear-gradient(from 0% 0% to 50% 0%, reflect,
 #000000 0%, #ffffff 100%);
}
```

Приведем названия наиболее часто используемых цветов:

- `black` — #000000 — черный;
- `white` — #FFFFFF — белый;
- `yellow` — #FFFF00 — желтый;
- `silver` — #C0C0C0 — серый;
- `red` — #FF0000 — красный;
- `green` — #008000 — зеленый;
- `gray` — #808080 — темно-серый;

- ❑ blue — #0000FF — синий;
- ❑ navy — #000080 — темно-синий;
- ❑ purple — #800080 — фиолетовый.

### 19.2.4. Строки

Строки указываются внутри двойных или одинарных кавычек. С помощью символа \ можно экранировать кавычку внутри строки:

```
.label { -fx-font-family: "Times New Roman"; }
.label { -fx-font-family: 'Arial'; }
```

### 19.2.5. Углы

Для обозначения углов используются следующие единицы:

- ❑ deg — градусы;
- ❑ rad — радианы;
- ❑ grad — грады;
- ❑ turn — повороты (полный круг равен 1turn).

## 19.3. Селекторы

Атрибуты определения стиля, указанные в отдельном файле, заключаются в фигурные скобки, перед которыми ставится *селектор*. Если атрибутов несколько, то они приводятся через точку с запятой в формате:

```
<Селектор> { <Атрибут 1>: <Значение 1>; ...; <Атрибут N>: <Значение N> }
```

### 19.3.1. Основные селекторы

В параметре <Селектор> могут быть указаны следующие селекторы:

- ❑ Тип — все узлы, имеющие указанное название класса:

```
Label { -fx-text-fill: #ff0000; }
```

- ❑ .Класс — все узлы, имеющие указанный стилевой класс:

```
.my_label { -fx-text-fill: #ff0000; }
...
Label label = new Label("Текст надписи");
label.getStyleClass().add("my_label");
```

- ❑ Тип.Класс — все узлы, имеющие указанное имя и стилевой класс:

```
Label.my_label { -fx-text-fill: #ff0000; }
```

- ❑ #Идентификатор — узел с указанным идентификатором:

```
#my_rect { -fx-fill: image-pattern(url(/application/img.png)); }
...
Rectangle rect = new Rectangle(150.0, 50.0);
rect.setId("my_rect");
```

Стили можно привязать сразу к нескольким селекторам, в этом случае селекторы указываются через запятую:

```
Label, Button { -fx-text-fill: #ff0000; }
```

Привязаться к другим элементам можно следующими способами:

- ❑ Селектор1 Селектор2 — все элементы, соответствующие параметру Селектор2, которые располагаются внутри контейнера, соответствующего параметру Селектор1:

```
HBox .label { -fx-text-fill: #ff0000; }
```

Цвет текста надписи станет красным, если надпись находится внутри HBox:

```
HBox hbox = new HBox();
Label label = new Label("Текст надписи");
hbox.getChildren().add(label);
```

- ❑ Селектор1 > Селектор2 — все элементы, соответствующие параметру Селектор2, которые являются дочерними для контейнера, соответствующего параметру Селектор1:

```
#box1 > .label { -fx-text-fill: #ff0000; }
```

Цвет текста надписи станет красным, если надпись находится внутри контейнера с идентификатором box1 и не вложена в другой контейнер:

```
VBox box1 = new VBox();
box1.setId("box1");
Label label = new Label("Надпись 1");
VBox box2 = new VBox(new Label("Надпись 2"));
box1.getChildren().addAll(label, box2);
```

В этом примере только Надпись 1 станет красного цвета, т. к. вторая надпись расположена внутри контейнера box2.

## 19.3.2. Псевдоклассы

В составе селектора могут быть также указаны следующие *псевдоклассы*:

- ❑ :visited — посещенная гиперссылка:

```
Hyperlink:visited { -fx-text-fill: #ff0000; }
```

- ❑ :pressed — нажатие кнопки мыши над узлом:

```
Hyperlink:pressed { -fx-text-fill: blue; }
```

- **:hover** — узел, над которым находится указатель мыши. Изменим цвет текста надписи на зеленый при наведении указателя мыши:

```
.label:hover { -fx-text-fill: green; }
```

- **:focused** — узел, имеющий фокус ввода. При получении фокуса ввода сделаем фон текстового поля зеленым:

```
TextField:focused { -fx-background-color: green; }
```

- **:disabled** — узел, недоступный для пользователя:

```
.label:disabled { -fx-background-color: red; }
```

- **:armed** — нажатие кнопки пользователем:

```
Button:armed { -fx-text-fill: #ff0000; }
```

- **:default** — кнопка по умолчанию:

```
Button:default { -fx-background-color: green; }
...
Button btnDefault = new Button("Кнопка по умолчанию");
btnDefault.setDefaultButton(true);
```

- **:cancel** — кнопка отмены:

```
Button:cancel { -fx-background-color: silver; }
...
Button btnCancel = new Button("Кнопка отмены");
btnCancel.setCancelButton(true);
```

- **:selected** — свойство `selected` класса `CheckBox` имеет значение `true`:

```
CheckBox:selected { -fx-text-fill: green; }
```

- **:determinate** — свойство `indeterminate` класса `CheckBox` имеет значение `false`:

```
CheckBox:determinate { -fx-text-fill: blue; }
```

- **:indeterminate** — свойство `indeterminate` класса `CheckBox` имеет значение `true`:

```
CheckBox:indeterminate { -fx-text-fill: red; }
```

Существуют и другие псевдоклассы, специфичные для компонентов. Полный список псевдоклассов смотрите в документации.

## 19.4. Форматирование шрифта

Каскадные таблицы стилей позволяют задать название и размер шрифта, его стиль и «жирность», а также цвет текста. Кроме того, можно указать одно из названий альтернативных семейств, поскольку на компьютере пользователя может не быть нужного шрифта.

### 19.4.1. Имя шрифта

Атрибут `-fx-font-family` позволяет задать имя шрифта:

```
Label label = new Label("Текст надписи");
label.setStyle("-fx-font-family: 'Courier New');
```

В ряде случаев шрифт может отсутствовать на компьютере пользователя. Поэтому можно указать одно из пяти типовых семейств шрифтов: `serif`, `sans-serif`, `cursive`, `fantasy` или `monospace`:

```
label.setStyle("-fx-font-family: fantasy;");
```

### 19.4.2. Стил ь шрифта

Атрибут `-fx-font-style` позволяет задать стиль шрифта. Он может принимать следующие значения:

- ☐ `normal` — нормальный шрифт;
- ☐ `italic` — курсивный шрифт;
- ☐ `oblique` — наклонный шрифт.

Пример:

```
label.setStyle("-fx-font-family: Arial; -fx-font-style: italic;");
```

### 19.4.3. Размер шрифта

Атрибут `-fx-font-size` позволяет задать размер шрифта:

```
label.setStyle("-fx-font-size: 16pt;");
```

Можно указать абсолютную величину или одну из типовых констант: `xx-small`, `x-small`, `small`, `medium`, `large`, `x-large` или `xx-large`:

```
label.setStyle("-fx-font-size: xx-large;");
```

Кроме того, можно указать относительную величину (например, значение в процентах) или одну из двух констант: `larger` или `smaller`:

```
label.setStyle("-fx-font-size: 150%;");
label2.setStyle("-fx-font-size: smaller;");
```

### 19.4.4. Цвет текста

Атрибут `-fx-text-fill` позволяет задать цвет текста:

```
label.setStyle("-fx-text-fill: red;");
```

### 19.4.5. Жирность шрифта

Атрибут `-fx-font-weight` позволяет управлять жирностью шрифта. Может принимать следующие значения:

- 100, 200, 300, 400, 500, 600, 700, 800, 900 — значение 100 соответствует самому бледному шрифту, а 900 — самому жирному;
- normal — нормальный шрифт. Соответствует значению 400;
- bold — полужирный шрифт. Соответствует значению 700;
- lighter — менее жирный, чем у родительского элемента;
- bolder — более жирный, чем у родительского элемента.

Пример:

```
label.setStyle("-fx-font-weight: bold;");
```

### 19.4.6. Одновременное указание характеристик шрифта

Атрибут `-fx-font` позволяет задать все характеристики шрифта одновременно:

```
-fx-font: [-fx-font-style][-fx-font-weight]
 -fx-font-size -fx-font-family
```

Пример:

```
label.setStyle("-fx-font: italic bold 16pt 'Courier New';");
```

### 19.4.7. Загружаемые шрифты

Помимо шрифтов, установленных на компьютере, и типовых семейств шрифтов, для вывода текста мы можем использовать загружаемые шрифты.

Загружаемый шрифт указывается с помощью правила `@font-face`:

```
@font-face {
 font-family: <Имя, под которым загружаемый шрифт будет доступен в
 таблице стилей>;
 src: url("<URL-адрес файла со шрифтом>")
}
```

Пример:

```
@font-face {
 font-family: MyFont;
 src: url(file:/C:/Windows/Fonts/Tahoma.ttf)
}
```

После этого мы можем использовать загруженный шрифт где угодно, указав заданное для него имя:

```
label.setStyle("-fx-font: 16pt MyFont;");
```

#### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH19_01_CSS`, приведенном в каталоге `CH19_CSS` сопровождающего книгу электронного архива (см. *приложение*).

## 19.5. Форматирование текста

Для надписей, кроме указания характеристик шрифтов, можно задать некоторые дополнительные параметры — например, горизонтальное выравнивание текста при использовании многострочного режима.

### 19.5.1. Подчеркивание и зачеркивание текста

Атрибут `-fx-underline` позволяет подчеркнуть текст, если указано значение `true`:

```
Label label = new Label("Подчеркнутый текст");
label.setStyle("-fx-underline: true;");
```

Для зачеркивания текста (применительно только к фигуре `Text`) нужно присвоить значение `true` атрибуту `-fx-strikethrough`:

```
Text text = new Text("Зачеркнутый текст");
text.setStyle("-fx-strikethrough: true;");
```

### 19.5.2. Обрезка или перенос текста на новую строку

Атрибут `-fx-ellipsis-string` задает строку, которая будет отображаться при обрезке текста (текст обрезается, если он не помещается на одной строке или внутри области). Значение по умолчанию: `...`. Пример:

```
Label label = new Label(createLongText(20));
label.setStyle("-fx-ellipsis-string: '>>>'");
```

Атрибут `-fx-text-overflow` задает режим обрезки текста, если он не помещается внутри области. Можно указать значения: `ellipsis`, `word-ellipsis`, `center-ellipsis`, `center-word-ellipsis`, `leading-ellipsis`, `leading-word-ellipsis` и `clip`. Описание значений см. в *разд. 11.3*. Пример обрезки по центру:

```
label.setStyle("-fx-text-overflow: center-ellipsis;");
```

Если атрибут `-fx-wrap-text` имеет значение `true`, то, если текст не помещается на одной строке, он будет переноситься на новую строку:

```
label.setStyle("-fx-wrap-text: true;");
```

### 19.5.3. Горизонтальное выравнивание текста

Атрибут `-fx-text-alignment` задает горизонтальное выравнивание текста при использовании многострочного режима. Этот атрибут может принимать следующие значения:

- ☐ `center` — выравнивание по центру;
- ☐ `left` — выравнивание по левому краю;
- ☐ `right` — выравнивание по правому краю;
- ☐ `justify` — выравнивание по ширине (по двум сторонам).



**Пример выравнивания по центру:**

```
Label label = new Label(createLongText(40));
label.setStyle("-fx-wrap-text: true; -fx-text-alignment: center;");
```

## 19.5.4. Взаимодействие между изображением и текстом

Атрибут `-fx-graphic` задает интернет-адрес (URL) изображения, которое будет отображаться рядом с текстом надписи или вместо него. Пример:

```
Label label = new Label("Текст надписи");
label.setStyle("-fx-graphic: url(/application/img.png);");
```

Атрибут `-fx-content-display` задает положение изображения относительно текста. Можно указать значения: `left`, `right`, `center`, `top`, `bottom`, `text-only` и `graphic-only`. Описание значений см. в разд. 11.3. Пример вывода изображения под текстом:

```
ImageView iv = new ImageView(createImage(50, 50, Color.BLUE));
Label label = new Label("Текст надписи", iv);
label.setStyle("-fx-content-display: bottom;");
```

Атрибут `-fx-graphic-text-gap` задает расстояние между изображением и текстом:

```
label.setStyle("-fx-graphic-text-gap: 20px;");
```

## 19.5.5. Выравнивание изображения и текста внутри области

Атрибут `-fx-alignment` задает выравнивание текста и изображения внутри свободной области компонента. Если такой свободной области нет, то и эффекта не будет видно. Можно указать значения: `baseline-center`, `baseline-left`, `baseline-right`, `bottom-center`, `bottom-left`, `bottom-right`, `center`, `center-left`, `center-right`, `top-center`, `top-left` и `top-right`. Пример выравнивания по центру области:

```
ImageView iv = new ImageView(createImage(30, 30, Color.BLUE));
Label label = new Label("Текст надписи", iv);
label.setPrefSize(400.0, 150.0);
label.setStyle("-fx-background-color: green; -fx-alignment: center;");
```

Атрибут `-fx-alignment` задает отступ от границы компонента до текста и изображения. Способы указания значения:

```
-fx-label-padding: <Значение для всех сторон>;
-fx-label-padding: <top> <right> <bottom> <left>;
```

**Пример:**

```
ImageView iv = new ImageView(createImage(30, 30, Color.BLUE));
Label label = new Label("Текст надписи", iv);
label.setPrefSize(400.0, 150.0);
label.setStyle("-fx-background-color: green; " +
 "-fx-alignment: top-right; -fx-label-padding: 20.0px;");
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH19_01_CSS`, приведенном в каталоге `CH19_CSS` сопровождающего книгу электронного архива (см. *приложение*).

