

Крис Баккет

Dart в действии

Dart in Action

CHRIS BUCKETT



MANNING

Shelter Island

Крис Баккет

Dart в действии

С предисловием Сета Лэдда



Москва, 2013

УДК 004Dart
ББК 32.973-018.2
Б19

Баккет К.

Б19 Dart в действии. — М.: ДМК Пресс, 2013. — 528 с.: ил.

ISBN 978-5-94074-918-9

Dart — язык программирования для разработки веб-приложений, созданный компанией Google. Он обладает современными объектно-ориентированными средствами, как Java или C#, не отказываясь при этом от свойственных JavaScript динамичности и ориентированности на функциональное программирование. Написанные на Dart приложения транслируются в JavaScript, но могут исполняться и непосредственно в браузерах, поддерживающих Dart. В комплекте с Dart поставляются библиотеки и инструментальные средства промышленного качества. На Dart могут быть написаны как клиентская, так и серверная часть приложения, что упрощает процесс разработки.

В этой книге вы познакомитесь с языком Dart и научитесь использовать его для создания приложений, работающих в браузере, — в настольной или мобильной ОС. Это не просто учебное пособие по языку, довольно быстро автор переходит к техническим аспектам работы с Dart. На большинство вопросов, возникающих при чтении, тут же даются ответы!

Издание предназначено веб-программистам разной квалификации, в том числе малознакомым с объектно-ориентированным программированием.

УДК 004Dart
ББК 32.973-018.2

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1-6172-9086-2 (анг.)

Copyright © 2013 by Manning
Publications Co.

ISBN 978-5-94074-918-9 (рус.)

© Оформление, перевод
ДМК Пресс, 2013



Содержание

Предисловие	14
Вступление	16
Благодарности	18
Об этой книге	20
Об иллюстрации на обложке	25
 ЧАСТЬ I. ВВЕДЕНИЕ В DART	 26
Глава 1. Здравствуй, Dart	27
1.1. Что такое Dart?	27
1.1.1. Знакомый синтаксис помогает в освоении языка	29
1.1.2. Архитектура одностраничного приложения	30
1.2. Первый взгляд на язык Dart	32
1.2.1. Строковая интерполяция	32
1.2.2. Факультативные типы в действии	34
1.2.3. Традиционная структура на основе классов	36
1.2.4. Определение подразумеваемого интерфейса	38
1.2.5. Фабричные конструкторы для предоставления реализации по умолчанию	39
1.2.6. Библиотеки и область видимости	40
1.2.7. Функции как полноценные объекты	43
1.2.8. Параллелизм с помощью изоляторов	45
1.3. Веб-программирование на языке Dart	46
1.3.1. dart:html: удобная библиотека для работы с моделью DOM браузера	47
1.3.2. Dart и HTML5	48
1.4. Инструментальная экосистема Dart	50
1.4.1. Редактор Dart	50
1.4.2. Виртуальная машина Dart	51
1.4.3. Dartium	51
1.4.4. dart2js: конвертер Dart в JavaScript	51

1.4.5. Управление пакетами с помощью pub	52
1.5. Резюме	53

Глава 2. Программа «Здравствуй, мир» на Dart..... 54

2.1. VM Dart для командных приложений	55
2.2. Программа «Здравствуй, мир» в редакторе Dart	57
2.2.1. Знакомство с инструментами, встроенными в Редактор Dart Editor	58
2.2.2. Dart-скрипты и HTML-файлы	61
2.2.3. Запуск приложения «Здравствуй, мир» в Dartium.....	62
2.2.4. Использование dart2js для конвертации в JavaScript	63
2.2.5. Генерация документации с помощью dartdoc	66
2.2.6. Отладка Dart-кода с помощью точек останова	66
2.3. Импорт библиотек для работы с пользовательским интерфейсом в браузере.....	68
2.3.1. Импорт библиотек Dart.....	69
2.3.2. Доступ к элементам DOM с помощью dart:html	70
2.3.3. Динамическое добавление новых элементов на страницу	71
2.4. Резюме	72

Глава 3. Создание и тестирование

Dart-приложения 74

3.1. Конструирование пользовательского интерфейса с помощью dart:html	75
3.1.1. Начальный HTML-файл	76
3.1.2. Создание элементов с помощью dart:html	77
3.1.3. Создание экземпляра Element из фрагмента HTML ...	78
3.1.4. Создание элементов по имени тега	80
3.1.5. Добавление элементов в HTML-документ.....	82
3.2. Добавление интерактивности с помощью событий браузера	86
3.2.1. Добавление предмета в список по нажатии кнопки ...	86
3.2.2. Применение гибкого синтаксиса функций в Dart для обработки событий.....	87
3.2.3. Реагирование на события браузера.....	90
3.2.4. Рефакторинг прослушивателя событий для повторного использования.....	91

3.2.5. Запрос HTML-элементов в dart:html	92
3.3. Инкапсуляция структуры и функциональности с помощью классов.....	95
3.3.1. Классы в Dart не таят неожиданностей	96
3.3.2. Конструирование класса PackItem.....	97
3.3.3. Инкапсуляция функциональности с помощью методов чтения и установки.....	99
3.4. Автономное тестирование программы.....	103
3.4.1. Создание автономных тестов	105
3.4.2. Определение ожидаемых результатов теста	106
3.4.3. Создание пользовательского сравнителя	107
3.5. Резюме	109
ЧАСТЬ II. ЯЗЫК DART	111
Глава 4. Функции и замыкания	112
4.1. Функции в Dart	113
4.1.1. Тип возвращаемого функцией значения и ключевое слово return	116
4.1.2. Передача функции данных с помощью параметров	119
4.2. Функции как полноценные объекты.....	126
4.2.1. Объявления локальных функций.....	128
4.2.2. Определение строгого типа функции	134
4.3. Замыкания.....	137
4.4. Резюме	140
Глава 5. Библиотеки и ограничение доступа	142
5.1. Определение и импорт библиотеки	143
5.1.1. Определение библиотеки с помощью ключевого слова library	145
5.1.2. Импорт библиотек.....	147
5.2. Скрытие функциональности путем ограничения доступа к частям библиотеки	155
5.2.1. Ограничение доступа в классах.....	157
5.2.2. Использование закрытых функций в библиотеках....	162
5.3. Организация исходного кода библиотеки	163
5.3.1. Ключевые слова part и part of.....	164
5.4. Упаковка библиотек	168

5.5. Скрипты – это исполняемые библиотеки	171
5.6. Резюме	173

Глава 6. Классы и интерфейсы

6.1. Определение простого класса	176
6.1.1. Программирование относительно интерфейса класса	177
6.1.2. Формализация интерфейса путем явного его определения	180
6.1.3. Реализация нескольких интерфейсов	181
6.1.4. Объявление акцессоров свойств	182
6.2. Конструирование классов и интерфейсов	184
6.2.1. Конструирование экземпляров класса	185
6.2.2. Проектирование и использование классов с несколькими конструкторами	187
6.2.3. Использование фабричных конструкторов для создания экземпляров абстрактных классов	187
6.2.4. Применение фабричных конструкторов для повторного использования объектов	190
6.2.5. Использование статических методов и свойств совместно с фабричными конструкторами	191
6.3. Создание константных классов с неизменяемыми полями	194
6.3.1. Финальные значения и свойства	195
6.3.2. Блок инициализации конструктора	195
6.3.3. Использование ключевого слова const для создания константного конструктора	196
6.4. Резюме	197

Глава 7. Расширение классов и интерфейсов

7.1. Расширение классов с помощью наследования	200
7.1.1. Наследование класса	201
7.1.2. Наследование конструкторов	203
7.1.3. Переопределение методов и свойств	205
7.1.4. Включение абстрактных классов в иерархию наследования	206
7.2. Все является объектом	210
7.2.1. Проверка отношения «является» Object	210
7.2.2. Использование отношения «является» применительно к Object	212

7.2.3. Использование метода toString(), унаследованного от класса Object	213
7.2.4. Перехват обращений к методу noSuchMethod()	215
7.2.5. Прочая функциональность класса Object	218
7.3. Знакомство с типом dynamic.....	219
7.3.1. Использование аннотации типа dynamic.....	220
7.4. Резюме	221

Глава 8. Классы коллекций

8.1. Работа с коллекциями данных.....	224
8.1.1. Коллекции объектов	226
8.1.2. Использование конкретных реализаций интерфейса Collection.....	230
8.1.3. Создание специализированных коллекций с помощью обобщенных типов.....	233
8.1.4. Хранение списков пар ключ–значение в обобщенных словарях	237
8.2. Создание обобщенных классов	242
8.2.1. Определение обобщенного класса.....	242
8.2.2. Использование своего обобщенного класса.....	244
8.2.3. Ограничения на параметрические типы.....	245
8.3. Перегрузка операторов	246
8.3.1. Перегрузка операторов сравнения.....	247
8.3.2. Неожиданное применение перегрузки операторов	249
8.3.3. Перегрузка операторов доступа по индексу	249
8.4. Резюме	252

Глава 9. Асинхронное программирование с применением обратных вызовов и будущих значений

9.1. Почему веб-приложение должно быть асинхронным.....	256
9.1.1. Преобразование приложения в асинхронное	259
9.2. Использование обратных вызовов в асинхронном программировании	263
9.2.1. Добавление асинхронных обратных вызовов в Лотерею Dart.....	265
9.2.2. Ожидание завершения всех асинхронных обратных вызовов перед продолжением	266

9.2.3. Вложенные обратные вызовы как средство управления порядком асинхронного выполнения	269
9.3. Знакомство с типами Future и Completer	271
9.3.1. Передача будущих значений из одного места программы в другое	274
9.3.2. Упорядочение асинхронных вызовов путем сцепления будущих значений	275
9.3.3. Ожидание завершения всех запущенных операций получения будущих значений	276
9.3.4. Преобразование обычных значений в будущее	278
9.4. Автономное тестирование асинхронных API.....	280
9.4.1. Тестирование асинхронных функций обратного вызова	282
9.4.2. Тестирование будущих значений	283
9.5. Резюме	285

ЧАСТЬ III. КЛИЕНТСКИЕ DART-ПРИЛОЖЕНИЯ

287

Глава 10. Создание веб-приложения на Dart

288

10.1. Структура одностраничного веб-приложения	289
10.1.1. Приложение DartExpense – постановка задачи	290
10.1.2. Структура Dart-приложения.....	293
10.1.3. Поток выполнения в Dart-приложении	296
10.2. Конструирование пользовательского интерфейса с помощью dart:html	299
10.2.1. Интерфейс Element	299
10.2.2. Конструкторы элементов в действии	303
10.2.3. Организация взаимодействия с представлениями и элементами	305
10.2.4. Построение простой обобщенной сетки	309
10.3. Обработка браузерных событий с помощью dart:html ...	313
10.3.1. Управление порядком прохождения события в браузере	315
10.3.2. Наиболее распространенные типы событий	317
10.4. Резюме	319