## 19.6. Размеры и внутренние отступы

Задать размеры контейнеров и компонентов позволяют следующие атрибуты:

- ☐ `-fx-pref-width` и `-fx-pref-height` — предпочтительные размеры;
- ☐ `-fx-min-width` и `-fx-min-height` — минимальные размеры;
- ☐ `-fx-max-width` и `-fx-max-height` — максимальные размеры.

Пример:

```
Label label = new Label("Текст надписи");
label.setStyle("-fx-background-color: green; " +
 "-fx-pref-width: 400px; -fx-pref-height: 100px; " +
 "-fx-min-width: 200px; -fx-min-height: 50px; " +
 "-fx-max-width: 400px; -fx-max-height: 150px;");
```

Внутренний отступ — это расстояние между узлом и границей области. Указать размер отступа позволяет атрибут `-fx-padding`. Способы указания значения:

```
-fx-padding: <Значение для всех сторон>;
-fx-padding: <top> <right> <bottom> <left>;
```

Пример:

```
Label label = new Label("Текст надписи");
label.setStyle("-fx-background-color: green; -fx-padding: 10px;");
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH19_01_CSS`, приведенном в каталоге `CH19_CSS` сопровождающего книгу электронного архива (см. *приложение*).

## 19.7. Рамки

Компоненты и контейнеры, классы которых наследуют класс `Region`, могут иметь одну или несколько рамок. Задать характеристики рамок можно с помощью CSS.

### 19.7.1. Стилль линии рамки

С помощью атрибута `-fx-border-style` можно задать стилль линии рамки. Формат:

```
-fx-border-style: <Стилль> [<Сдвиг>] [<Положение>] [<Форма соединения>]
 [<Форма окончания>]
```

Параметр `<Стилль>` может принимать следующие значения:

- ☐ `none` — линия не отображается;
- ☐ `solid` — сплошная линия;

- ☐ dotted — пунктирная линия;
- ☐ dashed — штриховая линия;
- ☐ segments(<Число 1>, ..., <Число N>) — четные индексы списка задают длину штриха, а нечетные — длину пропуска.

Пример добавления пунктирной рамки толщиной 5 px:

```
Label label = new Label("Текст надписи");
label.setPrefSize(400.0, 50.0);
label.setStyle("-fx-background-color: green; -fx-padding: 5px; " +
 "-fx-border-style: dotted; " +
 "-fx-border-width: 5px;");
```

Параметр <Сдвиг> задает смещение начала пунктирной обводки в формате:

phase <Число>

Пример:

```
label.setStyle("-fx-background-color: green; -fx-padding: 5px; " +
 "-fx-border-style: segments(25.0, 15.0) phase 10.0; " +
 "-fx-border-width: 5px;");
```

Параметр <Положение> задает положение рамки относительно границы:

- ☐ centered — по центру границы;
- ☐ inside — внутри области;
- ☐ outside — снаружи области.

Пример:

```
label.setStyle("-fx-background-color: green; -fx-padding: 5px; " +
 "-fx-border-style: dashed centered; " +
 "-fx-border-width: 5px;");
```

Параметр <Форма соединения> задает форму окончания в месте соединения двух линий рамки. Формат:

line-join <Форма>

В параметре <Форма> можно указать следующие значения:

- ☐ miter <limit> — обычные углы;
- ☐ bevel — скошенные углы;
- ☐ round — закругленные углы.

Параметр <Форма окончания> задает форму окончания линии или штриха. Формат:

line-cap <Форма>

В параметре <Форма> можно указать следующие значения:

- ☐ square — квадратные концы (прибавляются к длине линии или штриха);
- ☐ butt — концы никак не оформляются;
- ☐ round — закругленные концы (прибавляются к длине линии или штриха).

Пример:

```
label.setStyle("-fx-background-color: green; -fx-padding: 5px; " +
 "-fx-border-style: dashed line-join round line-cap round; " +
 "-fx-border-width: 25px;");
```

## 19.7.2. Толщина линии рамки

С помощью атрибута `-fx-border-width` можно задать толщину линии рамки. Способы указания значения:

```
-fx-border-width: <Значение для всех сторон>;
-fx-border-width: <top> <right> <bottom> <left>;
```

Пример добавления сплошной рамки толщиной 3 px:

```
Label label = new Label("Текст надписи");
label.setPrefSize(400.0, 50.0);
label.setStyle("-fx-background-color: green; -fx-padding: 5px; " +
 "-fx-border-style: solid; -fx-border-width: 3px;");
```

## 19.7.3. Цвет линии рамки

С помощью атрибута `-fx-border-color` можно задать цвет линий рамки. Способы указания значения:

```
-fx-border-color: <Значение для всех сторон>;
-fx-border-color: <top> <right> <bottom> <left>;
```

Пример:

```
Label label = new Label("Текст надписи");
label.setPrefSize(400.0, 50.0);
label.setStyle("-fx-background-color: green; -fx-padding: 5px; " +
 "-fx-border-style: solid; -fx-border-width: 5px; " +
 "-fx-border-color: red;");
```

## 19.7.4. Рамки со скругленными углами

Указать радиус скругления для углов рамки можно с помощью атрибута `-fx-border-radius`:

```
-fx-border-radius: <Радиус>[/ <Радиус>]
-fx-border-radius: <top-left> <top-right> <bottom-right> <bottom-left>
 [/ <top-left> <top-right> <bottom-right> <bottom-left>]
```

Перед символом слеша задается радиус скругления для горизонтальной четверти углов, а после него — радиус скругления для вертикальной четверти. Если после слеша значение не указать, то заданный радиус скругления будет применен и к горизонтальной, и к вертикальной четвертям:

```
Label label = new Label("Текст надписи");
label.setPrefSize(400.0, 50.0);
label.setStyle("-fx-background-color: green; -fx-padding: 5px; " +
 "-fx-border-style: solid; -fx-border-width: 5px; " +
 "-fx-border-radius: 10.0px;");
```

Пример указания разного радиуса для углов:

```
label.setStyle("-fx-background-color: green; -fx-padding: 5px; " +
 "-fx-border-style: solid; -fx-border-width: 1px; " +
 "-fx-border-radius: 10px 20px 30px 40px;");
```

## 19.7.5. Расстояние между рамкой и границей

Атрибут `-fx-border-insets` позволяет задать расстояние между рамкой и границей. Способы указания значения:

```
-fx-border-insets: <Значение для всех сторон>;
-fx-border-insets: <top> <right> <bottom> <left>;
```

Пример:

```
Label label = new Label("Текст надписи");
label.setPrefSize(400.0, 50.0);
label.setStyle("-fx-background-color: green; -fx-padding: 5px; " +
 "-fx-border-style: solid; -fx-border-width: 1px; " +
 "-fx-border-insets: 10px;");
```

Помимо рассмотренных атрибутов, JavaFX CSS содержит атрибуты, позволяющие создать рамку на основе изображения. Подробную информацию по этим атрибутам смотрите в документации.

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH19_01_CSS`, приведенном в каталоге `CH19_CSS` сопровождающего книгу электронного архива (см. *приложение*).

## 19.8. Фон

Каскадные таблицы стилей позволяют задать цвет фона или использовать изображение в качестве фона. Для фонового рисунка можно задать начальное положение, можно также управлять повторением фонового рисунка, что позволяет получить интересные спецэффекты.

### 19.8.1. Цвет фона

С помощью атрибута `-fx-background-color` можно задать цвет фона:

```
Label label = new Label("Текст надписи");
label.setPrefSize(400.0, 50.0);
label.setStyle("-fx-background-color: green;");
```

В качестве значения атрибута можно использовать слово `transparent`, которое означает, что фон прозрачный:

```
label.setStyle("-fx-background-color: transparent;");
```

## 19.8.2. Расстояние между фоном и границей

Атрибут `-fx-background-insets` задает расстояние между границей и фоном:

```
-fx-background-insets: <Значение для всех сторон>;
-fx-background-insets: <top> <right> <bottom> <left>;
```

Пример:

```
Label label = new Label("Текст надписи");
label.setPrefSize(400.0, 50.0);
label.setStyle("-fx-background-color: green; " +
 "-fx-background-insets: 5px; -fx-border-style: solid;");
```

## 19.8.3. Скругление углов фона

Атрибут `-fx-background-radius` позволяет указать радиус скругления углов фона:

```
-fx-background-radius: <Радиус>[/ <Радиус>];
-fx-background-radius:
 <top-left> <top-right> <bottom-right> <bottom-left>
 [/ <top-left> <top-right> <bottom-right> <bottom-left>]
```

Пример:

```
Label label = new Label("Текст надписи");
label.setPrefSize(400.0, 50.0);
label.setStyle("-fx-background-color: green; " +
 "-fx-background-radius: 10px;");
```

## 19.8.4. Фоновый рисунок

С помощью атрибута `-fx-background-image` можно задать интернет-адрес (URL) изображения, которое будет использовано в качестве фонового рисунка:

```
Label label = new Label("Текст надписи");
label.setPrefSize(400.0, 50.0);
label.setStyle("-fx-background-image: url(/application/img.png);");
```

## 19.8.5. Режим повтора фонового рисунка

Атрибут `-fx-background-repeat` задает режим повтора фонового рисунка. Он может принимать следующие значения:

- ☐ `repeat` — рисунок повторяется и по вертикали, и по горизонтали;
- ☐ `repeat-x` — рисунок повторяется по горизонтали;

- ☐ `repeat-y` — рисунок повторяется по вертикали;
- ☐ `no-repeat` — рисунок не повторяется;
- ☐ `space` — повторяет изображение без обрезки и масштабирования, при этом заполняет оставшееся место пустым пространством между изображениями;
- ☐ `round` — повторяет изображение таким образом, чтобы поместилось целое число изображений без пустого пространства между ними, при этом изображения масштабируются.

Пример повтора фонового рисунка по горизонтали и вертикали:

```
Label label = new Label("Текст надписи");
label.setPrefHeight(200.0);
label.setMaxWidth(Double.MAX_VALUE);
label.setStyle("-fx-background-image: url(/application/img.png); " +
 "-fx-background-repeat: repeat;");
```

### 19.8.6. Положение фонового рисунка

Атрибут `-fx-background-position` задает начальное положение фонового рисунка. Координаты указываются через пробел:

```
Label label = new Label("Текст надписи");
label.setPrefHeight(200.0);
label.setMaxWidth(Double.MAX_VALUE);
label.setStyle("-fx-background-image: url(/application/img.png); " +
 "-fx-background-repeat: no-repeat; " +
 "-fx-background-position: 50% 50%;");
```

Кроме того, могут быть указаны следующие значения:

- ☐ `left` — выравнивание по левому краю;
- ☐ `right` — по правому краю;
- ☐ `center` — по центру;
- ☐ `top` — по верху;
- ☐ `bottom` — по низу.

Пример:

```
label.setStyle("-fx-background-image: url(/application/img.png); " +
 "-fx-background-repeat: no-repeat; " +
 "-fx-background-position: left center;");
```

### 19.8.7. Размеры фонового рисунка

Атрибут `-fx-background-size` задает размеры фонового рисунка в виде двух значений, разделенных пробелом: первое значение задает ширину рисунка, а второе — высоту. Растянем фоновый рисунок на всю ширину и высоту:

```
Label label = new Label("Текст надписи");
label.setPrefHeight(200.0);
label.setMaxWidth(Double.MAX_VALUE);
label.setStyle("-fx-background-image: url(/application/img.png); " +
 "-fx-background-repeat: no-repeat; " +
 "-fx-background-size: 100% 100%;");
```

Можно также указать следующие значения:

- ☐ `auto` — не изменяет размеры фонового рисунка;
- ☐ `cover` — задает такие размеры фонового рисунка, чтобы он полностью покрывал фон. При этом какие-то фрагменты рисунка могут выйти за границы фона;
- ☐ `contain` — задает такие размеры фонового рисунка, чтобы он полностью помещался внутри фона, занимая его целиком. При этом какие-то части фона могут быть не покрыты рисунком;
- ☐ `stretch` — растягивает фоновый рисунок по ширине и высоте.

Заполним фоновым рисунком все пространство:

```
label.setStyle("-fx-background-image: url(/application/img.png); " +
 "-fx-background-repeat: no-repeat; " +
 "-fx-background-size: stretch;");
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH19_01_CSS`, приведенном в каталоге `CH19_CSS` сопровождающего книгу электронного архива (см. *приложение*).

## 19.9. Трансформации и эффекты

С помощью стилей можно применить к узлу различные трансформации и эффекты — например, изменить прозрачность, повернуть узел на некоторый угол, изменить масштаб, создать эффект тени и др.

### 19.9.1. Изменение прозрачности

Атрибут `-fx-opacity` позволяет изменить прозрачность узла. В качестве значения указывается вещественное число от 0.0 до 1.0. Пример:

```
Rectangle rect = new Rectangle(200.0, 20.0, Color.RED);
rect.setStyle("-fx-opacity: 0.7;");
```

### 19.9.2. Смещение

Сместить узел на некоторое расстояние позволяют атрибуты: `-fx-translate-x`, `-fx-translate-y` и `-fx-translate-z`:

```
Rectangle rect = new Rectangle(50.0, 50.0, Color.RED);
rect.setStyle("-fx-translate-x: 100.0; -fx-translate-y: 10.0;");
```



### 19.9.3. Масштабирование

Изменить масштаб узла можно с помощью атрибутов: `-fx-scale-x`, `-fx-scale-y` и `-fx-scale-z`. Пример увеличения ширины прямоугольника в два раза:

```
Rectangle rect = new Rectangle(50.0, 50.0, Color.RED);
rect.setStyle("-fx-scale-x: 2.0; -fx-scale-y: 1.0;");
```

### 19.9.4. Вращение

Повернуть узел на некоторый угол в градусах позволяет атрибут `-fx-rotate`:

```
Rectangle rect = new Rectangle(50.0, 50.0, Color.RED);
rect.setStyle("-fx-rotate: 45.0;");
```

### 19.9.5. Изменение режима наложения

Атрибут `-fx-blend-mode` позволяет изменить режим наложения. В качестве значения можно указать режимы: `src-over`, `src-atop`, `darken`, `multiply`, `color-burn`, `lighten`, `screen`, `color-dodge`, `overlay`, `soft-light`, `hard-light`, `difference` и `exclusion` (описание значений см. в *разд. 5.6*). Пример:

```
Rectangle rect = new Rectangle(50.0, 50.0, Color.GREEN);
Rectangle rect2 = new Rectangle(50.0, 50.0, Color.BLACK);
rect2.relocate(25.0, 25.0);
rect2.setStyle("-fx-blend-mode: overlay;");
Pane panel = new Pane();
panel.getChildren().addAll(rect, rect2);
```

### 19.9.6. Применение эффектов

Применить эффект к узлу позволяет атрибут `-fx-effect`. В качестве значения можно указать функции `dropshadow()` и `innershadow()`. Пример добавления внешней тени:

```
Rectangle rect = new Rectangle(50.0, 50.0, Color.BLUE);
rect.setStyle(
 "-fx-effect: dropshadow(gaussian, black, 10.0, 0.0, 5.0, 5.0);");
```

Пример добавления внутренней тени:

```
Rectangle rect = new Rectangle(50.0, 50.0, Color.GREEN);
rect.setStyle(
 "-fx-effect: innershadow(gaussian, black, 10.0, 0.0, -3.0, -3.0);");
```

#### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH19_01_CSS`, приведенном в каталоге `CH19_CSS` сопровождающего книгу электронного архива (см. *приложение*).

## 19.10. Изменение внешнего вида указателя мыши

Атрибут `-fx-cursor` задает форму указателя мыши при наведении на узел. Может принимать следующие значения: `default`, `hand`, `closed-hand`, `open-hand`, `wait`, `text`, `crosshair`, `move`, `e-resize`, `h-resize`, `n-resize`, `ne-resize`, `nw-resize`, `s-resize`, `se-resize`, `sw-resize`, `v-resize`, `w-resize` или интернет-адрес (URL) курсора. Пример:

```
Rectangle rect = new Rectangle(100.0, 100.0, Color.RED);
rect.setStyle("-fx-cursor: hand;");
```

## 19.11. Изменение характеристик фигуры

С помощью CSS можно изменить характеристики заливки и обводки фигуры, управлять сглаживанием, а также задавать диаметр скругления углов прямоугольника.

### 19.11.1. Фон фигуры

Атрибут `-fx-fill` задает характеристики фона фигуры:

```
Rectangle rect = new Rectangle(200.0, 20.0);
rect.setStyle("-fx-fill: green;");
```

### 19.11.2. Обводка

Указать характеристики обводки позволяют следующие атрибуты:

- ☐ `-fx-stroke` — цвет обводки;
- ☐ `-fx-stroke-width` — ширина (толщина) обводки:  

```
Rectangle rect = new Rectangle(200.0, 30.0, Color.WHITE);
rect.setStyle("-fx-stroke: red; -fx-stroke-width: 3px;");
```
- ☐ `-fx-stroke-type` — положение обводки относительно границы фигуры:
  - `centered` — по центру границы;
  - `inside` — внутри фигуры;
  - `outside` — снаружи фигуры;
- ☐ `-fx-stroke-dash-array` — задает значения для пунктирной линии. Четные индексы списка задают длину штриха, а нечетные — длину пропуска;
- ☐ `-fx-stroke-dash-offset` — смещение начала пунктирной обводки:

```
Rectangle rect = new Rectangle(200.0, 50.0, Color.WHITE);
rect.setStyle("-fx-stroke: red; -fx-stroke-width: 5px; " +
 "-fx-stroke-type: centered; -fx-stroke-dash-offset: 10.0; " +
 "-fx-stroke-dash-array: 25.0px, 15.0px;");
```

□ `-fx-stroke-line-cap` — форма окончания линии или штриха:

- `square` — квадратные концы (прибавляются к длине линии или штриха);
- `butt` — концы никак не оформляются;
- `round` — закругленные концы (прибавляются к длине линии или штриха).

Пример:

```
Line line = new Line(100.0, 100.0, 200.0, 200.0);
line.setStyle("-fx-stroke: blue; -fx-stroke-width: 10px; " +
 "-fx-stroke-line-cap: round;");
```

□ `-fx-stroke-line-join` — форма окончания в месте соединения двух линий обводки:

- `miter` — обычные углы;
- `bevel` — скошенные углы;
- `round` — закругленные углы;

□ `-fx-stroke-miter-limit` — ограничение длины угла при использовании стиля `-fx-stroke-line-join: miter`:

```
Polygon shape = new Polygon();
shape.getPoints().addAll(new Double[] {
 0.0, 0.0,
 150.0, 50.0,
 0.0, 100.0
});
shape.setStyle("-fx-stroke: red; -fx-stroke-width: 15px; " +
 "-fx-stroke-line-join: miter; -fx-fill: white; " +
 "-fx-stroke-miter-limit: 2.0;");
```

### 19.11.3. Сглаживание

При выводе некоторых фигур (например, круга, треугольника или текста) контур может отображаться в виде «лесенки». Управлять сглаживанием контуров позволяет атрибут `-fx-smooth`. Пример включения сглаживания для круга:

```
Circle shape = new Circle(50.0);
shape.setStyle("-fx-stroke: red; -fx-stroke-width: 5px; " +
 "-fx-smooth: true; -fx-fill: white;");
```

### 19.11.4. Скругление углов прямоугольника

Для фигуры `Rectangle` с помощью атрибутов `-fx-arc-width` и `-fx-arc-height` можно указать диаметр скругления углов:

```
Rectangle rect = new Rectangle(200.0, 100.0, Color.RED);
rect.setStyle("-fx-arc-width: 20px; -fx-arc-height: 20px;");
```

## 19.12. Изменение характеристик полос прокрутки

Изменить характеристики полосы прокрутки позволяют следующие атрибуты:

- ❑ `-fx-orientation` — задает ориентацию компонента. Значения: `horizontal` или `vertical`;
- ❑ `-fx-block-increment` — задает интервал, на который изменится значение при щелчке мышью на полосе;
- ❑ `-fx-unit-increment` — задает интервал, на который изменится значение при нажатии кнопок со стрелками, повороте колесика мыши, а также при вызове методов `increment()` и `decrement()`.

Пример:

```
ScrollBar scrollBar = new ScrollBar();
scrollBar.setPrefSize(200.0, 20.0);
scrollBar.setMin(0.0);
scrollBar.setMax(100.0);
scrollBar.setValue(0.0);
scrollBar.setStyle("-fx-block-increment: 5.0; " +
 "-fx-orientation: horizontal; " +
 "-fx-unit-increment: 1.0;");
```

Полосы прокрутки поддерживают два псевдокласса:

- ❑ `:horizontal` — горизонтальная полоса прокрутки;
- ❑ `:vertical` — вертикальная полоса прокрутки.

Структура стилевых классов компонента `ScrollBar`:

```
.scroll-bar
 .increment-button
 .increment-arrow
 .decrement-button
 .decrement-arrow
 .track
 .thumb
```

Пример изменения размеров кнопки увеличения значения и цвета стрелки на ней:

```
.scroll-bar .decrement-button {
 -fx-pref-width: 20px; -fx-pref-height: 20px;
}
.scroll-bar:horizontal .decrement-button .decrement-arrow {
 -fx-background-color: red;
}
```

Изменим цвет фона ползунка горизонтальной полосы прокрутки:

```
.scroll-bar:horizontal .thumb {
 -fx-background-color: green;
}
```

Изменить характеристики контейнера `ScrollPane` позволяют следующие атрибуты:

- ❑ `-fx-hbar-policy` — задает режим отображения горизонтальной полосы прокрутки: `as_needed`, `always` или `never`;
- ❑ `-fx-vbar-policy` — задает режим отображения вертикальной полосы прокрутки: `as_needed`, `always` или `never`;
- ❑ `-fx-pannable` — если атрибут имеет значение `true`, то содержимое области можно будет перемещать с помощью мыши;
- ❑ `-fx-fit-to-width` — если атрибут имеет значение `true`, то узел внутри области будет менять свою ширину одновременно с изменением размера области, при условии, что узел поддерживает изменение размера. Например, контейнер будет заполнять всю доступную ширину области;
- ❑ `-fx-fit-to-height` — если атрибут имеет значение `true`, то узел внутри области будет менять свою высоту одновременно с изменением размера области, при условии, что узел поддерживает изменение размера.

Структура стилевых классов контейнера `ScrollPane`:

```
.scroll-pane
 .viewport
 .scroll-bar:horizontal
 .scroll-bar:vertical
 .corner
```

Изменим толщину полос прокрутки:

```
.scroll-pane .scroll-bar:horizontal { -fx-pref-height: 20px; }
.scroll-pane .scroll-bar:vertical { -fx-pref-width: 20px; }
```

Зададим цвет фона:

```
.scroll-pane .viewport { -fx-background-color: black; }
```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH19_01_CSS`, приведенном в каталоге `CH19_CSS` сопровождающего книгу электронного архива (см. приложение).

В этой главе мы рассмотрели только наиболее часто используемые атрибуты стилей. Каждый узел может иметь свой собственный набор атрибутов и псевдоклассов. В большинстве случаев названия атрибутов совпадают с названиями свойств узла, но названия обычно используются в нижнем регистре и между словами вставляется дефис, а перед названием атрибута — префикс `-fx-`. Например, свойство `prefWidth` соответствует атрибуту `-fx-pref-width`. Полный список таких атрибутов стилей смотрите в документации.

# ГЛАВА 20



## Разное

В этой главе мы рассмотрим возможность вывода текста надписей на различных языках в зависимости от настроек локали, научимся отображать окно с заставкой на время запуска приложения, а также создадим дистрибутив для запуска приложения, включающий только необходимые модули и компоненты.