Глава 11. Навигация и локальное хранение

данных	321
11.1. Интеграция навигации с браузером	323

11.1.1. Применение функции <code>pushState()</code> для добавления элементов в браузерную историю навигации	324
11.1.2. Подписка на событие <code>popState</code>	326
11.2. Использование куков браузера для повышения удобства работы	329
11.2.1. Сохранение данных в кукe	330
11.2.2. Чтение данных из кука	332
11.3. Локальное хранение данных с помощью Web Storage API	334
11.3.1. Преобразование объектов Dart в формат JSON	335
11.3.2. Преобразование JSON-строк в Dart-объекты	340
11.3.3. Сохранение данных в браузере	341
11.4. Резюме	346

Глава 12. Взаимодействие с другими системами и языками

12.1. Взаимодействие с JavaScript	349
12.1.1. Отправка данных из Dart в JavaScript	352
12.1.2. Получение в JavaScript данных, посланных из Dart	355
12.1.3. Отправка данных из JavaScript в Dart	358
12.2. Взаимодействие с внешними серверами	361
12.2.1. Правило одного домена	363
12.2.2. Использование JSONP для запроса данных у внешнего сервера	364
12.3. Построение допускающих установку браузерных приложений, не требующих сервера	367
12.3.1. Применение технологии AppCache для запуска приложений в автономном режиме	368
12.3.2. Создание пакета приложения, допускающего установку в Chrome	373
12.4. Резюме	377

ЧАСТЬ IV. DART НА СТОРОНЕ СЕРВЕРА

Глава 13. Работа с файлами и протоколом HTTP на сервере

13.1. Запуск серверного Dart-скрипта	381

13.1.1. Доступ к аргументам командной строки	384
13.1.2. Доступ к файлам и папкам с помощью dart:io.....	386
13.2. Обслуживание HTTP-запросов от браузера.....	393
13.2.1. Класс <code>HttpServer</code>	395
13.2.2. Передача статических файлов по HTTP.....	397
13.3. REST API для клиентов	399
13.3.1. Отправка содержимого каталога в формате JSON	402
13.3.2. Отправка содержимого файла в формате JSON	403
13.3.3. Добавление пользовательского интерфейса на стороне клиента	404
13.4. Резюме	409

Глава 14. Отправка, синхронизация

и сохранение данных	410
14.1. Передача приложения <code>DartExpense</code> с сервера	411
14.2. Использование веб-сокетов для организации двусторонней связи	411
14.2.1. Соединение через веб-сокет на стороне клиента	414
14.2.2. Обработка подключения через веб-сокет на сервере	416
14.2.3. Использование веб-сокетов для межбраузерной синхронизации	419
14.3. Сохранение данных в базе <code>CouchDB</code> с помощью класса <code>HttpClient</code>	426
14.3.1. Краткое введение в <code>CouchDB</code>	427
14.3.2. Совместное использование модельного класса <code>Expense</code> в коде клиента и сервера.....	430
14.3.3. Добавление поддержки сохранения данных на сервере	431
14.4. Резюме	437

Глава 15. Организация параллелизма

с помощью изоляторов	438
15.1. Использование изоляторов как единиц работы	439
15.1.1. Создание изолятора	440
15.1.2. Односторонняя связь с изолятором	442
15.1.3. Двусторонняя связь с изолятором	446

15.2. Динамическая загрузка кода.....	452
15.2.1. Создание изолятора для загружаемого файла.....	454
15.2.2. Определение динамически загружаемого исходного файла	455
15.3. Запуск нескольких исполнителей.....	457
15.4. Резюме	463

Приложение А. Справочное руководство

по языку	464
A.1. Объявления переменных	464
A.1.1. Объявление переменных с ключевым словом <code>var</code> или именем типа.....	465
A.1.2. Объявление финальных (доступных только для чтения) переменных	466
A.1.3. Синтаксис литералов	466
A.1.4. Обобщенные списки и словари	474
A.2. Функции	475
A.2.1. Длинная синтаксическая форма.....	477
A.2.2. Короткая синтаксическая форма.....	477
A.2.3. Параметры функции.....	478
A.2.4. Функции как полноценные объекты	479
A.3. Управление потоком выполнения и итерирование	482
A.3.1. Ветвление потока выполнения	482
A.3.2. Циклы и итерирование	487

Приложение В. Определение классов

и библиотек	491
V.1. Классы и интерфейсы.....	491
V.1.1. Определение классов	491
V.1.2. Наследование классов	502
V.1.3. Абстрактные классы.....	505
V.1.4. Неявные интерфейсы.....	506
V.1.5. Статические свойства и методы.....	508
V.2. Библиотеки и ограничение доступа	509
V.2.1. Определение библиотек	509
V.2.2. Ограничение доступа к элементам библиотеки.....	510

Предметный указатель.....	512



Предисловие

Услышав, что мы приступаем к работе над Dart – структурным, масштабируемым языком с быстрой виртуальной машиной, развитым редактором и компилятором на JavaScript, я поначалу не поверил. «Быть может, именно этот проект сделает веб-программирование проще для таких разработчиков, как я сам?» – задавался я вопросом в надежде получить утвердительный ответ. Имея опыт работы со структурными языками и мощными инструментальными средствами разработки, я ожидал какого-нибудь более продуктивного способа создавать современные крупномасштабные веб-приложения. Проект Dart казался как раз тем, что я искал. Я вырос на таких объектно-ориентированных языках, как C++, Java и Ruby, которые применял при построении своих первых интерактивных веб-сайтов, а позже обогащенных веб-приложений с развитой клиентской частью. Я научился продуктивно работать с классами, объектами и модульным кодом. Я ценил интегрированные среды разработки (IDE) за встроенные в них средства анализа, рефакторинга и навигации, потому что они помогали мне писать большие и сложные приложения. Жизнь была прекрасна. Я искал новые направления приложения своих знаний, и мне повезло получить работу в команде, работавшей над браузером Chrome. Поначалу я осваивал современный браузер и с воодушевлением принялся изучать многочисленные новые возможности HTML5. Ныне сеть веб развивается так быстро и у нее появилось столько пользователей, что находиться в центре событий безумно интересно. Жизнь стала еще прекрасней.

И хотя итеративный процесс веб-разработки с его коротким циклом мне очень нравился, все же не хватало столь любимых мной структурных языков и полезных инструментов. Я хотел, чтобы программы для современных браузеров можно было писать в IDE, умеющих выполнять автозавершение кода, на языках с настоящими классами... перечислять можно долго.

Поэтому, услышав о Dart, я ухватился за представившуюся возможность. Писать для самой интересной платформы, да еще не от-

казываясь от навыков и инструментов, с которыми хорошо знаком и умеешь работать? Да кто же будет против!

И я был не единственным разработчиком, кто с радостью присоединился к проекту. Автор этой книги Крис Бакетт был одним из первых, кто принял Dart. Он завел свой блог Dartwatch в тот самый день, как Google анонсировала Dart, и этот блог по сей день живет и процветает. Крис участвовал в проекте с самого начала, поэтому естественно, что именно он написал одну из первых книг, призванных помочь другим разработчикам в изучении Dart.

Крис – потрясающий автор, потому что сумел написать книгу в условиях, когда язык и его библиотеки претерпевали постоянные изменения. Ему удалось осветить многочисленные аспекты проекта Dart. Особенно мне понравились примеры, в которых иллюстрируется не только функциональность базового языка, но и более сложные вопросы, касающиеся HTML5. Крис рассматривает одностраничные приложения и показывает, как с помощью Dart создаются современные приложения, работающие в браузере. Но и это не все – вы научитесь программировать на Dart даже серверные приложения!

После года напряженной работы, десятков тысяч сохранений в системе управления версиями, исправления тысяч ошибок и непрерывающейся обратной связи с сообществом мечта о структурном языке для веб-программирования стала реальностью. И хотя разработка Dart еще не завершена, сегодня мы – благодаря книге Криса – уже можем создавать замечательные приложения для современного Интернета. Ликуйте!

Сет Лэдд,
разработчик и пропагандист
Google



Вступление

В октябре 2011 года подтвердились слухи о том, что Google разработала новый язык, ориентированный на разработку сложных веб-приложений характерного для Google масштаба. Спустя месяц в сети был обнародован внутренний документ Google под названием «Future of JavaScript» (Будущее JavaScript), из которого следовало, что в Google ведутся работы над языком, более подходящим для веб-разработки, чем JavaScript. Упоминалось даже предположительное название языка – Dash. Идея о создании нового языка родилась из-за медленного развития JavaScript, обусловленного отчасти тем, что в нем заинтересовано слишком много сторон и комитетов. Целью же было показать, каким мог бы стать JavaScript, если бы он был придуман сегодня. Основная задача формулировалась так: «сохранить динамическую природу JavaScript, но повысить производительность и обеспечить поддержку со стороны инструментальных средств при разработке крупных проектов». Кроме того, язык должен был допускать кросс-компиляцию на JavaScript. Апробационная техническая версия языка была представлена миру и получила название Dart.

Я как раз тогда закончил работу над крупным заказным справочным приложением, написанным на GWT и предназначенным для развертывания на устройствах с сенсорными экранами в среде, неблагоприятной для компьютеров. Технология Google Web Toolkit (GWT) была разработана Google для кросс-компиляции Java на JavaScript. GWT позволяет разработчику воспользоваться модульной структурой, типобезопасностью и инструментальной поддержкой Java при написании ориентированных на браузеры программ, не требующих подключения дополнительных модулей, таких как Flash или Silverlight. Потратив предшествующие два года на написание GWT-кода и координацию работы программистов из трех стран, я понимал, насколько ценны средства проверки кода в точках интеграции – то, чего так не хватает языку JavaScript. Меня также привлекала возможность использовать один и тот же код на сто-

роне клиента и сервера – я видел, какие из этого можно извлечь преимущества.

Горя желанием узнать как можно больше о новом языке Dart, я прочитал всю имевшуюся документацию. На тот момент она состояла из исходного кода, нескольких примеров и спецификации языка. Мне казалось, что раз уж я потратил время на то, чтобы уложить эти знания у себя в голове, то было бы хорошо поделиться ими с публикой с помощью блога. Тогда-то я и завел блог Dartwatch, где опубликовал серию статей о том, как решаются типичные задачи в Dart: организация проекта, создание классов, взаимодействие с браузером. Одно цепляло за собой другое – и в конце концов издательство Manning обратилось ко мне с предложением написать книгу о Dart. Через год результат этой работы вышел из печати.