### 20.1. Интернационализация приложения

В предыдущих главах книги мы задавали текст надписей явным образом:

```
Button btnClose = new Button("Закрыть");
```

Это было сделано лишь для сокращения объема книги и упрощения изучения нового материала. В реальных приложениях лучше так не делать. Ведь приложением могут пользоваться люди в разных странах, и этим людям будет гораздо приятнее видеть интерфейс программы на родном языке. Чтобы язык текста надписей соответствовал настройкам локали, нужно создать файлы с расширением `properties`, внутри которых прописать текст на разных языках. Названия файлов состоят из следующих частей:

```
<Название приложения>.properties
<Название приложения>_<Язык>.properties
<Название приложения>_<Язык>_<Страна>.properties
```

Первый файл будет использоваться по умолчанию, если нет файла для указанного языка, второй файл является общим для указанного языка, а третий файл учитывает и язык, и страну. Примеры названий файлов:

```
MyApp.properties
MyApp_en.properties
MyApp_ru_RU.properties
```

Создать такие файлы позволяет класс `Properties`:

```
// import java.util.Properties;
// import java.io.*;
FileOutputStream fout = null;
```



В первом параметре указывается название приложения, перед которым можно добавить название каталога, внутри которого расположены файлы. Второй параметр позволяет задать локаль. Если параметр не указан, то будет использоваться локаль по умолчанию. Пример создания объекта для русской локали:

```
// import java.util.Locale;
ResourceBundle bundle = ResourceBundle.getBundle(
 "bundles.MyApp", new Locale("ru", "RU"));
```

Теперь внутри программы мы можем получить значение по ключу. Для этого используется метод `getString()`. Если ключ не найден в файлах, то генерируется исключение. Формат метода:

```
public final String getString(String key)
```

Пример указания значения для текста кнопки:

```
Button btnClose = new Button(bundle.getString("btnClose"));
```

Для русской локали этот код будет преобразован к следующему виду:

```
Button btnClose = new Button("Закреть");
```

Для английской локали — к такому виду:

```
Button btnClose = new Button("Close");
```

Теперь пользователь увидит текст надписи на родном языке или на языке по умолчанию, если файл для его родного языка отсутствует в каталоге `bundles`.

### **ЭЛЕКТРОННЫЙ АРХИВ**

Полный текст программы содержится в проекте `FX_CH20_01_I18n`, приведенном в каталоге `CH20_other` сопровождающего книгу электронного архива (см. *приложение*).

При использовании FXML нужно передать ссылку на объект `ResourceBundle` с помощью конструктора класса или метода `setResources()` из класса `FXMLLoader`. Формат метода:

```
public void setResources(ResourceBundle resources)
```

Получить ссылку позволяет метод `getResources()`. Формат метода:

```
public ResourceBundle getResources()
```

Пример:

```
ResourceBundle bundle = ResourceBundle.getBundle(
 "bundles.MyApp", new Locale("ru", "RU"));
FXMLLoader loader = new FXMLLoader();
loader.setResources(bundle);
loader.setLocation(getClass().getResource("Sample.fxml"));
```

После этого можно внутри FXML-файла указывать значения в формате: `%<Ключ>`. Эти значения будут автоматически преобразованы в строки с учетом настроек объекта `ResourceBundle`. Пример:

```
<Button fx:id="btnExit" mnemonicParsing="false" text="%btnExit" />
```



Если мы реализуем интерфейс `Initializable`, то объект `ResourceBundle` будет доступен через второй параметр метода `initialize()`:

```
public void initialize(URL location, ResourceBundle resources)
```

**Пример:**

```
// <Button fx:id="btnClose" mnemonicParsing="false" text="Close" />
@Override
public void initialize(URL location, ResourceBundle resources) {
 btnClose.setText(resources.getString("btnClose"));
}
```

Если внутри класса контроллера нужна ссылка на объект `ResourceBundle`, то можно передать ее конструктору класса контроллера и сохранить в поле:

```
loader.setControllerFactory(new Callback<Class<?>, Object>() {
 @Override
 public Object call(Class<?> param) {
 return new SampleController(primaryStage, bundle);
 }
});
```

**Пример реализации конструктора:**

```
// private Stage window;
// private ResourceBundle bundle;
public SampleController(Stage window, ResourceBundle bundle) {
 this.bundle = bundle;
 this.window = window;
}
```

**Пример указания заголовка окна в методе `initialize()` контроллера:**

```
this.window.setTitle(this.bundle.getString("windowTitle"));
```

### **ЭЛЕКТРОННЫЙ АРХИВ**

Полный текст программы содержится в проекте `FX_CH20_01_I18n_FXML`, приведенном в каталоге `CH20_other` сопровождающего книгу электронного архива (см. приложение).

## **20.2. Настройка запуска приложения**

Когда приложение становится большим, его загрузка может занимать некоторое время. При этом пользователь может решить, что приложение не запустилось, и попытается запустить его еще раз. Чтобы сообщить пользователю о состоянии загрузки, обычно показывают стартовое окно с заставкой. Создать такую заставку можно тремя способами: указав в командной строке опцию `-splash`, добавив в файл манифеста JAR-архива директиву `SplashScreen-Image` или с помощью класса `Preloader`. Давайте рассмотрим эти способы подробно.

## 20.2.1. Опция командной строки **-splash**

Самый простой способ отобразить заставку с изображением — указать в командной строке при запуске приложения опцию `-splash`:

`-splash:<Путь к изображению>`

Пример:

```
java.exe -splash:photo.jpg --module-path C:\JavaSE\JavaFX\12\lib
--add-modules ALL-MODULE-PATH -jar FX_CH20_02_splash.jar
```

Внутри приложения после отображения основного окна нужно закрыть окно с заставкой. Получить ссылку на окно с заставкой позволяет статический метод `getSplashScreen()` из класса `SplashScreen`. Формат метода:

```
import java.awt.SplashScreen;
public static SplashScreen getSplashScreen()
```

Чтобы закрыть окно с заставкой, нужно вызвать метод `close()`:

```
public void close() throws IllegalStateException
```

Во время запуска приложения можно получить контекст рисования и что-нибудь нарисовать на заставке — например, полосу с индикацией текущего состояния загрузки. Для этого предназначен метод `createGraphics()`:

```
public Graphics2D createGraphics() throws IllegalStateException
```

Пример закрытия окна с заставкой:

```
SplashScreen splash = SplashScreen.getSplashScreen();
if (splash != null) splash.close();
```

### **ЭЛЕКТРОННЫЙ АРХИВ**

Полный текст программы содержится в проекте `FX_CH20_02_splash`, приведенном в каталоге `CH20_other` сопровождающего книгу электронного архива (см. *приложение*).

## 20.2.2. Директива ***SplashScreen-Image***

Если приложение упаковано в JAR-архив, то можно добавить изображение в корневую папку JAR-архива, а в файле манифеста прописать название файла с изображением в директиве `SplashScreen-Image`:

```
Manifest-Version: 1.0
Main-Class: application.Main
SplashScreen-Image: photo.jpg
```

Не забудьте в конец файла манифеста добавить пустую строку!

Теперь при двойном щелчке мышью на JAR-архиве сначала отобразится окно с заставкой, а затем — после загрузки — основное окно приложения. Внутри приложения после отображения основного окна нужно закрыть окно с заставкой.

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH20_02_SplashScreen_Image`, приведенном в каталоге `CH20_other` сопровождающего книгу электронного архива (см. *приложение*).

## 20.2.3. Класс *Preloader*

Предыдущие два способа используют класс `SplashScreen` из модуля `java.desktop`. В библиотеке `JavaFX` существует свой способ отображения окна с заставкой. Для этого нужно наследовать класс `Preloader` и внутри метода `start()` создать окно с заставкой. Инструкция импорта:

```
// Модуль javafx.graphics
import javafx.application.Preloader;
```

Иерархия наследования:

`Object` – `Application` – `Preloader`

Мы можем полностью контролировать процесс загрузки, переопределив в своем классе следующие методы из класса `Preloader`:

```
public void handleProgressNotification(ProgressNotification info)
public void handleStateChangeNotification(StateChangeNotification info)
public boolean handleErrorNotification(ErrorNotification info)
public void handleApplicationNotification(PreloaderNotification info)
```

С помощью метода `notifyPreloader()` из класса `Application` можно передавать данные методу `handleApplicationNotification()` объекта `Preloader`, сообщая о процессе загрузки. Формат метода:

```
public final void notifyPreloader(PreloaderNotification info)
```

Например, сообщим об окончании загрузки:

```
this.notifyPreloader(new Preloader.StateChangeNotification(null));
```

Внутри метода `handleApplicationNotification()` закроем окно с заставкой:

```
@Override
public void handleApplicationNotification(PreloaderNotification info) {
 if (info instanceof StateChangeNotification) {
 this.window.close();
 }
}
```

Чтобы сообщить приложению об окне с заставкой, нужно создать свойство с названием `javafx.preloader` и присвоить ему полное имя класса, наследующего класс `Preloader`. Далее передать статическому методу `launch()` из класса `Application` ссылку на класс с основным приложением и параметры, переданные в командной строке. Формат метода:

```
public static void launch(Class<? extends Application> appClass,
 String... args)
```

**Пример:**

```
public static void main(String[] args) {
 System.setProperty("javafx.preloader", "application.MyPreloader");
 Application.launch(Main.class, args);
}
```

Содержимое класса `MyPreloader` с окном-заставкой приведено в листинге 20.3, а содержимое класса `Main` с основным окном приложения — в листинге 20.4.

**Листинг 20.3. Содержимое файла `MyPreloader.java`**

```
package application;

import javafx.application.Application;
import javafx.application.Preloader;
import javafx.scene.Scene;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;
import javafx.stage.StageStyle;

public class MyPreloader extends Preloader {

 private Stage window;

 public static void main(String[] args) {
 System.setProperty("javafx.preloader", "application.MyPreloader");
 Application.launch(Main.class, args);
 }

 @Override
 public void start(Stage stage) throws Exception {
 Image image = null;
 try {
 image = new Image(getClass().getResourceAsStream("/photo.jpg"));
 if (image.isError()) new RuntimeException();
 } catch (Exception e) {
 System.err.println("Не удалось загрузить изображение");
 return;
 }
 window = stage;

 Scene scene = new Scene(new VBox(new ImageView(image)));
 stage.initStyle(StageStyle.UNDECORATED);
 stage.setTitle("Окно с заставкой");
 }
}
```

```

 stage.setScene(scene);
 stage.show();
 }

 @Override
 public void handleApplicationNotification(
 PreloaderNotification info) {
 if (info instanceof StateChangeNotification) {
 this.window.close();
 }
 }
}

```

#### Листинг 20.4. Содержимое файла Main.java

```

package application;

import javafx.application.Application;
import javafx.application.Preloader;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class Main extends Application {

 public void start(Stage stage) throws Exception {
 VBox root = new VBox(5.0, new Button("Нажми меня"));
 root.setAlignment(Pos.CENTER);

 Thread.sleep(5000); // Имитация длительной загрузки

 Scene scene = new Scene(root, 400.0, 200.0);
 stage.setTitle("Настройка запуска приложения");
 stage.setScene(scene);
 stage.show();

 this.notifyPreloader(new Preloader.StateChangeNotification(null));
 }
}

```

#### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH20_02_Preloader`, приведенном в каталоге `CH20_other` сопровождающего книгу электронного архива (см. приложение).

## 20.3. Взаимодействие между JavaFX и Swing

Библиотека JavaFX содержит два класса: `SwingNode` и `JFXPanel`, с помощью которых можно использовать компоненты из библиотеки Swing внутри JavaFX-приложения, и наоборот.

### 20.3.1. Класс *SwingNode*

Класс `SwingNode` позволяет использовать компоненты из библиотеки Swing внутри JavaFX-приложения. Инструкция импорта:

```
// Модуль javafx.swing
import javafx.embed.swing.SwingNode;
```

Иерархия наследования:

Object — Node — SwingNode

Конструктор класса:

```
SwingNode()
```

Добавить компонент позволяет метод `setContent()`. Формат метода:

```
public void setContent(JComponent content)
```

Получить ссылку на компонент можно с помощью метода `getContent()`. Формат метода:

```
public JComponent getContent()
```

Пример добавления кнопки (класс `JButton`) приведен в листинге 20.5.

#### Листинг 20.5. Класс *SwingNode*

```
package application;

import javax.swing.JButton;
import javax.swing.SwingUtilities;

import javafx.application.Application;
import javafx.embed.swing.SwingNode;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class Main extends Application {

 private JButton btn;

 public static void main(String[] args) {
 Application.launch(args);
 }
}
```

```

public void start(Stage stage) throws Exception {
 VBox root = new VBox(5.0);
 root.setAlignment(Pos.CENTER);

 SwingNode swingNode = new SwingNode();
 root.getChildren().add(swingNode);

 SwingUtilities.invokeLater(() -> {
 btn = new JButton("Нажми меня");
 btn.addActionListener(event -> {
 System.out.println("Нажата кнопка");
 });
 swingNode.setContent(btn);
 });

 Scene scene = new Scene(root, 350.0, 150.0);
 stage.setTitle("Класс SwingNode");
 stage.setScene(scene);
 stage.show();
}
}

```

### **ЭЛЕКТРОННЫЙ АРХИВ**

Полный текст программы содержится в проекте `FX_CH20_03_SwingNode`, приведенном в каталоге `CH20_other` сопровождающего книгу электронного архива (см. приложение).