За минувший год у Dart было время созреть, а его разработчики внимательно прислушивались к отзывам и реагировали на них. Близились время первой контрольной точки Dart, и в первоначальную спецификацию языка было внесено множество изменений по предложениям первых приверженцев, пытавшихся применить его для решения реальных задач. Сообщество этих приверженцев также занималось разработкой различных инструментов и библиотек, в том числе драйверов баз данных, библиотек двумерной и трехмерной графики и каркасов. Многие из них можно найти на сайте GitHub или Dartwatch.

Первая контрольная точка Dart (Milestone 1) – это крупное достижение, дающее разработчикам возможность использовать базовый язык для создания впечатляющих библиотек и API, которые помогут превратить Dart в язык «все включено», каким его и видят идеологи в Google. С каждым днем Dart становится все лучше, а благодаря открытости проекта вы можете наблюдать за тем, что многочисленные разработчики помещают в репозиторий исходного кода (и даже принять в этом участие). Надеюсь, что эта книга поможет вам создать замечательные приложения на Dart.



Благодарности

Выяснилось, что написание книги – вовсе не такое простое дело, как мне казалось поначалу. Не будь поддержки со стороны всех участвующих в проекте сотрудников издательства Manning, вы вряд ли читали бы ее сегодня. Спасибо Майклу Стивенсу (Michael Stephens), который вообще предложил мне заняться этим делом, – проект оказался очень интересным. Многие сотрудники издательства трудились над редактированием, корректурой, подготовкой иллюстраций и выполнением кучи другой работы, связанной с производством книги, – спасибо всем вам.

Особо хочется отметить двух сотрудников Manning. Во-первых, спасибо Бέρту Бейтсу (Bert Bates) за то, что еще на ранних этапах работы он показал мне, как из сухого справочного материала сделать текст, который было бы приятно читать. Когда я начинал писать очередную главу, на периферии сознания неотступно присутствовала мантра: «Объясни Бέρту, почему ему нужно это знать...» Во-вторых, спасибо редактору Сюзанне Клайн (Susanna Kline), которая следила, чтобы каждая глава строго соответствовала своему названию, не позволяла мне расслабляться и заставляла выдерживать график.

Вокруг Dart сформировалось весьма активное сообщество разработчиков, которое пользуется для общения списком рассылки dartlang и системой Google+. Я хочу поблагодарить его участников Джона Эванса (John Evans) и Кэвина Мура (Kevin Moore) за техническое редактирование, а также Адама Сингера (Adam Singer), Мэтью Батлера (Matthew Butler) и Ладислава Тона (Ladislav Thon), чьи предложения всегда были желанны.

Благодарю также всех членов сообщества, которые читали книгу и давали отзывы на разных этапах ее написания: Андрэ Роберже (André Roberge), Кард Тасвелл (Carl Taswell), Чэд Дэвис (Chad Davis), Крэйг Ланкастер (Craig Lancaster), Дилан Скотт (Dylan Scott), Глен Стокол (Glenn Stokol), Йон Скит (Jon Skeet), Оливье Нугье (Olivier Nougier), Рик Гофф (Rick Goff), Родни Боллинджер (Rodney Bollinger), Рокеш Дженки (Rokesh Jankie), Стив Притти

(Steve Pretty), Терри Бэрч (Terry Birch) и Вильгельм Леман (Willhelm Lehman).

Также спасибо всем, кто писал в форуме книги, указывая на неизбежные опечатки, и всем читателям предварительных вариантов – участникам программы Manning's Early Access Program (MEAP).

Наконец, спасибо всем разработчикам Dart, в том числе Сету Лэдду, который помогал мне и многим другим ранним приверженцам оставаться в курсе изменений по мере эволюции Dart от первоначальной версии к той, что мы имеем сегодня. Отдельная благодарность Сету за любезное согласие написать предисловие к этой книге.



Об этой книге

Данная книга призвана помочь вам изучить язык Dart, понять его экосистему и писать на нем код, работающий в современных браузерах и на сервере. Вы будете использовать последние технологии HTML5, которые позволяют создавать приложения, работающие в браузере без подключения к сети, а также писать на Dart серверный код, поддерживающий двустороннюю связь с браузерами.

Будучи структурным языком, Dart идеален для разработки крупномасштабных приложений силами территориально разнесенной команды. А благодаря наличию инструментов для автоматической проверки кода, написанного всеми членами команды, Dart облегчает жизнь разработчикам.

Предполагаемая аудитория

Эта книга рассчитана на разработчиков, недовольных отсутствием должной языковой структуры и инструментальных средств для создания браузерных приложений. Если вы на рабочем уровне владеете Java, C# или JavaScript, то сможете сразу же приступить к работе с Dart.

И программисты, предпочитающие писать интерактивные пользовательские интерфейсы, и те, кому больше по душе разработка эффективного серверного кода, обнаружат, что Dart в сочетании с современными браузерными технологиями привносит на сторону клиента структуру, характерную для сервера, а на сторону сервера – гибкость, динамичность и скорость разработки, присущую клиентским приложениям.

Эта книга поможет быстро освоить идеи языка Dart как начинающему веб-разработчику, так и поднаторевшему в написании структурного кода. Все новые концепции поясняются на примерах. В тексте отмечаются как черты, роднящие Dart с другими языками, в частности Java и JavaScript, так и различия между ними.

Для Dart, как и для Java, имеются великолепные инструментальные средства. С другой стороны, Dart, как и JavaScript, не требует этапа компиляции, а это значит, что вы сможете очень быстро приступить к созданию клиентских и серверных приложений на Dart.

Структура книги

Эта книга построена так, чтобы читатель как можно скорее начал работать с Dart. Она состоит из четырех частей. Первая часть содержит обзорные главы.

- ❑ В главе 1 приводятся общие сведения об идеях и средствах языка и рассказывается о том, почему вообще Dart появился. Вы узнаете о философии, стоящей за Dart, и о том, какого рода крупномасштабные веб-приложения можно разрабатывать с его помощью.
- ❑ В главе 2 рассматривается более широкая экосистема, сложившаяся вокруг Dart, в том числе богатейшие инструментальные средства, которые вы получаете, выбирая структурный язык веб-разработки, созданный компанией, являющейся лидером на этом рынке. Обладая необходимыми техническими ресурсами, Google создала не только сам язык, но и IDE, специализированный браузер для быстрой разработки, серверную виртуальную машину и другие инструменты, нужные для создания высококачественного кода.
- ❑ В главе 3 мы напишем простенькое веб-приложение, чтобы понять, как Dart взаимодействует с браузером. Мы сконструируем пользовательский интерфейс, будем прослушивать события браузера и создадим автономные тесты, подтверждающие корректность кода.

Во второй части рассматриваются базовые языковые средства.

- ❑ Глава 4 посвящена функциям, которые в Dart являются полноценными объектами. Пишущим на JavaScript некоторые приемы функционального программирования покажутся знакомыми, тогда как разработчики на Java и C# найдут много новых для себя идей, типичных для браузерных приложений.
- ❑ В главе 5 мы продолжим конструировать приложение, воспользовавшись встроенной в Dart системой библиотек, и покажем, как эта система соотносится с ограничением доступа. Механизм ограничения доступа в Dart, возможно, станет сюрпризом для разработчиков на Java и C# и окажется желанным подарком для программистов на JavaScript.

- ❑ В главах 6, 7 и 8 исследуется структура классов и интерфейсов в Dart. Классы образуют костяк любого сколько-нибудь масштабного приложения, поэтому без умения строить эффективные иерархии классов и пользоваться библиотечными классами, написанными другими людьми, никак не обойтись.
- ❑ В главе 9 мы вернемся к функциональному программированию и разберемся с асинхронной природой API доступа к веб. Вы узнаете, как работать с будущими значениями, то есть с переменными, которые получают значение в будущем. Это подготовит почву для работы с API, предоставляемыми клиентскими и серверными библиотеками Dart.

В третьей части обсуждается разработка клиентских приложений, работающих в браузере.

- ❑ В главе 10 вы узнаете о цикле обработки событий и о создании пользовательских интерфейсов в Dart.
- ❑ В главе 11 в структуру приложения добавляются браузерная навигация, сохранение данных на стороне клиента и обработка данных, представленных в формате JSON.
- ❑ Добравшись до главы 12, вы уже будете готовы подключить свое приложение к внешним системам, например к внешнему JavaScript-коду и сторонним серверным API. Хотя Dart рассчитан на все современные браузеры, в этой главе мы научимся создавать пакет для развертывания в качестве приложения Chrome в магазине Google Web Store.

В части 4 речь пойдет об интерфейсе с серверным кодом.

- ❑ В главе 13 мы напишем на Dart командное приложение, обращающееся к файловой системе и отправляющее данные по протоколу HTTP, то есть разработаем простой файловый сервер.
- ❑ В главе 14 мы подключим клиентское приложение к серверной базе данных и организуем двустороннее взаимодействие по технологии Web Sockets для проталкивания данных клиенту.
- ❑ В главе 15, уже зная о взаимодействии с сервером, вы сможете понять, как Dart решает проблему параллелизма с помощью системы изоляторов – модели многопоточности на основе передачи сообщений, более безопасной, чем эквивалентные механизмы в Java и C#. Мы также воспользуемся системой изоляторов для динамической загрузки кода на Dart в работающее приложение. Тем самым мы зложим фундамент для разработки расширений и подключаемых модулей.

В приложениях содержится краткое справочное руководство по базовому языку Dart с примерами, иллюстрирующими синтаксические особенности и странности Dart.

Графические выделения и загрузка исходного кода

Весь исходный код в тексте выделяется моноширинным шрифтом. В книге много фрагментов кода и диаграмм, которые представляют собой законченные аннотированные листинги, иллюстрирующие основные идеи. Все это, как правило, тесно связано с окружающим текстом и составляет неотъемлемую часть изучения Dart.

Иногда код приходилось переформатировать, чтобы он поместился на страницу, но в общем случае код уже написан с учетом ограничений по ширине строки. Сами примеры часто упрощены, чтобы продемонстрировать какую-то важную концепцию, не отвлекаясь на детали, но в основном тексте и в аннотациях к коду приводятся дополнительные подробности.

Весь исходный код, представленный в этой книге, можно скачать с сайта издательства по адресу www.manning.com/DartinAction.

Требования к программному обеспечению

Для работы с Dart требуется как минимум Dart SDK, который можно скачать с сайта www.dartlang.org. В состав Dart SDK включены редактор Dart Editor, специализированный браузер Dartium (необходимый для быстрой разработки на Dart) и конвертер Dart в JavaScript. Имеются версии для Windows, Mac и Linux.

Автор в сети

Приобретение книги «Dart в действии» открывает бесплатный доступ к закрытому форуму, организованному издательством Manning Publications, где вы можете оставить свои комментарии к книге, задать технические вопросы и получить помощь от автора и других пользователей. Получить доступ к форуму и подписаться на список рассылки можно на странице www.manning.com/DartinAction. Там же написано, как зайти на форум после регистрации, на какую помощь можно рассчитывать, и изложены правила поведения в форуме.

Издательство Manning обязуется предоставлять читателям площадку для общения с другими читателями и автором. Однако это не означает, что автор обязан как-то участвовать в обсуждениях; его

присутствие на форуме остается чисто добровольным (и не оплачивается). Мы советуем задавать автору какие-нибудь хитроумные вопросы, чтобы его интерес к форуму не угасал!

Форум автора в сети и архивы будут доступны на сайте издательства до тех пор, пока книга не перестанет печататься.

Об авторе

Крис Бакетт работает техническим консультантом по созданию бизнес-приложений масштаба предприятия, доступных через веб. Крис ведет популярный блог Dartwatch.com и активно участвует в списке рассылки dartlang.



Об иллюстрации на обложке

Рисунок на обложке книги «Dart в действии» называется «Островитянка из города Задар в Далмации». Репродукция взята из опубликованного в 2006 году собрания костюмов и этнографических описаний XIX века под названием «Далмация», составленного профессором Франо Каррара (1812–1854) – археологом, историком и первым директором Музея античности в городе Сплит, Хорватия. Рисунок был получен при содействии библиотекаря этнографического музея Сплита (бывшего Музея античности), который расположен в той части средневекового центра города, которая восходит еще к временам римского владычества: рядом с руинами дворца императора Диоклетиана, датируемыми примерно 304 г. до н. э. В книге собраны тщательно раскрашенные изображения людей из разных районов Хорватии, сопровождаемые описаниями костюмов и деталей повседневного быта.

Задар – древний город, расположенный на Адриатическом побережье Хорватии, существовавший уже в каменном веке. Прямо напротив Задара расположены острова Углян и Пашман, отделенные от города узким Задарским проливом. Раньше мыс, на котором стоит город, был отделен от материка глубоким рвом, теперь засыпанным. В этом регионе ощущается влияние различных народов, правивших здесь на протяжении веков: от греков и римлян до венецианцев и австрийцев. Ныне это часть Республики Хорватия.

Манера одеваться с XIX века сильно изменилась, и различия между областями, когда-то столь разительные, сгладились. Теперь трудно отличить друг от друга даже выходцев с разных континентов, что уж говорить о соседних городках и местностях. Мы обменяли культурное разнообразие на иное устройство личной жизни – основанное на многогранном и стремительном технологическом развитии.

Во времена, когда один компьютер не отличишь от другого, издательство Manning откликается на новации и инициативы в компьютерной отрасли обложками своих книг, на которых представлено широкое разнообразие местных укладов быта в позапрошлом веке. Мы возвращаем его в том виде, в каком оно запечатлено на рисунках из собрания Каррары.



ЧАСТЬ I. ВВЕДЕНИЕ В DART

Dart – замечательный язык для разработки веб-приложений. В главе 1 мы расскажем о том, зачем был создан Dart и как с его помощью решаются некоторые задачи, с которыми сталкиваются веб-разработчики. Вы узнаете о возможностях языка и о том, почему одностраничные приложения – подходящая архитектура для применения Dart.

В главе 2 мы начнем изучать развитую экосистему инструментальных средств, окружающую Dart. Dart – не просто язык, это еще и полный набор средств для разработки, включая IDE, специализированный браузер для тестирования и отладки, а также конвертер из Dart в JavaScript.

В главе 3 мы напишем на Dart простое приложение, на примере которого продемонстрируем методику создания одностраничных приложений, работающих в браузере. По ходу дела мы познакомимся с языком, в том числе с классами, функциями и переменными. К концу главы у нас будет готов проект с работоспособным пользовательским интерфейсом и автономными тестами, после чего мы сможем перейти к систематическому изучению базового языка в части II.



Глава 1. Здравствуй, Dart

В этой главе

- ❑ Основные сведения о платформе разработки на Dart.
- ❑ Взгляд на язык Dart.
- ❑ Инструментальные средства для разработки Dart-приложений.

Dart – удивительный язык, открывающий возможность создавать сложные веб-приложения быстрее и с меньшим количеством ошибок, чем раньше. В этой главе мы узнаем, как сочетаются язык Dart и его инструментальная экосистема, рассмотрим основные особенности языка и увидим, как его можно использовать для построения одностраничных веб-приложений.

1.1. Что такое Dart?

Dart – структурный язык программирования с открытым исходным кодом, предназначенный для создания сложных веб-приложений, работающих в браузере. Выполнить приложение, написанное на Dart, можно либо в браузере, который поддерживает Dart напрямую, либо предварительно откомпилировав код на JavaScript. Dart обладает знакомым программистам синтаксисом, в нем есть классы и необязательные типы, он является однопоточным. Реализована также модель параллелизма на основе изоляторов, которая допускает параллельное выполнение, – мы обсудим ее в главе 15. Помимо исполнения в браузере и конвертации в JavaScript, написанный на Dart код можно запускать из командной строки в виртуальной машине Dart, что позволяет писать клиентскую и серверную части приложения на одном языке.

Синтаксис Dart очень похож на синтаксис Java, C# и JavaScript. Это не случайно – при создании Dart ставилась задача разработать знакомый язык. Вот крохотный скрипт на Dart, состоящий из единственной функции `main`:

<pre>main() { var d = "Dart"; String w = "World"; print("Hello \${d} \${w}"); }</pre>	← Функция <code>main()</code> – точка входа, с которой начинается исполнение полностью загруженного скрипта ← Типизация необязательна (тип не указан) ← Аннотация типа (задан тип <code>String</code>) ← Для вывода на консоль браузера или на <code>stdout</code> производится подстановка в строку

Этот скрипт можно внедрить в HTML-страницу с помощью тега `<script type="application/dart">` и выполнить в браузере Dartium (издание браузера Google Chrome, предназначенное для разработки на Dart). С помощью программы `dart2js` можно также преобразовать его в JavaScript и затем исполнить в любом современном браузере или запустить из командной строки на сервере, воспользовавшись виртуальной машиной Dart (BM Dart).

Но Dart – не просто язык. На рис. 1.1 изображена экосистема инструментов, включающая несколько сред исполнения, редактор, языковые утилиты и полный набор библиотек – все, что нужно для комфортной разработки сложных веб-приложений.

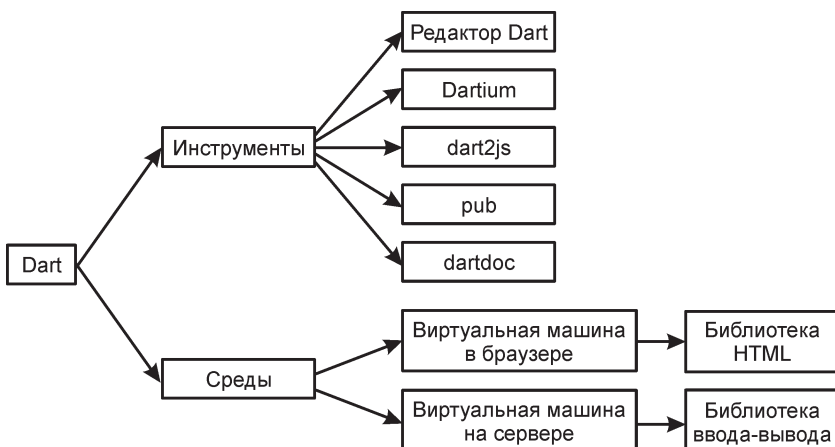


Рис. 1.1. Dart – больше, чем просто язык.
Проект Dart включает целую экосистему

Помимо великолепной инструментальной экосистемы, помогающей при создании приложений, Dart спроектирован так, что покажется знакомым разработчикам как серверных приложений на Java или C#, так и клиентских – на JavaScript или ActionScript.

Основным инструментом разработчика на Dart является браузер Dartium, который позволяет писать и редактировать код и наблюдать за его исполнением. Сочетание Dartium с редактором Dart дополнительно позволяет отлаживать трафик между клиентом и сервером.

1.1.1. Знакомый синтаксис помогает в освоении языка

Одним из важнейших проектных решений было стремление сделать Dart знакомым программистам на JavaScript и на Java/C#. Это помогает быстро освоить новый язык. Если вы владеете хотя бы одним из вышеупомянутых языков, то чтение и понимание написанного на Dart кода не вызовет особых затруднений.

Разработчикам на Java и C# привычны системы типов, классы, наследование и связанные с этим идеи. С другой стороны, на JavaScript пишут как дизайнеры пользовательского интерфейса, которые просто копируют чужой код, чтобы сделать страницу интерактивной (и никогда в жизни не пользовались типами), так и профессиональные программисты, понимающие, что такое замыкание и прототипическое наследование. Чтобы удовлетворить потребности столь разнообразной аудитории, в Dart введена факультативная типизация, позволяющая либо не указывать тип нигде (объявляя переменные с помощью ключевого слова `var`, как в JavaScript), либо везде добавлять аннотации типа (`String`, `int`, `Object` и т. п.), либо применять смешанный подход.

Использование информации о типе позволяет программисту документировать свои намерения, что полезно как для автоматизированных инструментальных средств, так и для коллег. Типичный процесс разработки Dart-приложения подразумевает постепенное построение системы типов по мере того, как программа обретает форму. Добавление или удаление информации о типе не влияет на работу программы, зато позволяет виртуальной машине более эффективно проверять правильность кода. Таким образом, система типов в Dart лежит где-то посередине между динамической типизацией в JavaScript и статической в Java и C#.

В табл. 1.1 сравниваются различные характеристики Dart, Java и JavaScript.

Dart – универсальный язык, позволяющий, подобно JavaScript и Java, создавать приложения разных типов. Но по-настоящему он сверкает при разработке сложных веб-приложений.

Таблица 1.1. Сравнение некоторых характеристик Dart, Java и JavaScript

Характеристика	Dart	Java	JavaScript
Система типов	Факультативная, динамическая	Строгая, статическая	Слабая, динамическая
Функции – полноценные объекты	Да	Можно имитировать с помощью анонимных функций	Да
Замыкания	Да	Да, с помощью анонимных классов	Да
Классы	Да, одиночное наследование	Да, одиночное наследование	Прототипические
Интерфейсы	Да, множественное наследование	Да, множественное наследование	Нет
Параллелизм	Да, на основе изоляторов	Да, на основе потоков	Да, на основе рабочих процессов в HTML5

1.1.2. Архитектура одностороннего приложения

Односторонние приложения типа Google Mail, Google Instant Search и Google Maps – именно то, для чего предназначен Dart. Исходный код всего приложения (или, по крайней мере, наиболее часто используемых его частей) загружается единственной веб-страницей. Этот – работающий в браузере – код отвечает за построение пользовательского интерфейса и загрузку с сервера отображаемых в нем данных (см. рис. 1.2).