## **20.3.2. Класс *JFXPanel***

Класс `JFXPanel` позволяет использовать компоненты JavaFX внутри Swing-приложения. Инструкция импорта:

```

// Модуль javafx.swing
import javafx.embed.swing.JFXPanel;

```

**Иерархия наследования:**

```

Object — java.awt.Component — java.awt.Container —
 javax.swing.JComponent — JFXPanel

```

**Конструктор класса:**

```

JFXPanel()

```

Для добавления компонентов следует воспользоваться методом `setScene()`, который принимает объект сцены. Формат метода:

```

public void setScene(Scene newScene)

```

Получить ссылку на объект сцены можно с помощью метода `getScene()`. Формат метода:

```

public Scene getScene()

```

Пример добавления кнопки (класс `Button`) приведен в листинге 20.6.

**Листинг 20.6. Класс `JFXPanel`**

```
package application;

import java.awt.Container;

import javax.swing.JFrame;
import javax.swing.SwingUtilities;
import javafx.application.Platform;
import javafx.embed.swing.JFXPanel;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.VBox;

public class Main {

 public void createUI() {
 JFrame window = new JFrame("Класс JFXPanel");
 window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
 window.setSize(400, 150);
 window.setLocationRelativeTo(null);

 JFXPanel panel = new JFXPanel();
 Platform.runLater(() -> {
 VBox root = new VBox(5.0);
 root.setAlignment(Pos.CENTER);

 Button btn = new Button("Нажми меня");
 root.getChildren().add(btn);
 btn.setOnAction(event -> {
 System.out.println("Нажата кнопка");
 });

 Scene scene = new Scene(root);
 panel.setScene(scene);
 });

 Container pane = window.getContentPane();
 pane.add(panel);

 window.setVisible(true);
 }
}
```



```

public static void main(String[] args) {
 SwingUtilities.invokeLater(() -> {
 Main app = new Main();
 app.createUI();
 });
}
}

```

### ЭЛЕКТРОННЫЙ АРХИВ

Полный текст программы содержится в проекте `FX_CH20_03_JFXPanel`, приведенном в каталоге `CH20_other` сопровождающего книгу электронного архива (см. приложение).

## 20.4. Утилита *jlink.exe*: создание дистрибутива

В предыдущих главах книги мы подключали JAR-архивы с модулями из состава JavaFX SDK. Однако имеется возможность создать дистрибутив, в состав которого входит JRE со всеми модулями, необходимыми для запуска приложения. Прежде чем создавать дистрибутив, нужно определить зависимость приложения от модулей. Для этого предназначена утилита `jdeps.exe`, входящая в состав JDK (расположена в каталоге `C:\OpenJDK\jdk-11\bin`). Запускаем командную строку и выполняем, например, следующую команду (JAR-архив `FX_CH01_01_Start.jar` с приложением расположен в каталоге `C:\book`):

```

C:\book>jdeps --module-path C:\JavaSE\JavaFX\12\lib -s
FX_CH01_01_Start.jar
FX_CH01_01_Start.jar -> java.base
FX_CH01_01_Start.jar -> javafx.controls
FX_CH01_01_Start.jar -> javafx.graphics

```

После определения необходимых модулей для создания дистрибутива можно воспользоваться утилитой `jlink.exe` (расположена в каталоге `C:\OpenJDK\jdk-11\bin`). Предварительно в каталоге `C:\book` создаем каталог `jlink`, в который и будем сохранять файлы. В командной строке выполняем следующую команду:

```

C:\book>jlink --no-header-files --no-man-pages --compress=2 --strip-debug
--module-path C:\JavaSE\JavaFX\12\jmods
--add-modules java.base,javafx.graphics,javafx.controls
--output C:\book\jlink\jrefx12

```

В этой команде мы указали следующие опции:

- ☐ `--no-header-files` — не добавлять заголовочные файлы;
- ☐ `--no-man-pages` — не добавлять файлы со справочной информацией;
- ☐ `--compress` — степень сжатия;
- ☐ `--strip-debug` — удалить отладочную информацию;
- ☐ `--module-path` — путь поиска модулей (обратите внимание: указывается путь к файлам `jmod`, а не к JAR-архивам с модулями из состава JavaFX SDK);

- `--add-modules` — добавляемые модули;
- `--output` — путь к каталогу, в который будут сохранены файлы дистрибутива (обратите внимание: каталог не должен существовать, он будет создан автоматически).

Для получения списка всех опций выполните следующую команду:

```
C:\book>jlink --help
```

После выполнения команды в каталоге `C:\book\jlink\jrefx12` были созданы все файлы и модули, необходимые для запуска приложения из JAR-архива `FX_CH01_01_Start.jar`. Выведем список всех доступных модулей:

```
C:\book>C:\book\jlink\jrefx12\bin\java --list-modules
java.base@11
java.datatransfer@11
java.desktop@11
java.prefs@11
java.xml@11
javafx.base
javafx.controls
javafx.graphics
jdk.unsupported@11
```

Как видно из результата, модули из библиотеки JavaFX были добавлены в дистрибутив, поэтому дополнительно подключать их больше нет необходимости. Давайте запустим приложение из командной строки:

```
C:\book>C:\book\jlink\jrefx12\bin\java -jar FX_CH01_01_Start.jar
```

Теперь попробуем создать ехе-файл, при щелчке на котором запустим виртуальную машину Java из дистрибутива и передадим ей путь к JAR-архиву. При этом сделаем так, чтобы черное окно при запуске не отображалось. Предварительно внутри каталога с дистрибутивом создаем каталог `jar` и копируем в него JAR-архив `FX_CH01_01_Start.jar`.

Нам понадобится библиотека MinGW-W64. Для загрузки библиотеки переходим на страницу <https://sourceforge.net/projects/mingw-w64/>, скачиваем файл `mingw-w64-install.exe` и запускаем процесс установки. Библиотеку устанавливаем в каталог `C:\MinGW64`. Запускаем командную строку и проверяем работоспособность:

```
C:\book>set Path=C:\MinGW64\mingw64\bin;%Path%
```

```
C:\book>gcc --version
```

```
gcc (x86_64-win32-seh-rev0, Built by MinGW-W64 project) 7.3.0
Copyright (C) 2017 Free Software Foundation, Inc.
```

```
This is free software; see the source for copying conditions. There is
NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR
PURPOSE.
```

Чтобы ехе-файл содержал значок, создадим файл ресурсов. Вначале в каталог C:\book добавляем файл со значком — например, my\_icon.ico. Затем создаем текстовый файл с названием resources.rc и вставляем в него следующую инструкцию:

```
001 ICON "my_icon.ico"
```

Компилируем файл ресурсов с помощью программы windres.exe:

```
C:\book>windres --use-temp-file -iresources.rc -oresources.o
```

В результате в каталоге C:\book будет создан файл resources.o, который мы должны передать компилятору при сборке программы. Текст программы приведен в листинге 20.7. Файл с названием runjar.c нужно сохранить в каталоге C:\book.

#### Листинг 20.7. Содержимое файла runjar.c

```
#include <stdio.h>
#include <process.h>

int main(int argc, char *argv[], char **penv) {
 _execl("bin\\javaw.exe", "bin\\javaw.exe",
 "-jar", "jar\FX_CH01_01_Start.jar", NULL);
 return 0;
}
```

Теперь можно создать ехе-файл, выполнив следующую команду:

```
C:\book>gcc -Wall -O3 -mwindows
 -o C:\book\jlink\jrefx12\MyApp.exe runjar.c resources.o
```

После компиляции в каталоге C:\book\jlink\jrefx12 будет создан файл MyApp.exe. При двойном щелчке левой кнопкой мыши на значке файла запустится JavaFX-приложение из JAR-архива FX\_CH01\_01\_Start.jar. В таком виде дистрибутив приложения можно передать пользователю, и ему не понадобится дополнительно устанавливать JDK и библиотеку JavaFX. При этом размер дистрибутива в моем случае составил 45,3 Мбайт без сжатия и 31,2 Мбайт в ZIP-архиве.

## Заключение

Вот и закончилось наше путешествие в мир JavaFX. Материал книги описывает лишь базовые возможности этой замечательной технологии. А здесь мы уточним, где найти дополнительную информацию.

Самым важным источником информации является официальный сайт библиотеки JavaFX: <https://openjfx.io/>. На этом сайте вы найдете новости, а также ссылки на все другие ресурсы в Интернете, посвященные JavaFX. Не следует также забывать о существовании страницы: <https://gluonhq.com/products/javafx/>, на которой доступна для скачивания самая последняя версия библиотеки JavaFX.

На странице [<Версия>/](https://openjfx.io/javadoc/) расположена документация по JavaFX, которая обновляется в режиме реального времени. Библиотека постоянно совершенствуется, появляются новые классы, изменяются параметры, добавляются пакеты и т. д. Регулярно посещайте эту страницу, и вы будете в курсе самых свежих новшеств JavaFX.

На странице Википедии <https://ru.wikipedia.org/wiki/JavaFX> вы найдете историю развития библиотеки JavaFX, а также действующие ссылки на все другие ресурсы в Интернете, посвященные этой библиотеке.

Библиотека JavaFX позволяет создавать не только оконные, но и мобильные приложения под Android и iOS. Подробную информацию о создании мобильных приложений можно найти на странице: <https://gluonhq.com/products/mobile/>.

Если в процессе изучения JavaFX у вас возникнут какие-либо недопонимания, то не следует забывать, что Интернет предоставляет множество ответов на самые разнообразные вопросы. Достаточно в строке запроса поискового портала (например, <http://www.google.com/>) набрать свой вопрос. Наверняка уже кто-то сталкивался с подобной проблемой и описал ее решение на каком-либо сайте.

Свои замечания и пожелания вы можете оставить на странице книги на сайте издательства «БХВ-Петербург»: <http://www.bhv.ru/>. Все замеченные опечатки и неточности прошу присылать на E-mail: [mail@bhv.ru](mailto:mail@bhv.ru) — не забудьте только указать название книги и имя автора.

# ПРИЛОЖЕНИЕ

## Описание электронного архива

Электронный архив к книге выложен на FTP-сервер издательства по адресу: <ftp://ftp.bhv.ru/9785977540728.zip>. Ссылка доступна и со страницы книги на сайте [www.bhv.ru](http://www.bhv.ru).

Структура архива представлена в табл. П.1.

*Таблица П.1. Структура электронного архива*

Файл или каталог	Описание
Listings.doc	Содержит все пронумерованные листинги из книги, а также некоторые полезные фрагменты кода
Readme.txt	Описание электронного архива
projects	Содержит вложенные каталоги с проектами. Каждый проект состоит из двух файлов: JAR-архив и BAT-файл. JAR-архив содержит скомпилированные файлы с классами, файл манифеста, необходимые ресурсы (например, изображения), а также файлы с исходным кодом классов. Чтобы посмотреть исходный код примеров, достаточно распаковать JAR-архив с помощью любого архиватора. BAT-файл содержит код для запуска JAR-архива в Windows с отображением окна консоли, в которое будет выводиться отладочная информация. Запустить файл можно двойным щелчком левой кнопкой мыши на значке BAT-файла. Предварительно нужно указать в файле <code>projects\path_to_JRE.txt</code> полный путь к JRE, а в файле <code>projects\path_to_JavaFX.txt</code> — полный путь до библиотеки JavaFX. В составе пути следует избегать пробелов и русских букв, а в файлах должна быть только одна строка без символов перевода строки, иначе получите ошибку

# Предметный указатель

## A

acceptTransferModes() 203, 206, 207  
Accordion 156  
ADD 237  
addEventFilter() 161, 166  
addEventHandler() 161  
addListener() 162, 417, 418  
addMnemonic() 186  
Affine 324  
Alert 92, 177, 664  
AlertType 665  
ALL\_DESCENDANTS 616  
ALL\_TABS 154  
ALL-DEFAULT 25  
ALL-MODULE-PATH 25  
ALL-SYSTEM 25  
ALWAYS 121, 124  
AmbientLight 406  
AnchorLocation 645  
AnchorPane 147  
Animation 362, 365  
Animation.Status 363  
AnimationTimer 171, 387  
Application 48, 64, 66, 714, 744  
APPLICATION\_MODAL 85  
Arc 245  
ArcTo 275  
ArcType 247, 308  
AreaChart 694  
AudioClip 699, 700  
AudioTrack 710  
auto 733  
Axis 679, 687