В односторонних приложениях используется быстрая виртуальная машина на стороне клиента, позволяющая перенести часть обработки с сервера на клиент. В результате сервер способен обслужить больше запросов, так как избавлен от необходимости верстать страницу. Благодаря включенным в Dart библиотекам для работы с HTML, поддерживающим современные HTML5-технологии хранения и кэширования данных в браузере, удастся еще больше повысить производительность приложения и даже дать пользователю работать без подключения к сети.

У любого Dart-скрипта имеется единственная точка входа – функция `main()`, – с которой начинается исполнение скрипта виртуальной машиной Dart. Можно полагаться на то, что весь код, составляющий

приложение, уже загружен к моменту вызова `main`; в Dart невозможно определить и выполнить новую функцию, как в JavaScript не существует ни функции `eval()`, ни какого-либо иного способа модификации исполняемого кода. Именно эта особенность помогает писать на Dart приложения, отвечающие одностраничной архитектуре, потому что есть уверенность, что исполняемый код заранее известен. VM Dart пользуется этой гарантией для ускорения инициализации приложения, применяя мгновенные снимки кучи, – это позволяет загрузить Dart-приложение намного быстрее эквивалентного приложения на JavaScript.

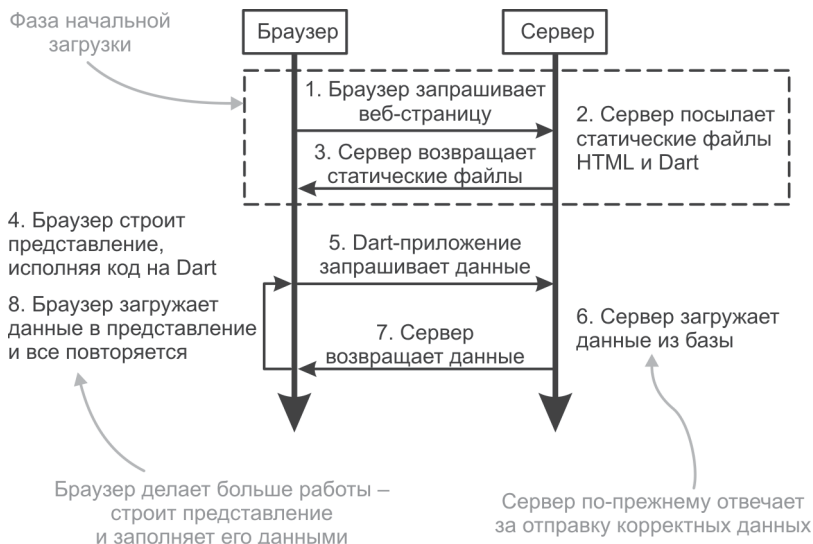


Рис. 1.2. Одностраничное приложение работает в браузере и обращается к серверу только за данными

Памятка

- Dart – язык разработки веб-приложений со знакомым синтаксисом.
- Благодаря наличию инструментальной экосистемы разработка на Dart более продуктивна, чем на эквивалентных динамических языках.
- Факультативная система типов в Dart лежит посередине между динамической типизацией в JavaScript и статической типизацией в Java.
- Аннотации типов могут оказаться очень полезными при коллективной разработке, так как упрощают автоматизированную проверку исходного кода.
- Dart идеален для разработки одностраничных веб-приложений.

Познакомившись с Dart как с платформой разработки, можно приступить к практическому освоению некоторых важнейших особенностей языка.

1.2. Первый взгляд на язык Dart

Dart – полнофункциональный современный язык программирования. Хотя корнями он уходит в Smalltalk, в нем ощущается влияние ряда других языков, в частности Java, C# и JavaScript. В этом разделе мы познакомимся с некоторыми базовыми концепциями языка и кратко упомянем о более сложных механизмах, которые будут подробно рассмотрены впоследствии.

Dart – развивающийся язык

На момент написания этой книги Dart находится на этапе перехода от апробационной технической версии к выпускной версии, которую Google называет «Milestone 1» (Контрольная точка 1). Контрольная точка 1 – это не версия 1, а проведенная на песке черта, от которой можно плясать при разработке и совершенствовании окружающего базовый язык библиотек. Платформа Dart мыслится как полнофункциональная среда разработки типа «все включено», содержащая все необходимое для создания сложных веб-приложений. И компания Google вместе с сообществом Dart сейчас занята разработкой таких библиотек.

Кроме того, контрольная точка 1 определяет состояние, в котором уже можно приступать к созданию приложений в уверенности, что несовместимые изменения в синтаксисе языка будут появляться нечасто. Однако изменения в сопутствующих библиотеках вполне вероятны, поэтому в редактор Dart включена полезная функция наведения порядка (Clean-up), позволяющая применить к коду изменения, внесенные в базовый язык и в библиотеки.

1.2.1. Строковая интерполяция

В веб-приложениях строки используются повсеместно. Dart предлагает несколько способов преобразовать выражение в строку, в том числе функцию `toString()`, встроенную в базовый класс `Object`, и строковую интерполяцию.

Для строковой интерполяции используется знак `$` или выражение `${ }` внутри одиночных или двойных кавычек. Чтобы подставить в строку переменную, необходимо предпослать ей префикс `$`, например: `$name`. Если же требуется подставить результат вычисления выражения, в частности вызова метода, то следует воспользоваться фигурными скобками:

```
"Ответ равен ${5 + 10}"
```

Для создания строки, занимающей несколько строчек, используются три двойные кавычки, а строковые литералы (внутри которых знак \$ не имеет специального смысла) создаются с помощью префикса `r`, например: `r'literal string'`. Для конкатенации двух строк оператор `+` не применяется; необходимо пользоваться либо строковой интерполяцией (`$forename $surname`), либо – если речь идет о заранее известных значениях – просто записать две строки одна за другой. Например, выражение

```
var title = "Dart " "в " "действии";
```

порождает строковую переменную со значением `"Dart в действии"`.

В листинге ниже показаны некоторые операции со строками. Для вывода результатов используется встроенная в Dart функция `print`, которая печатает на стандартный выход, если выполняется на сервере, или на отладочную консоль браузера – если в браузере.

Листинг 1.1. Строковая интерполяция в Dart

```
void main() {
  var h = "Hello";
  final w = "World";
  print('$h $w');

  print(r'$h $w');

  var helloWorld = "Hello " "World";
  print(helloWorld);

  print("${helloWorld.toUpperCase()}");
  print("Ответ равен ${5 + 10}");

  var multiline = """
<div id='greeting'>
  "Hello World"
</div>""";
  print(multiline);

  var o = new Object();
  print(o.toString());
  print("$o");
}
```

\$ используется для вычисления простых переменных	← При наличии префикса <code>r</code> выводится строковый литерал без интерполяции
Соседние строковые константы конкатенируются	Вычисляемое выражение должно быть заключено в фигурные скобки <code>{ }</code>
← При записи многострочных строк первый символ новой строки после <code>"""</code> игнорируется Многострочные строки могут содержать одиночные и двойные кавычки	В случае строковой интерполяции автоматически вызывается функция <code>toString()</code>

Эта программа печатает следующее:

```
Hello World
$h $w
Hello World
HELLO WORLD
Ответ равен 15
<div id='greeting'>
  "Hello World"
</div>
Instance of 'Object'
Instance of 'Object'
```

Во время экспериментов с Dart вы часто будете использовать строковую интерполяцию и функцию `print`, чтобы печатать переменные и подставлять значения в HTML-код.

1.2.2. Факультативные типы в действии

Одно из основных отличий JavaScript от Dart – наличие в Dart концепции типов. Однако типы факультативны, и, чтобы воспользоваться строгой типизацией, необходимо включить в код аннотации типов.

Факультативные аннотации типов могут встречаться в объявлениях переменных, параметров функций и возвращаемых значений, а также в определениях классов. Ниже показаны четыре способа объявить строковую переменную `message`. В первых двух аннотации типа нет вообще, а в двух оставшихся есть аннотация `String`, сообщающая программистам и инструментам о том, что в этой переменной будет храниться строковое значение.

<pre>var messageA; var messageB = "Hello Dart";</pre>	Нет аннотаций типов
<pre>String messageC; String messageD = "Hello Dart";</pre>	Аннотация типа присутствует

В двух из четырех случаев переменная `message` инициализируется в момент объявления. Если известно, что значение не изменится после объявления, то следует добавить ключевое слово `final`:

<pre>final messageE = "Hello Dart";</pre>	← Использование <code>final</code> без аннотации типа
<pre>final String messageF = "Hello Dart";</pre>	← Использование <code>final</code> с аннотацией типа

Подробнее о ключевом слове `final` мы будем говорить ниже.

Чтобы продемонстрировать полезность факультативной типизации, рассмотрим приведенный ниже код, в котором функция `trueIfNull()` принимает два параметра и возвращает `true`, если оба равны `null` (и `false` в противном случае). Сейчас в этом коде нет аннотаций типов, но мы объясним, как ими можно было бы воспользоваться для выражения намерений.

```
trueIfNull(a, b) {  
    return a == null && b == null;  
}
```

Функция принимает
два значения

```
main() {  
    final nums = trueIfNull(1,2); ← Сохраняет "false" в динамической переменной nums  
    final strings = trueIfNull("Hello ", null); ← Сохраняет "true" в динамической  
                                                переменной strings  
  
    print("$nums");  
    print("$strings");  
}
```

Выводит переменные
nums и strings на консоль

Поскольку в этом фрагменте аннотаций типов нет, то мы ничего не можем сказать о намерениях разработчика. Имя функции `trueIfNull(a,b)` позволяет предположить, что она принимает два значения типа `int` и возвращает значение типа `bool` (`true` или `false`), но, возможно, разработчик имел в виду что-то другое — например, хотел вернуть строку `"true"` вместо булевого значения. Факультативная типизация в Dart позволяет разработчику документировать свои намерения в виде информации о типах параметрах и возвращаемого значения.

```
bool trueIfNull(int a, int b) { ← Добавлены типы параметров и возвращаемого значения  
    return a == null && b == null;  
}
```

```
main() {  
    final bool nums = trueIfNull(1,2);  
    final bool strings = trueIfNull("Hello ", null);  
    print("$nums");  
    print("$strings");  
}
```

В объявления переменных
добавлена информация о типе

Примечание. В примере выше встречается тип `bool`. В Dart, в отличие от JavaScript, существует единственный способ представить значение

истинности «ложь», а именно ключевым словом `false`. Нуль и `null` не числятся как «ложь».

Добавление аннотаций типа не изменяет работу Dart-приложения, а только сообщает полезную информацию, которой инструменты и ВМ могут воспользоваться для проверки кода и нахождения ошибок типизации. Любая информация о типе может помочь инструментам в проведении статического анализа кода (в редакторе или в командной утилите непрерывной интеграции), а также на этапе выполнения. Программисты, которым в будущем придется сопровождать ваш код, тоже скажут «спасибо».

Совет. Указывайте конкретные типы (например, `String`, `List` и `int`) там, где это полезно для целей документирования, например при объявлении параметров функций, возвращаемых значений и открытых членов класса. Там же, где это не так, например в теле функции, используйте ключевое слово `var` или `final` без аннотации типа. Такой подход рекомендуется в руководстве по стилю программирования на Dart, доступном на сайте www.dartlang.org. Привыкайте к смешанному стилю, поскольку именно так и предполагается писать на Dart.

Факультативная типизация лежит в основе многих механизмов Dart и многократно упоминается на страницах этой книги там, где синтаксические отличия от Java и JavaScript заслуживают пояснения.

Функции как таковые рассматриваются в главе 4.

1.2.3. Традиционная структура на основе классов

В Dart классы и интерфейсы используются традиционным для объектно-ориентированных языков способом, без каких бы то ни было сюрпризов. Поддерживается наследование одному классу и нескольким интерфейсам. Если вы незнакомы с объектно-ориентированным программированием, то, наверное, имеет смысл почитать что-нибудь на эту тему – ресурсов в веб предостаточно. Пока отметим лишь, что объектно-ориентированная модель, принятая в Dart, похожа на то, что есть в Java и C#, и совсем не похожа на JavaScript. Подробнее классы рассматриваются в главах 6 и 7.

По умолчанию все классы в Dart наследуют классу `Object`. У них могут быть открытые и закрытые члены, а удобный синтаксис ме-

тодов чтения и установки позволяет использовать поля и свойства взаимозаменяемым образом, не затрагивая пользователей класса. Ниже приведен пример простого класса.

Листинг 1.2. Простой класс в Dart

```
class Greeter {  ← Ключевое слово class определяет новый класс
  var greeting;  ← Открытое свойство
  var _name;     ← Имена закрытых свойств начинаются с символа _

  sayHello() {   ← Открытый метод
    return "$greeting ${this.name}"; ← Строковая интерполяция
  }

  get name => _name;          | Синтаксис сокращенной записи методов чтения
  set name(value) => _name = value; | и установки
}

main() {
  var greeter = new Greeter(); ← Ключевое слово new создает новый экземпляр класса Greeter
  greeter.greeting = "Hello ";  | Синтаксис присваивания значений полям и свойствам
  greeter.name = "World";       | одинаков
  print(greeter.sayHello());
}
```

В этом простом классе заключено немало интересного. Закрытые члены обозначаются добавлением знака подчеркивания `_` в начало имени. Это принятое в Dart соглашение позволяет читателю кода сразу сказать, что доступ к методу или свойству осуществляется в закрытой области видимости.

Синтаксис методов чтения и установки также полезен, потому что позволяет одинаково работать как с полями, так и со свойствами. Поэтому проектировщик класса может сначала раскрыть поле (как в случае `greeting`), а затем преобразовать его в свойство, к которому можно обращаться только с помощью методов чтения и установки — аксессоров (как в случае `name`); при этом никаких изменений в вызывающий код вносить не придется.

Ключевое слово `this`, которое вызывает столько путаницы в JavaScript, тоже используется традиционным для ООП способом. Оно ссылается на конкретный экземпляр самого класса, а не на владельца класса в данный момент времени (как в JavaScript).

Классы факультативны

В отличие от Java и C#, классы в Dart факультативны. Функции могут находиться в области видимости верхнего уровня, не будучи членами классами. Иными словами, объявлять класс, лишь для того чтобы создать функцию, необязательно. Если выясняется, что класс содержит только служебные методы, то, возможно, заводить его не стоило, а лучше было определить функции на верхнем уровне.

1.2.4. Определение подразумеваемого интерфейса

В Dart, как и в Java и C#, имеются интерфейсы, но для их определения применяется структура класса. Предполагается, что любой класс неявно определяет интерфейс, включающий его открытые члены. В листинге 1.3 определены класс `Welcomer` и функция верхнего уровня `sayHello()`, которая принимает в качестве параметра экземпляр `Welcomer`. Помимо ключевого слова `extends` для реализации наследования в смысле Java и C#, можно также использовать определяемый классом интерфейс с помощью ключевого слова `implements`. Класс `Greeter` реализует открытые методы класса `Welcomer`, что дает возможность использовать его вместо `Welcomer`. Таким образом, программист может принимать во внимание только интерфейс класса, а не его конкретную реализацию.

Листинг 1.3. У любого класса есть неявный интерфейс

```
class Welcomer {  
    printGreeting() => print("Hello ${name}");  
    var name;  
}
```

Классу `Welcomer` можно унаследовать...

```
class Greeter implements Welcomer {  
    printGreeting () => print("Greetings ${name}");  
    var name;  
}
```

...но у него имеется также неявный интерфейс, реализуемый классом `Greeter`

```
void sayHello(Welcomer welcomer) {  
    welcomer.printGreeting();  
}
```

← Ожидается аргумент типа `Welcomer`

```
main() {  
    var welcomer = new Welcomer();  
    welcomer.name = "Tom";  
}
```

```
sayHello(welcomer);

var greeter = new Greeter();
greeter.name = "Tom";
sayHello(greeter);
```

← Поскольку Greeter реализует интерфейс Welcomer, его можно использовать вместо Welcomer

Способность реализовывать класс, не имеющий явно определенного интерфейса, – весьма полезная особенность Dart. Благодаря ей можно без особого труда создавать подставные классы (mocks) или предоставлять собственную реализацию класса; необязательно явно наследовать общему базовому классу.

1.2.5. Фабричные конструкторы для предоставления реализации по умолчанию

Помимо синтаксиса конструкторов, аналогичного Java и C#, в Dart имеется понятие фабричного конструктора (factory constructor). Можно определить базовый класс, используемый как интерфейс, и включить в него фабричный конструктор, который предоставит конкретный экземпляр по умолчанию. Это особенно полезно, когда в большинстве случаев требуется всего одна реализация некоторого интерфейса.

В листинге 1.4 приведен пример класса IGreetable, в котором имеется фабричный конструктор, возвращающий экземпляр класса Greeter. Класс Greeter реализует интерфейс IGreetable и позволяет пользователям этого интерфейса использовать реализацию Greeter, предоставляемую по умолчанию, не зная о том, что именно они используют. Таким образом, проектировщик класса Greeter может изменить конкретную реализацию, а пользователи интерфейса IGreetable об этом не узнают.

Листинг 1.4. Фабричные конструкторы предоставляют реализации по умолчанию

```
abstract class IGreetable {
    String sayHello(String name);
```

← Определяется интерфейс
← Объявляется метод, который необходимо реализовать

```
    factory IGreetable() {
        return new Greeter();
    }
```

Фабричный конструктор
возвращает
экземпляр Greeter

```

}

class Greeter implements IGreetable {
  sayHello(name) {
    return "Hello $name";
  }
}

void main() {
  IGreetable myGreetable = new IGreetable();
  var message = myGreetable.sayHello("Dart");
  print(message);
}

```

Greeter
реализует
интерфейс
IGreetable

Создается экземпляр IGreetable, который
← возвращает реализацию Greeter
← Используется реализация Greeter

Важно отметить, что благодаря этой возможности многие встроенные классы на самом деле являются интерфейсами – в частности, `String` и `int`. У них есть конкретные реализации, возвращаемые фабричными конструкторами. Я буду рассматривать классы, интерфейсы и их взаимодействие с факультативной системой типов во второй части книги.

1.2.6. Библиотеки и область видимости

В Dart имеется возможность организовывать логические наборы исходных файлов. Конечно, можно поместить все Dart-приложение в единственный файл с расширением `.dart`, но это плохо с точки зрения организации кода и навигации по нему. Для решения подобной проблемы в язык Dart включено понятие библиотеки. В Dart библиотекой называется совокупность исходных файлов, которые можно было бы объединить в один файл, но это не сделано, чтобы человеку было проще работать с кодом.

Выше я уже отмечал, что классы в Dart факультативны. Так сделано, чтобы функции можно было определять на верхнем уровне библиотеки. В Dart библиотека представляет собой один или несколько файлов с расширением `.dart`, сгруппированных исходя из каких-то логических соображений; в каждом файле может быть несколько классов (в том числе нуль) и несколько функций верхнего уровня (в том числе нуль). Библиотека Dart может импортировать другие библиотеки, необходимые для работы включенного в нее кода.

Для определения библиотеки служит ключевое слово `library`, для импорта библиотек – ключевое слово `import`, а для ссылки на другие

исходные файлы – ключевое слово `part`. Все это иллюстрируется в листинге ниже.

Листинг 1.5. Библиотеки и исходные файлы

```
library "my_library";    ← Объявляется, что этот файл – библиотека

import "../lib/my_other_library.dart"; ← Импортируется библиотека из другой папки

part "greeter.dart";    | Включаются другие исходные файлы (в частности, содержащий класс
part "leaver.dart";    | Greeter)

greetFunc() {           ← Определяется функция в библиотечной области видимости верхнего уровня
  var g = new Greeter(); ← Используется класс из файла greeter.dart
  sayHello(g);           ← Вызывается функция из области видимости верхнего уровня библиотеки
}                         my_other_library
```

Из этого листинга видно, что можно определить метод в области видимости верхнего уровня, то есть не являющийся частью какого-либо класса. Поэтому класс следует создавать, только если может потребоваться экземпляр объекта, а не просто как контейнер для группы взаимосвязанных функций.

Библиотека может включать группу исходных файлов в одну область видимости. При этом число исходных файлов произвольно (в том числе нуль), и библиотека, включающая несколько отдельных файлов, эквивалентна библиотеке из одного файла, содержащего те же самые файлы внутри себя. Поэтому каждый исходный файл может ссылаться на код, находящийся в другом исходном файле, при условии что оба являются частями одной и той же библиотеки. Кроме того, любой исходный файл может обращаться к коду, который находится в импортированной библиотеке, как показано на примере файла `my_other_library.dart` на рис. 1.3.