## B

BACK 394  
Background 105

BackgroundFill 105, 107  
BackgroundImage 106, 109  
BackgroundPosition 109, 110  
BackgroundRepeat 109  
BackgroundSize 110  
BarChart 689, 690  
BASELINE 260  
BASELINE\_CENTER 120  
BASELINE\_LEFT 121  
BASELINE\_OFFSET\_SAME\_AS\_HEIGHT 260  
BASELINE\_RIGHT 121  
BEVEL 243  
bind() 421  
bindBidirectional() 420  
BLACK 263  
Blend 352  
BlendMode 237, 352  
Bloom 342  
BLUE 237  
BlurType 334, 336, 337  
bold 724  
BOLD 263  
bolder 724  
BooleanExpression 422  
BooleanProperty 415  
BooleanStringConverter 420  
Border 111  
BorderImage 111  
BorderPane 48, 140, 624  
BorderStroke 111, 112  
BorderStrokeStyle 112  
BorderWidths 112  
BOTTOM 260, 431  
BOTTOM\_CENTER 121  
BOTTOM\_LEFT 121  
BOTTOM\_RIGHT 121  
BoundingBox 221

Bounds 221  
 Box 394  
 BoxBlur 341  
 brighter() 229  
 BubbleChart 693  
 BufferedImage 294, 295  
 BUTT 242  
 Button 435, 436  
 BUTTON\_ORDER\_LINUX 438  
 BUTTON\_ORDER\_MAC\_OS 438  
 BUTTON\_ORDER\_NONE 438  
 BUTTON\_ORDER\_WINDOWS 438  
 ButtonBar 437  
 ButtonBase 433  
 ButtonData 438, 662  
 ButtonType 662  
 BYTE\_BGRA 289  
 BYTE\_BGRA\_PRE 289  
 BYTE\_INDEXED 289  
 BYTE\_RGB 289

## C

Camera 403  
 CANCELLED 456  
 Canvas 301, 302, 409  
 CASPIAN 715  
 CategoryAxis 686, 687  
 CellDataFeatures 557, 601  
 CellEditEvent 577, 619  
 CENTER 110, 121, 260, 264, 431  
 CENTER\_ELLIPSIS 429  
 CENTER\_LEFT 121  
 CENTER\_RIGHT 121  
 CENTER\_WORD\_ELLIPSIS 429  
 CENTERED 241  
 centerOnScreen() 80  
 changed() 418  
 ChangeListener 418  
 Chart 675  
 CheckBox 444  
 CheckBoxTreeCell 594  
 CheckBoxTreeItem 592  
 checkedObservableList() 514  
 CheckMenuItem 634  
 ChoiceBox 522  
 ChoiceDialog 668  
 CHORD 247  
 Circle 255  
 CLIP 429  
 Clipboard 197  
 ClipboardContent 199, 200

close() 91  
 CLOSED\_HAND 196  
 ClosePath 272  
 Color 96, 226, 228, 240, 303  
 color() 226  
 COLOR\_BURN 237  
 COLOR\_DODGE 237  
 ColorAdjust 346  
 ColorInput 354  
 ColorPicker 535  
 ColumnConstraints 127, 131, 135  
 ComboBox 530  
 ComboBoxBase 527  
 CONFIRMATION 665  
 CONSTRAINED\_RESIZE\_POLICY 562, 605  
 consume() 91, 162, 166, 167, 169, 176  
 contain 733  
 CONTENT\_BOTTOM\_LEFT 645  
 CONTENT\_BOTTOM\_RIGHT 645  
 CONTENT\_TOP\_LEFT 645  
 CONTENT\_TOP\_RIGHT 645  
 ContentDisplay 431  
 CONTEXT\_MENU\_REQUESTED 653  
 ContextMenu 623, 652  
 ContextMenuEvent 653  
 Control 623, 649, 652  
 COPY 207  
 CornerRadii 107, 108  
 cover 733  
 createGraphics() 743  
 createScreenCapture() 298  
 CROSSHAIR 196  
 CSS 713  
 CubicCurve 249  
 CubicCurveTo 276  
 cursive 723  
 Cursor 196  
 CustomMenuItem 637  
 CycleMethod 231, 234  
 Cylinder 396

## D

DARKEN 237  
 darker() 229  
 DateFormat 420  
 DatePicker 537, 538  
 DateTimeStringConverter 420  
 DECORATED 75  
 DEFAULT 110, 196  
 deriveColor() 230  
 desaturate() 229

Dialog 71, 655, 658, 664  
DialogPane 660, 661  
DIFFERENCE 237  
Dimension2D 219  
DirectoryChooser 669, 670  
DISAPPEAR 196  
DISCRETE 367  
DisplacementMap 356  
DISPOSED 704  
Distant 349  
DoubleExpression 422, 424  
DoubleProperty 415  
DoubleSpinnerValueFactory 475  
DoubleStringConverter 420  
drag & drop 202  
DRAG\_DETECTED 189, 205, 209  
DRAG\_DONE 204, 206  
DRAG\_DROPPED 204, 206  
DRAG\_ENTERED 203  
DRAG\_ENTERED\_TARGET 204  
DRAG\_EXITED 203  
DRAG\_EXITED\_TARGET 204  
DRAG\_OVER 203, 206  
Dragboard 205, 208, 209  
DragEvent 202, 207  
DrawMode 393  
DropShadow 333  
dropshadow() 734  
Duration 359, 360

## E

e(fx)clipse 34, 49  
E\_RESIZE 196  
EASE\_BOTH 367  
EASE\_IN 367  
EASE\_OUT 367  
Eclipse 27  
EditEvent 591  
Effect 333  
Ellipse 254  
ELLIPSIS 429  
EMPTY 107, 108, 113, 219  
emptyObservableList() 513  
ERROR 665  
EVEN\_ODD 270  
Event 167, 168  
EventHandler 64, 162  
EventTarget 168  
EXCLUSION 237  
ExecutorService 462  
exit() 68

exports 55  
EXTRA\_BOLD 263  
EXTRA\_LIGHT 263

## F

FadeTransition 368  
FAILED 456  
fantasy 723  
FileChooser 671, 672  
FILL 393  
FillRule 270, 309  
FillTransition 377  
fireEvent() 167–169  
FIXED 154  
FloatExpression 422  
FloatMap 356, 357  
FloatProperty 415  
FloatStringConverter 420  
FlowPane 138  
FocusModel 545, 568, 589, 611  
Font 262, 263, 313, 428  
font() 263  
FontPosture 263  
FontSmoothingType 314  
FontWeight 263  
Format 420  
fromFXImage() 294  
FRONT 394  
Future 452  
-fx-alignment 726  
-fx-arc-height 736  
-fx-arc-width 736  
-fx-background-color 88, 105, 730  
-fx-background-image 89, 731  
-fx-background-insets 731  
-fx-background-position 732  
-fx-background-radius 731  
-fx-background-repeat 89, 731  
-fx-background-size 732  
-fx-blend-mode 734  
-fx-block-increment 737  
-fx-border-color 729  
-fx-border-insets 730  
-fx-border-radius 729  
-fx-border-style 727  
-fx-border-width 729  
FXCollections 512, 520  
-fx-content-display 726  
-fx-cursor 735  
-fx-effect 734  
-fx-ellipsis-string 725



- fx-fill 735
- fx-fit-to-height 738
- fx-fit-to-width 738
- fx-font 724
- fx-font-family 723
- fx-font-size 723
- fx-font-style 723
- fx-font-weight 723
- fx-graphic 726
- fx-graphic-text-gap 726
- fx-hbar-policy 738
- fx-max-height 727
- fx-max-width 727
- fx-min-height 727
- fx-min-width 727
- FXML 23, 52, 55, 56, 741
- FXMLLoader 53, 59, 741
- fx-opacity 733
- fx-orientation 737
- fx-padding 727
- fx-pannable 738
- fx-pref-height 727
- fx-pref-width 727
- fx-rotate 734
- fx-scale-x 734
- fx-scale-y 734
- fx-scale-z 734
- fx-smooth 736
- fx-strikethrough 725
- fx-stroke 735
- fx-stroke-dash-array 735
- fx-stroke-dash-offset 735
- fx-stroke-line-cap 736
- fx-stroke-line-join 736
- fx-stroke-miter-limit 736
- fx-stroke-type 735
- fx-stroke-width 735
- fx-text-alignment 725
- fx-text-fill 723
- fx-text-overflow 725
- fx-translate-x 733
- fx-translate-y 733
- fx-translate-z 733
- fx-underline 725
- fx-unit-increment 737
- fx-vbar-policy 738
- fx-wrap-text 725

## G

- GAUSSIAN 334
- GaussianBlur 339
- getAccelerators() 187, 634

- getAlignment() 121
- getBackground() 105
- getBlue() 228
- getBorder() 111
- getBounds() 81
- getBrightness() 228
- getBundle() 740
- getChildren() 117, 118, 123, 140, 143, 148, 412
- getClip() 238
- getController() 59
- getCursor() 196
- getDefault() 262
- getDpi() 81
- getDragboard() 208
- getEffect() 333
- getEventType() 168
- getFamilies() 262
- getFamily() 263
- getFill() 96, 239, 303
- getFocusOwner() 184
- getFontNames() 262
- getFullScreenExitHint() 82
- getFullScreenExitKeyCombination() 83
- getGreen() 228
- getHeight() 78, 95, 99
- getHue() 228
- getIcons() 87
- getId() 410
- getLayoutX() 102
- getLayoutY() 102
- getMaxHeight() 78, 99
- getMaxWidth() 78, 99
- getMinHeight() 78, 99
- getMinWidth() 78, 99
- getMnemonics() 186
- getModality() 86
- getName() 263
- getOpacity() 84, 113, 229
- getOwner() 86
- getPadding() 100
- getParameters() 66
- getParent() 413
- getPixelColor() 298
- getPrefHeight() 99
- getPrefWidth() 99
- getPrimary() 81
- getRed() 228
- getResource() 53, 87
- getResourceAsStream() 87
- getResources() 741
- getRoot() 94, 412
- getSaturation() 228

getScene() 95, 411  
getScreenCapture() 299  
getScreens() 81  
getSize() 264  
getSource() 168  
getSplashScreen() 743  
getString() 741  
getStyle() 264  
getStyleClass() 715  
getStylesheets() 88, 105, 714  
getSystemClipboard() 197  
getTarget() 169, 411  
getTitle() 73  
getTransferMode() 206, 207  
getTranslateX() 103  
getTranslateY() 103  
getTypeSelector() 715  
getUserAgentStylesheet() 714  
getUserData() 410  
getValue() 418  
getVisualBounds() 81  
getWidth() 78, 95, 99  
getWindow() 95, 411  
getWindows() 73  
getX() 80, 95  
getY() 80, 95  
Glow 343  
GRAPHIC\_ONLY 431  
GraphicsContext 302, 314  
gray() 227  
grayRgb() 228  
grayscale() 229  
GREEN 237  
GridPane 125  
Group 102, 116

## H

H\_RESIZE 196  
HALTED 704  
HAND 196  
handle() 64, 162  
handleApplicationNotification() 744  
HARD\_LIGHT 237  
HBox 119  
hide() 74  
HitInfo 267  
HLineTo 273  
HORIZONTAL 138  
HPos 128  
hsb() 227  
HTML 494

HTMLEditor 493, 494  
Hyperlink 434

## I

IDENTITY\_STRING\_CONVERTER 487  
Image 87, 283, 286, 291, 294  
ImageCursor 196  
ImageInput 355  
ImageIO 295  
ImagePattern 96, 240, 296, 303  
ImageView 238, 286, 291, 409  
INDEFINITE 360  
INFORMATION 665  
init() 64  
Initializable 62, 742  
initialize() 60, 62, 742  
initModality() 85  
initOwner() 86  
initStyle() 75  
InnerShadow 335  
innershadow() 734  
Insets 100  
INSIDE 241  
INT\_ARGB 289  
INT\_ARGB\_PRE 289  
IntegerExpression 422  
IntegerProperty 415  
IntegerSpinnerValueFactory 475  
IntegerStringConverter 420  
interpolate() 230  
Interpolator 367  
invalidated() 417  
InvalidationListener 417, 516  
invert() 229  
isAccepted() 206, 207  
isAlwaysOnTop() 84  
isBound() 421  
isConsumed() 169  
isDisable() 115  
isDisabled() 115  
isFocused() 83, 184  
isFocusTraversable() 184  
isFullScreen() 82  
isIconified() 82  
isMaximized() 82  
isMnemonicParsing() 185  
isMouseTransparent() 192  
isResizable() 79  
isVisible() 114  
italic 723  
ITALIC 263

**J**

Java Development Kit 19  
Java Runtime Environment 19  
JAVA\_HOME 20  
javac.exe 19  
JavaFX 20  
JavaFX SDK 21  
javafx.base 22  
javafx.controls 22  
javafx.fxml 23  
javafx.graphics 22  
javafx.media 22  
javafx.swing 23  
javafx.web 23  
JavaFX-свойства 414  
JavaScript 501  
jdeps.exe 750  
JDK 19  
JFXPanel 748  
jlink.exe 750  
JRE 19  
JSObject 506  
JUSTIFY 264

**K**

KEY\_PRESSED 179  
KEY\_RELEASED 179  
KEY\_TYPED 179  
KeyCode 181  
KeyEvent 178, 179, 180  
KeyFrame 384  
KeyValue 386

**L**

Label 427, 473  
Labeled 426, 428  
large 723  
larger 723  
launch() 48, 66, 744  
LEADING\_ELLIPSIS 429  
LEADING\_WORD\_ELLIPSIS 429  
LEFT 264, 431  
Light 349  
LIGHT 263  
Light.Distant 350  
Light.Point 350, 351  
Light.Spot 351  
LightBase 405  
LIGHTEN 237

lighter 724  
Lighting 348  
Line 244  
LINE 393  
LINEAR 367  
LinearGradient 96, 230, 232, 240, 303  
LineChart 687  
LineTo 274  
ListChangeListener 517  
ListSpinnerValueFactory 475  
ListView 540, 542  
load() 53  
loadFont() 263  
loadFonts() 263  
LocalDateTimeStringConverter 420  
LocalTimeStringConverter 420  
localToParent() 103  
localToScene() 103  
localToScreen() 104  
LOGICAL 260  
LOGICAL\_VERTICAL\_CENTER 260  
LongExpression 422  
LongProperty 415  
LongStringConverter 420  
lookup() 413  
lookupAll() 414, 468

**M**

main() 48  
MatrixType 324  
Media 699, 702, 703  
MediaException 702  
MediaPlayer 699, 703, 705, 707  
MediaPlayer.Status 704  
MediaView 699, 708, 709  
medium 723  
MEDIUM 263  
Menu 623, 625, 627  
MENU\_VALIDATION\_EVENT 634  
MenuBar 623, 624  
MenuItem 638, 639, 641  
MenuItem 187, 623, 626–630, 633, 636, 637  
Mesh 399  
MeshView 398  
MessageFormat 420  
MinGW-W64 751  
MITER 243  
Mnemonic 186  
Modality 85, 657  
MODENA 715  
Modulepath 41  
monospace 723

MotionBlur 340  
MOUSE\_CLICKED 189  
MOUSE\_DRAG\_ENTERED 210  
MOUSE\_DRAG\_ENTERED\_TARGET 210  
MOUSE\_DRAG\_EXITED 210  
MOUSE\_DRAG\_EXITED\_TARGET 210  
MOUSE\_DRAG\_OVER 211  
MOUSE\_DRAG\_RELEASED 211  
MOUSE\_DRAGGED 189  
MOUSE\_ENTERED 190  
MOUSE\_ENTERED\_TARGET 190  
MOUSE\_EXITED 190  
MOUSE\_EXITED\_TARGET 191  
MOUSE\_MOVED 189  
MOUSE\_PRESSED 188  
MOUSE\_RELEASED 188  
MouseDownEvent 209, 210  
MouseEvent 188, 192  
MOVE 196, 207  
MoveTo 272  
MT\_2D\_2x3 324  
MT\_2D\_3x3 324  
MT\_3D\_3x4 324  
MT\_3D\_4x4 324  
MultipleSelectionModel 549, 550, 565, 589, 608  
MULTIPLY 237

## N

N\_RESIZE 196  
NE\_RESIZE 196  
NEVER 121, 124  
NO\_CYCLE 231  
NO\_REPEAT 109  
Node 97, 102, 103, 113, 114, 115, 184, 224, 236, 238, 329, 409, 410, 411, 715  
NON\_ZERO 270  
NONE 85  
normal 723, 724  
NORMAL 263  
notifyPreloader() 744  
NULL\_SOURCE\_TARGET 169  
NumberAxis 681, 682  
NumberExpressionBase 424  
NumberFormat 420  
NumberStringConverter 420  
NW\_RESIZE 196

## O

ObjectExpression 422  
ObjectProperty 415

oblique 723  
Observable 417, 418  
observableArrayList() 512  
ObservableList 511, 514  
observableList() 513  
ObservableValue 418  
ON\_HIDDEN 527  
ON\_HIDING 527  
ON\_SHOWING 527  
ON\_SHOWN 527  
ONE 360  
ONE\_PASS\_BOX 334  
ONLY\_FIRST\_LEVEL 616  
OPEN 247  
OPEN\_HAND 196  
OpenJDK 19  
opens 23, 55  
Orientation 138, 145, 157, 470  
OrientationType 372  
ORTHOGONAL\_TO\_TANGENT 372  
OUTSIDE 241  
OVERLAY 237  
OverrunStyle 429

## P

Paint 225  
Pane 102, 116, 118  
ParallelCamera 404  
ParallelTransition 380  
Parent 97, 409, 714  
parentToLocal() 104  
PasswordField 491  
Path 269  
PATH 19  
PathElement 271  
PathTransition 372  
PAUSED 363, 704  
PauseTransition 382  
PerspectiveCamera 404  
PerspectiveTransform 344  
PhongMaterial 393, 401  
PickResult 654  
PieChart 676  
PieChart.Data 677  
PixelFormat 289  
PixelReader 288, 289  
PixelWriter 292  
Platform 68, 70, 172, 451  
PLAYING 704  
Point 349  
POINT\_NORMAL\_TEXCOORD 399

POINT\_TEXCOORD 399  
 Point2D 215, 216  
 Point3D 323, 389  
 PointLight 406  
 Polygon 257  
 Polyline 248  
 Popup 646  
 PopupControl 647  
 PopupWindow 71, 642, 643  
 Pos 120, 124, 136, 139, 142, 146, 431, 490  
 Preloader 744  
 Priority 121, 124, 128, 130  
 ProgressBar 451  
 ProgressIndicator 449  
 Properties 739  
 Property 420, 421  
 PropertyValueFactory 556, 558

## Q

QuadCurve 251  
 QuadCurveTo 277

## R

RadialGradient 96, 233, 240, 303  
 RadioButton 441  
 RadioMenuItem 635, 636  
 ReadOnlyBooleanProperty 416  
 ReadOnlyBooleanWrapper 416  
 ReadOnlyDoubleProperty 416  
 ReadOnlyDoubleWrapper 416  
 ReadOnlyFloatProperty 416  
 ReadOnlyFloatWrapper 416  
 ReadOnlyIntegerProperty 416  
 ReadOnlyIntegerWrapper 416  
 ReadOnlyLongProperty 416  
 ReadOnlyLongWrapper 416  
 ReadOnlyObjectProperty 416  
 ReadOnlyObjectWrapper 416, 558  
 ReadOnlyStringProperty 416  
 ReadOnlyStringWrapper 416  
 READY 456, 704  
 Rectangle 252  
 Rectangle2D 219  
 RED 237  
 REFLECT 231  
 Reflection 338  
 Region 97, 100, 111, 409, 727  
 REGULAR 263  
 relocate() 102  
 removeEventFilter() 165

removeEventHandler() 165  
 removeListener() 166, 417, 418  
 removeMnemonic() 186  
 REORDER 154  
 REPEAT 109, 231  
 requestFocus() 83, 184  
 ResourceBundle 740, 741  
 rgb() 226  
 RIGHT 264, 431  
 Robot 211, 298  
 Rotate 323, 329  
 RotateTransition 375  
 ROUND 109, 242, 243, 247  
 RowConstraints 129, 131, 135  
 runLater() 70, 172, 451  
 RunnableFuture 452  
 RUNNING 363, 456

## S

S\_RESIZE 196  
 sans-serif 723  
 saturate() 229  
 Scale 328  
 ScaleTransition 373  
 ScatterChart 697  
 Scene 49, 60, 93, 409, 714  
 Scene Builder 36, 52  
 sceneToLocal() 104  
 SCHEDULED 456  
 ScheduledService 462  
 Screen 80  
 SCREEN 237  
 screenToLocal() 104  
 SCROLL 194  
 SCROLL\_FINISHED 194  
 SCROLL\_STARTED 194  
 ScrollBar 462  
 ScrollBarPolicy 466  
 ScrollEvent 194  
 ScrollPane 465, 468, 738  
 SE\_RESIZE 196  
 SELECTED\_TAB 154  
 SelectionModel 525, 550, 565, 608  
 SEMI\_BOLD 263  
 Separator 470  
 SeparatorMenuItem 636  
 SepiaTone 347  
 SequentialTransition 381  
 Series 684  
 serif 723  
 Service 460

- setAccelerator() 187, 633
- setAlignment() 120
- setAlwaysOnTop() 83
- setBackground() 105
- setBlendMode() 237
- setBorder() 111
- setBottom() 141
- setCenter() 142
- setClip() 238
- setContextMenu() 623, 652
- setCursor() 196
- setDisable() 115, 165
- setDropCompleted() 204, 207
- setEffect() 333
- setFill() 89, 96, 225, 239, 303
- setFocusTraversable() 184
- setFullScreen() 82
- setFullScreenExitHint() 82
- setFullScreenExitKeyCombination() 82
- setHeight() 77
- setIconified() 81
- setId() 410
- setLabelFor() 186, 432
- setLayoutX() 102
- setLayoutY() 102
- setLeft() 141
- setMaxHeight() 78, 98
- setMaximized() 82
- setMaxSize() 98
- setMaxWidth() 78, 98
- setMinHeight() 78, 97
- setMinSize() 98
- setMinWidth() 77, 97
- setMnemonicParsing() 185
- setMouseTransparent() 191
- setOnAction() 64, 161
- setOnCloseRequest() 77, 91
- setOpacity() 84, 113
- setPadding() 100
- setPrefHeight() 97
- setPrefSize() 97
- setPrefWidth() 97
- setResizable() 79
- setResources() 741
- setRight() 141
- setRoot() 94, 412
- setRotate() 331
- setScaleX() 329
- setScaleY() 329
- setScaleZ() 329
- setScene() 94
- setStyle() 88, 105, 713
- setTitle() 73
- setTooltip() 649
- setTop() 141
- setTranslateX() 102, 322, 327
- setTranslateY() 102, 322, 327
- setTranslateZ() 327
- setUserAgentStylesheet() 714
- setUserData() 410
- setVisible() 114
- setWidth() 77
- setX() 79
- setY() 79
- Shadow 337
- Shape 238, 239, 240, 243
- Shape3D 393
- Shear 331
- show() 73
- showAndWait() 74
- Side 109, 675, 676
- SimpleBooleanProperty 415
- SimpleDoubleProperty 415
- SimpleFloatProperty 415
- SimpleIntegerProperty 415
- SimpleLongProperty 415
- SimpleObjectProperty 415
- SimpleStringProperty 415, 424
- SingleSelectionModel 525
- singletonObservableList() 513
- sizeToScene() 78
- sleep() 69
- Slider 446
- small 723
- smaller 723
- snapshot() 297
- SOFT\_LIGHT 237
- SOMETIMES 121, 124
- SPACE 109
- Sphere 397
- Spinner 473
- SpinnerValueFactory 474
- splash 743
- SplashScreen 743
- SplashScreen-Image 743
- SPLINE() 367
- SplitMenuButton 641
- SplitPane 157
- Spot 349
- SQUARE 242
- SRC\_ATOP 237
- SRC\_OVER 237
- StackedAreaChart 696
- StackedBarChart 691

StackPane 143  
 Stage 48, 49, 60, 71, 73, 409  
 StageStyle 75  
 STALLED 704  
 start() 48, 64  
 startDragAndDrop() 205, 209  
 startFullDrag() 209  
 Stop 232, 234  
 stop() 64  
 STOPPED 363, 704  
 stretch 733  
 StringConverter 420  
 StringExpression 422, 425  
 StringProperty 415, 425  
 StrokeLineCap 242, 305  
 StrokeLineJoin 243, 306  
 StrokeTransition 378  
 StrokeType 241  
 STYLE\_CLASS\_ARROWS\_ON\_LEFT\_  
   HORIZONTAL 475  
 STYLE\_CLASS\_ARROWS\_ON\_LEFT\_  
   VERTICAL 475  
 STYLE\_CLASS\_ARROWS\_ON\_RIGHT\_  
   HORIZONTAL 475  
 STYLE\_CLASS\_SPLIT\_ARROWS\_  
   HORIZONTAL 475  
 STYLE\_CLASS\_SPLIT\_ARROWS\_  
   VERTICAL 475  
 SubScene 402, 714  
 SubtitleTrack 710  
 SUCCEEDED 456  
 SVG 278  
 SVGPath 278, 279  
 SW\_RESIZE 196  
 Swing 23, 747, 748  
 SwingFXUtils 294  
 SwingNode 747  
 synchronizedObservableList() 513