Чтобы избежать конфликтов имен, которые могли бы возникнуть, если написанная вами и импортированная библиотека обе содержат класс с именем `Greeter`, можно назначить библиотеке префикс:

```
import "../lib/my_other_library.dart" as other;
```

а затем ссылаться на содержащиеся в ней классы следующим образом:

```
other.otherLibFunction("blah");
```

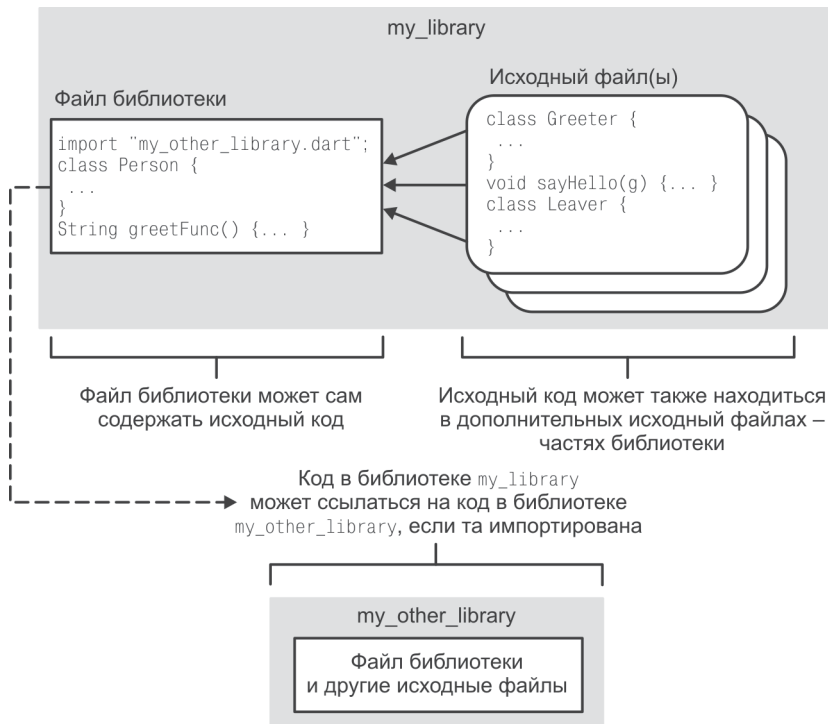


Рис. 1.3. Файл `my_library.dart` состоит из файлов `greeter.dart` и `leaver.dart` и использует другую библиотеку с именем `my_other_library.dart` (которая, в свою очередь, состоит из нескольких исходных файлов)

Это позволяет присваивать классам и методам разумные имена, не опасаясь засорить глобальное пространство имен.

Библиотека может содержать точку входа в приложение, но тогда в ней должна быть функция `main()`. Такая функция может присутствовать в нескольких библиотеках, но выполняется та, что находится в библиотеке, упомянутой в теге `<script>`.

Все функции и классы, содержащиеся в вашей библиотеке, становятся доступны любой импортирующей ее программе, то есть являются открытыми. Чтобы запретить импортирующей программе доступ к отдельным функциям или классам, пометьте их как закрытые.

Закрытость в классах и библиотеках

Хотя библиотеки и классы полезны для обеспечения модульности приложения, рекомендуется скрывать внутренние механизмы их работы. По счастью, в Dart имеется простой способ закрыть внутренности от любопытного взгляда: достаточно поместить в начало имени метода, функции, класса или свойства знак подчеркивания (`_`). В результате элемент становится доступен только изнутри самой библиотеки.

Закрытость распространяется только на клиентов библиотеки

Вернемся к предыдущему примеру. Если классы `Greeter` и `Leaver` являются частями одной и той же библиотеки, то они могут обращаться к закрытым элементам друг друга (по аналогии с конструкцией `package private` в Java). Это означает также, что свойство или функция `_greeterPrivate()` доступна из любого другого класса в той же библиотеке. Но если класс `Greeter` импортируется в другую библиотеку, то в последней этот метод не виден (рис. 1.4).

Закрытыми могут быть функции верхнего уровня библиотеки, классы, поля, свойства и методы классов (сообща называемые членами), а также конструкторы классов. Внутри библиотеки закрытость игнорируется, то есть из любого файла-части, включенного с помощью ключевого слова `part`, можно обращаться к закрытым элементам в любой другой части той же самой библиотеки `library`. Однако клиенты библиотеки не могут обращаться к ее закрытым элементам (как и к закрытым элементам классов, определенных в этой библиотеке).

1.2.7. Функции как полноценные объекты

В Dart с функциями можно обращаться, как с объектами, — так же, как в JavaScript: разрешается передавать функцию в виде параметра, сохранять функцию в переменной и использовать анонимные (безымянные) функции для обратных вызовов. В следующем листинге приведены примеры разнообразного употребления функций.

Листинг 1.6. Функции как полноценные объекты

```
String sayHello(name) => "Hello $name";
```

 ← ❶ Сокращенный синтаксис объявления функции

```
main() {
```

```
  var myFunc = sayHello;
```

```
  // ...
```

 ← ❷ Функция присваивается переменной

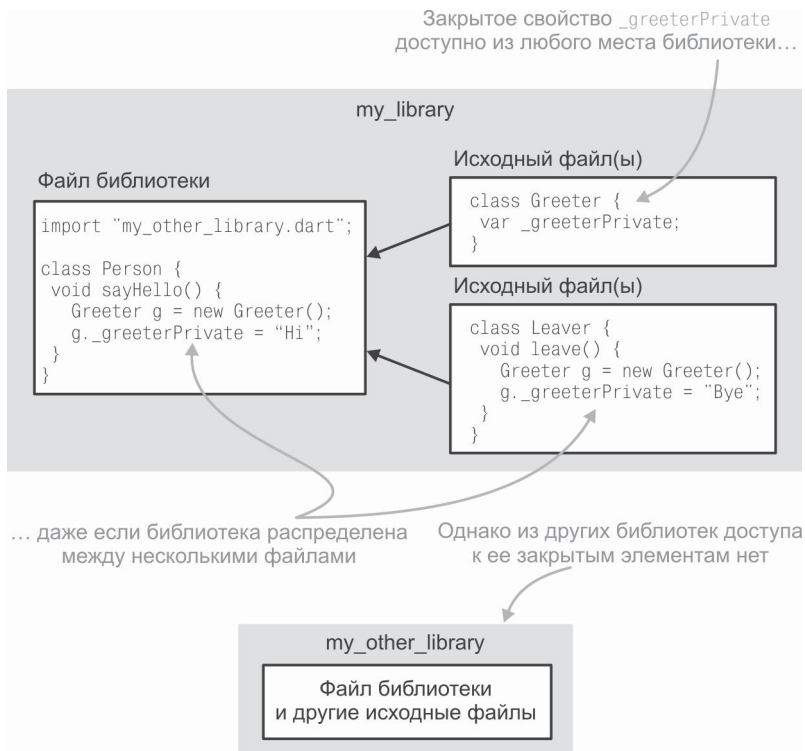


Рис. 1.4. Закрытые элементы – поля, методы, библиотечные функции и классы – скрыты внутри библиотеки. Закрытость обозначается префиксом `_`. Пользователи библиотеки не имеют доступа к закрытым элементам

```
print(myFunc("World"));    ← Вызывается функция, хранящаяся в переменной

var mySumFunc = (a,b) {    ← Определяется анонимная функция
  return a+b;
};

var c = mySumFunc(1,2);    ← Вызывается анонимная функция
print(c);
}
```

❶ – это однострочная функция, возвращающая значение; она определена с помощью сокращенного синтаксиса. Следующие две функции идентичны:

```
String sayHello(name) {  
    return "Hello $name";  
}
```

```
String sayHello(name) => "Hello $name";
```

Определив функцию, мы можем получить ссылку на нее и сохранить ее в переменной **❷**. Эту переменную можно передавать, как любое другое значение. Анонимные функции наподобие той, что сохранена в переменной `mySumFunc`, часто применяются в качестве обработчиков событий. Нередко можно встретить такой код:

```
myButton.on.click.add((event) {  
    // что-то сделать  
});
```

Здесь анонимная функция выделена полужирным шрифтом.

1.2.8. Параллелизм с помощью изоляторов

Dart – однопоточный язык. И хотя такое решение может показаться идущим вразрез с современной аппаратной архитектурой, в которой доступных приложению процессоров становится все больше и больше, оно упрощает написание программ на Dart.

В Dart единицей работы является изолятор (*isolate*), а не процесс или поток. У каждого изолятора имеется собственная область памяти (разделение памяти между изоляторами не допускается), что облегчает реализацию изолированной модели защиты. Один изолятор может передавать сообщения другому. Когда изолятор получает сообщение (оно может содержать и данные), вызывается обработчик события, который обрабатывает это сообщение точно так же, как сообщение нажатия кнопки. Изолятор-приемник получает копию сообщения, посланного изолятором-отправителем. Изменения полученных данных не видны отправляющей стороне, получатель должен отправить ответное сообщение.

На веб-странице каждый скрипт (содержащий функцию `main()`) выполняется в отдельном изоляторе. Различные части могут быть реализованы разными скриптами; например, один читает новостную ленту, другой занимается синхронизацией автономного хранилища и т. д. Программа на Dart может создать новый изолятор по анало-

гии с тем, как в Java или C# запускается новый поток. Но изолятор отличается от потока тем, что обладает собственной памятью. Не существует никакого способа предоставить общий доступ из нескольких потоков к одной переменной; единственный способ коммуникации между изоляторами – передача сообщений.

Изоляторы используются также для динамической загрузки внешнего кода. Код, не являющийся частью приложения, можно загрузить в защищенную область и исполнять независимо от приложения, организовав коммуникацию с помощью обмена сообщениями. Такое поведение идеально для разработки архитектуры подключаемых модулей.

В реализации виртуальной машины Dart можно при необходимости использовать несколько процессорных ядер для запуска изоляторов. А при трансляции изоляторов на JavaScript они становятся рабочими веб-процессами в смысле HTML5.

Памятка

- В Dart имеется факультативная типизация (для документирования).
 - Библиотеки позволяют организовывать код, распределяя его между несколькими исходными файлами.
 - Механизм ограничения доступа встроен в язык.
 - Функции – полноценные объекты и могут быть определены вне классов.
 - В Dart реализован параллелизм с помощью изоляторов, обменивающихся сообщениями.
-

Бегло познакомившись с основами Dart, мы теперь можем посмотреть, как он применяется в веб-приложениях.

1.3. Веб-программирование на языке Dart

Одна из целей Dart – облегчить жизнь разработчикам. А поскольку Dart – в первую очередь язык программирования для веб, то проектировщики немало постарались, чтобы сделать API манипулирования моделью DOM браузера максимально удобным. В JavaScript доступ к DOM был сущим кошмаром, пока не появилась библиотека jQuery, сделавшая работу с браузером простой и естественной. По аналогии с ней и была написана библиотека `dart:html`.