## T

Tab 151  
 TabClosingPolicy 154  
 TableCell 570  
 TableColumn 555, 556, 559, 570, 576  
 TableColumnBase 556, 560, 562, 567, 576,  
   579, 599, 603, 605, 621  
 TablePosition 567  
 TableRow 571  
 TableSelectionModel 569, 608  
 TableView 553, 561, 563, 571, 572  
 TableViewFocusModel 568

TableViewSelectionModel 565  
 TabPane 150, 153  
 TANGENT() 367  
 Task 452, 456, 460  
 Text 258, 260, 264, 473, 725  
 TEXT 196  
 TEXT\_ONLY 431  
 TextAlignment 264, 268, 314, 430  
 TextArea 491, 492  
 TextBoundsType 260  
 TextField 489, 490  
 TextFieldListCell 546  
 TextFieldTableCell 575  
 TextFieldTreeCell 590  
 TextFieldTreeTableCell 618  
 TextFlow 267, 268  
 TextFormatter 486  
 TextInputControl 447, 478–480, 485  
 TextInputDialog 667  
 THIN 263  
 Thread 69  
 THREE\_PASS\_BOX 334  
 TickMark 680  
 TilePane 144  
 Timeline 383  
 Timer 171  
 TimerTask 172  
 TitledPane 154  
 toBack() 83, 236  
 toFront() 83, 236  
 toFXImage() 295  
 Toggle 440, 441  
 ToggleButton 440  
 ToggleGroup 442, 443, 636  
 ToolBar 469  
 Tooltip 642, 649, 650  
 TOP 260, 431  
 TOP\_CENTER 121  
 TOP\_LEFT 121  
 TOP\_RIGHT 121  
 Track 710, 711  
 TransferMode 207  
 Transform 323  
 Transition 171, 366  
 Translate 326  
 TranslateTransition 369  
 TRANSPARENT 76, 90, 226  
 TreeItem 581, 583, 584  
 TreeItemPropertyValueFactory 600  
 TreeModificationEvent 586, 595  
 TreeSortMode 616  
 TreeTableCell 613, 614

TreeTableColumn 599, 600, 602, 610, 613  
TreeTablePosition 610  
TreeTableRow 614  
TreeTableView 595, 598, 603, 606, 614–617  
TreeTableViewFocusModel 611  
TreeTableViewSelectionModel 608  
TreeView 579, 580, 587, 589–592  
TriangleMesh 399, 401  
TWO\_PASS\_BOX 334

## U

UNAVAILABLE 154  
unbind() 421  
unbindBidirectional() 420  
UNCONSTRAINED\_RESIZE\_POLICY 561, 604  
UNDECORATED 76  
UNIFIED 76  
UNKNOWN 360, 704  
unmodifiableObservableList() 513  
USE\_COMPUTED\_SIZE 98, 648  
USE\_PREF\_SIZE 98, 648  
UTILITY 76

## V

V\_RESIZE 196  
ValueAxis 682  
valueOf() 227  
VBox 122  
VertexFormat 399  
VERTICAL 138  
VideoTrack 710, 711  
VISUAL 260  
VLineTo 273  
VPos 130, 259, 314

## W

W\_RESIZE 196  
WAIT 196  
WARNING 665  
web() 227  
WebEngine 497, 498, 500, 501  
WebHistory 508

WebView 495  
Window 60, 71, 73, 644  
WINDOW\_BOTTOM\_LEFT 645  
WINDOW\_BOTTOM\_RIGHT 645  
WINDOW\_CLOSE\_REQUEST 175, 176  
WINDOW\_HIDDEN 175  
WINDOW\_HIDING 175  
WINDOW\_MODAL 85  
WINDOW\_SHOWING 174  
WINDOW\_SHOWN 175  
WINDOW\_TOP\_LEFT 645  
WINDOW\_TOP\_RIGHT 645  
WindowEvent 91, 174  
windres.exe 752  
WORD\_ELLIPSIS 429  
Worker 456, 460  
Worker.State 456  
WORKER\_STATE\_CANCELLED 459  
WORKER\_STATE\_FAILED 459  
WORKER\_STATE\_READY 458  
WORKER\_STATE\_RUNNING 458  
WORKER\_STATE\_SCHEDULED 458  
WORKER\_STATE\_SUCCEEDED 458  
WorkerStateEvent 457  
WritableImage 291, 295  
write() 295

## X

X\_AXIS 323  
x-large 723  
x-small 723  
xx-large 723  
xx-small 723  
XYChart 678, 683  
XYChart.Data 685  
XYChart.Series 684

## Y

Y\_AXIS 323

## Z

Z\_AXIS 323  
ZERO 215, 360



**А**

Абсолютное позиционирование 116  
Аккордеон 156  
Анимация 359  
◊ параллельная 380  
◊ пауза 382  
◊ последовательная 381  
Аудио 699

**Б**

Безье 249, 276  
Буфер  
◊ глубины 94  
◊ обмена 197

**В**

Видео 699  
Видимость узла 114  
Вкладка 151  
Вращение 329, 375, 734  
Всплывающая подсказка 649  
Всплывающее окно 642, 646, 647  
Выделение 483  
Выражения 422

**Г**

Гиперссылка 434  
Гистограмма 689  
Градиент  
◊ линейный 230  
◊ радиальный 233  
Граф сцены 409  
График 675  
Графика  
◊ 2D 215  
◊ 3D 389

**Д**

Диаграмма 675  
◊ ось 679, 686  
Диалоговое окно 655  
Дистрибутив 750  
Доступность узла 115  
Дуга 245, 275, 308

**З**

Заливка 303  
Значок в заголовке окна 87

**И**

Иерархический список 579, 595  
Изображения 283, 315  
◊ доступ к пикселям 288  
◊ загрузка 283  
◊ отображение в окне 286  
◊ снимок сцены и узла 297  
◊ создание 290  
◊ сохранение в файл 295  
Индикатор хода процесса 449, 451  
Интернационализация приложения 739  
Искажение 356  
Источник света 405

**К**

Камера 403  
Каскадные таблицы стилей 713  
Клавиши  
◊ быстрого доступа 185, 632  
◊ коды 181  
Клавиши-ускорители 187, 633  
Кнопка 435  
◊ с меню 638  
Компоненты 409  
Контейнер 93, 119  
Контекстное меню 652  
Контраст 346  
Координаты точки 215  
Круг 255  
Куб 394  
Курсор 196, 735

**Л**

Линия 244, 248, 273, 274, 307  
◊ кривая Безье 249, 276

**М**

Маска слоя 238  
Масштаб 322, 328, 373, 734  
Менеджер компоновки 119  
Меню 623, 625  
◊ контекстное 652  
◊ панель 624

- ◇ пункт 628
- ◇ пункт с переключателем 635
- ◇ пункт с флажком 634
- ◇ разделительная линия 636
- Многоугольник 257, 313
- Модальное окно 85

## Н

- Надпись 426
- Насыщенность 346

## О

- Обводка 240, 304
- Окно 71
  - ◇ всплывающее 642, 646, 647
  - ◇ диалоговое 655
  - ◇ заголовок 73
  - ◇ закрытие 91
  - ◇ местоположение 79
  - ◇ модальное 85
  - ◇ отображение 73
  - ◇ поверх других окон 84
  - ◇ полноэкранный режим 82
  - ◇ прозрачность 84
  - ◇ произвольной формы 90
  - ◇ разворачивание 81
  - ◇ размеры 77
  - ◇ сворачивание 81
  - ◇ смена значка 87
  - ◇ создание 71
  - ◇ сокрытие 74
  - ◇ стиль 75
  - ◇ фокус ввода 83
- Освещение 348
- Ось диаграммы 679, 686
- Отражение 338
- Отступ
  - ◇ внешний 122, 125
  - ◇ внутренний 100

## П

- Панель
  - ◇ аккордеон 156
  - ◇ инструментов 468
  - ◇ с вкладками 150
  - ◇ с заголовком 154
  - ◇ с изменяемыми размерами 157
  - ◇ с полосами прокрутки 465

- Пароль 491
- Переключатель 440, 441
- Плеер 703
- Поворот 322, 329, 375, 734
- Подсказка 649
- Полоса прокрутки 462, 737
- Прозрачность
  - ◇ окна 84
  - ◇ узла 113, 368, 733
- Прозрачный фон 97, 107
- Прямоугольник 252, 311
- Псевдоклассы 721

## Р

- Разделительная линия 470
- Размеры 97, 219, 221
- Размытие 339–341
- Рамка 111, 727
- Режимы наложения 236, 352, 734

## С

- Свечение 342, 343
- Связывание 419, 421
- Сглаживание 94, 243, 736
- Сдвиг 331
- Сектор 245, 308
- Селектор 720
- Сепия 347
- Сигналы 161
- Скриншот 298
- Смещение 321, 326, 369, 733
- Снимок
  - ◇ сцены и узла 297
  - ◇ экрана 298, 299
- События 161
  - ◇ блокировка обработчика 165
  - ◇ генерация 167, 211
  - ◇ закрытие окна 176
  - ◇ иерархия 169
  - ◇ клавиатуры 178
  - ◇ мыши 187
  - ◇ назначение обработчика 161
  - ◇ окна 174
  - ◇ перетаскивание 202
  - ◇ прокрутка колеса мыши 194
  - ◇ типы событий 170
  - ◇ удаление обработчика 165
  - ◇ фокус ввода 183

Список 511, 540

- ◊ для выбора даты 537
- ◊ для выбора цвета 535
- ◊ иерархический 579, 595
- ◊ раскрывающийся 522, 530

Стили 713

- ◊ приоритет 716

Субсцена 402

Сфера 397

Сцена 93

## Т

Таблица 553

Таймер 170, 387

Текст 258, 313, 426, 473, 725

- ◊ выделение 265, 483
- ◊ выравнивание 264
- ◊ зачеркнутый 260, 725
- ◊ копирование и вставка 485
- ◊ подчеркнутый 260, 725
- ◊ положение курсора 266

Текстовое поле 473, 489

- ◊ для ввода пароля 491

- ◊ многострочное 491

Текстура 296

Тень 333, 335, 337

Траектория 269, 308

Трансформации 321, 366

Трансформация перспективы 344

## У

Узел 97

- ◊ поиск 409

Указатель мыши 196, 735

## Ф

Флажок 444

Фон 88, 89, 96, 105, 296, 730

- ◊ прозрачный 97, 107

## Х

Холст 301

## Ц

Цвет 226, 377, 535, 718

Цветовой тон 346

Цилиндр 396

## Ш

Шкала

- ◊ времени 383

- ◊ с ползунком 446

Шрифт 261, 722

## Э

Экран 80

Эллипс 254, 312

Эффект 734

Эффекты 333

## Я

Яркость 346

# JavaFX

**Прохоронок Николай Анатольевич**, профессиональный программист, имеющий большой практический опыт создания и продвижения сайтов, анализа и обработки данных, автор книг «Основы Java», «OpenCV и Java. Обработка изображений и компьютерное зрение», «HTML, JavaScript, PHP и MySQL. Джентльменский набор Web-мастера», «Python 3. Самое необходимое», «Python 3 и PyQt 5. Разработка приложений» и др., многие из которых выдержали несколько переизданий и стали бестселлерами.

Описываются базовые возможности библиотеки JavaFX, позволяющей создавать приложения с графическим интерфейсом на языке Java. Книга ориентирована на тех, кто уже знаком с языком программирования Java и хотел бы научиться разрабатывать оконные приложения, насыщенные графикой, анимацией и интерактивными элементами.

Рассматриваются способы обработки событий, управление свойствами окна, создание формы с помощью программы Scene Builder, а также все основные компоненты (кнопки, текстовые поля, списки, таблицы, меню и др.) и варианты их размещения внутри окна.

Книга содержит большое количество практических примеров, помогающих начать разрабатывать приложения с графическим интерфейсом самостоятельно. Весь материал тщательно подобран, хорошо структурирован и компактно изложен, что позволяет использовать книгу как удобный справочник.



Примеры из книги можно скачать по ссылке <ftp://ftp.bhv.ru/9785977540728.zip>, а также на странице книги на сайте [www.bhv.ru](http://www.bhv.ru).

191036, Санкт-Петербург,  
Гончарная ул., 20

Тел.: (812) 717-10-50,  
339-54-17, 339-54-28

E-mail: [mail@bhv.ru](mailto:mail@bhv.ru)  
Internet: [www.bhv.ru](http://www.bhv.ru)

ISBN 978-5-9775-4072-8



# В ПОДЛИННИКЕ®