1.3.1. *dart:html: удобная библиотека для работы с моделью DOM браузера*

На момент написания этой книги для Dart еще не было библиотеки виджетов пользовательского интерфейса. И хотя команда Dart открыто заявила, что Dart будет языком типа «все включено», пока идет работа над ранними публичными версиями, больше времени приходится уделять совершенствованию самого языка, а не высокоуровневым абстракциям. Однако уже написана библиотека `dart:html`, которая рассматривается как эквивалент ядра `jQuery`.

Если вам уже доводилось работать с `jQuery` или другим подобным каркасом, то вы знаете, как CSS-селекторы используются для доступа к элементам DOM, например `DIV`’у с `id="myDiv"` или ко всем элементам `<p>`. Библиотека `dart:html` упрощает эту задачу. Вместо того чтобы вызывать различные функции получения элементов, например `getElementsById()` или `getElementsByName()`, как то делается при работе со встроенными в браузер API, в `dart:html` используются всего два метода выборки элементов: `query()`, который возвращает один элемент, и `queryAll()`, который возвращает список элементов. А поскольку используются списки Dart, к ним применимы все стандартные функции работы со списками, в частности `contains()` и `isEmpty()`, а также синтаксис доступа к массиву: `element.children[0]`. В следующем листинге приведены примеры работы с DOM с помощью библиотеки `dart:html`.

Листинг 1.7. Взаимодействие с браузером

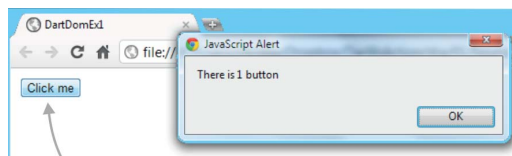
```
import 'dart:html';    ← Импортируется библиотека dart:html

void main() {
  var button = new Element.tag("button");    ← Создается новый элемент button
  button.text = "Click me";
  button.onClick.add((event) {
    List buttonList = queryAll("button");
    window.alert("There is ${buttonList.length} button");
  });
  document.body.children.add(button);    ← Кнопка добавлена в тело HTML-документа
}
```

Добавляется анонимная
 функция (выделена
 полужирным курсивом),
 которая обрабатывает
 событие `onClick`

Здесь для создания кнопки используется именованный конструктор. С кнопкой связывается обработчик события (в виде анонимной

функции), после чего она добавляется в тело документа. При запуске этого примера мы наблюдаем картину, показанную на рис. 1.5.



Нажатие кнопки приводит к вызову обработчика события, который был добавлен методом `button.onClick.add (...)`

Рис. 1.5. Результат нажатия кнопки, созданной в листинге 1.7

При таком взаимодействии с браузером можно создать сложный пользовательский интерфейс, применяя только Dart и CSS. Мы рассмотрим эту тему в главе 4.

Совет. При написании кода для браузера не забывайте, что функция `print`, использованная в предложении `print("Hello World")`, выводит результат на консоль браузера, а не на страницу. В Chrome для доступа к консоли браузера нужно выбрать пункт **Tools** ⇒ **JavaScript Console** (Инструменты ⇒ Консоль JavaScript) из меню «Настройка и управление».

1.3.2. Dart и HTML5

Библиотека `dart:html` позволяет не только непосредственно взаимодействовать с моделью DOM браузера, но и работать с такими элементами HTML5, как холст, WebGL, события перемещения устройства и данные геолокации. Картинка, изображенная на рис. 1.6, порождена кодом из листинга 1.8, в котором используется HTML5 Canvas API.

В коде рисования на холсте мы сначала добавляем тег HTML5 `<canvas>` в DOM, а затем получаем от него двумерный контекст рисования. После этого в контекст выводятся текст и фигуры.

Листинг 1.8. Рисование на холсте браузера

```
import 'dart:html';
import 'dart:math';

void main() {
  CanvasElement canvas = new Element.tag("canvas");
```

Создается новый элемент
← CanvasElement

```
canvas.height = 300;
canvas.width = 300;
document.body.children.add(canvas);

var ctx = canvas.getContext("2d");

ctx.fillText("hello canvas", 10, 10);

ctx.beginPath();
ctx.arc(50, 50, 20, 0, PI * 2, true);
ctx.closePath();
ctx.fill();
}
```

← Холст добавляется в тело документа

← Получаем от холста контекст рисования

← Выводится текст

Рисуется
закрашенный круг

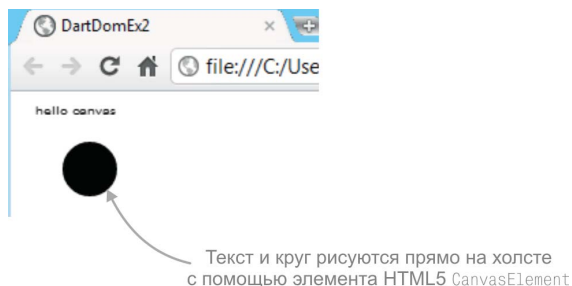


Рис. 1.6. Рисование на холсте браузера

Создав и добавив на страницу элемент `CanvasElement`, мы получаем область, в которой можно рисовать с помощью стандартных методов типа `drawImage`, `fillText` и `lineTo`. Подробнее мы будем рассматривать эту тему в главе 10.

Библиотека `dart:html` дает нам доступ ко всем стандартным элементам браузера. А поскольку библиотека DOM, которая составляет часть `dart:html`, сгенерирована из описания на языке определения интерфейсов WebKit IDL, мы имеем доступ ко всей функциональности современных браузеров, предоставляемой Dart.

Памятка

- Библиотека `dart:html` – это взгляд на модель DOM браузера со стороны Dart.
- Поддержка HTML5 – неотъемлемая часть языка Dart.

Итак, у нас есть некоторое представление о том, как Dart исполняется в браузере. Теперь самое время познакомиться с инструментами, помогающими писать программы на этом языке.

1.4. Инструментальная экосистема Dart

Инструменты Dart считаются частью платформы Dart и потому развиваются так же быстро, как сам язык. Разработчики обычно осваивают язык с помощью имеющихся (или отсутствующих) инструментов, и Google тратит на этот аспект немало сил. Начнем с редактирования кода.

1.4.1. Редактор Dart

Хотя писать программы на Dart можно в любом текстовом редакторе, лучше всего для этой цели приспособлен редактор Dart. Он построен на платформе Eclipse Rich Client Platform (RCP) – каркаса для создания редакторов кода. Редактор Dart располагает такими привычными средствами, как автозавершение кода, навигация, выделение структуры кода, а также умеет производить статический анализ кода, выдавая предупреждения и сообщения об ошибках. Инструмент статического анализа доступен также в виде командной утилиты, которая может включаться в систему непрерывной интеграции для обнаружения ошибок на ранних стадиях. На рис. 1.7 показаны некоторые средства редактора Dart.

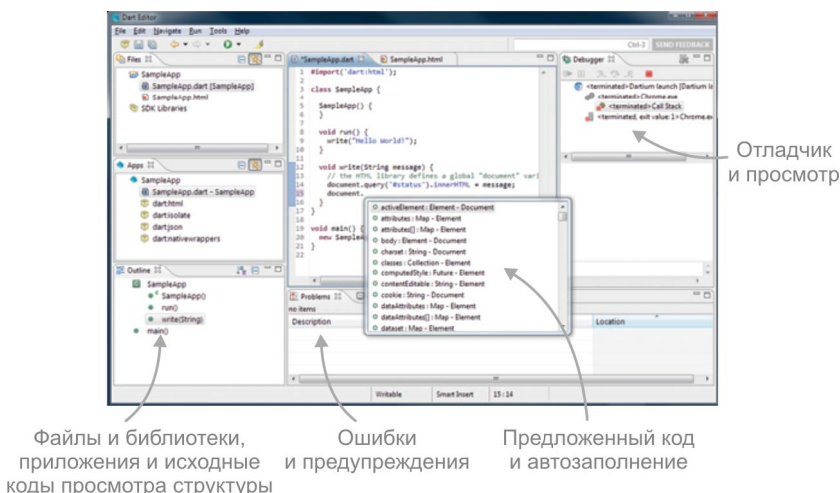


Рис. 1.7. Редактор Dart Editor, показано простое приложение для браузера и окно автозавершения кода

Редактор Dart позволяет писать код, а если этот код ассоциирован с HTML-страницей, то его можно преобразовать в JavaScript и исполнить в браузере по своему выбору с помощью инструмента dart2js. Если используется браузер Dartium – Chrome со встроенной VM Dart, – то этап преобразования в JavaScript можно опустить и исполнять Dart-код непосредственно. Dartium также умеет общаться с редактором Dart Editor, что дает возможность сквозной пошаговой отладки.

Если код не связан ни с какой HTML-страницей, то редактор может имитировать его запуск из командной строки с выводом на консоль `stdout`.

1.4.2. Виртуальная машина Dart

VM Dart – неотъемлемая часть языка Dart. Одно из ее применений – исполнение Dart-кода из командной строки, например запуск HTTP-сервера или простого скрипта (аналог пакетного файла или скрипта оболочки). Другое – встраивание в объемлющее приложение типа браузера Dartium.

1.4.3. Dartium

Dartium – это специализированная версия браузера Chromium (вариант Google Chrome с открытым исходным кодом), в которую встроена VM Dart. Он распознает тип скрипта `application/dart` и исполняет написанный на Dart код непосредственно без промежуточного преобразования в JavaScript. В Dartium включены инструменты разработчика, знакомые многим программистам, которые создают сайты и веб-приложения в Chrome. В сочетании с редактором Dart он позволяет вести пошаговую отладку: вы можете поставить точки останова в редакторе, а затем обновить страницу приложения в Dartium; встроенный в редактор отладчик остановится в достигнутой точке и позволит просмотреть переменные и продолжить исполнение в пошаговом режиме.

Благодаря браузеру Dartium разработка на Dart становится не сложнее, чем на JavaScript. Обновление страницы в браузере – все, что нужно для исполнения Dart-кода.

1.4.4. dart2js: конвертер Dart в JavaScript

Инструмент dart2js служит для компиляции Dart в JavaScript и вызывается из редактора Dart Editor или из командной строки

в виде автономной утилиты. Он компилирует все библиотеки и исходные файлы, составляющие Dart-приложение, в один JavaScript-файл. Результирующий код вполне понятен, хотя в случае, когда разработка на Dart ведется с применением Dartium, читать его вряд ли имеет смысл.

dart2js также включает в JavaScript ссылки на исходный код, позволяющие перейти от порожденного JavaScript-кода к коду на Dart, из которого он получен. Это недавнее дополнение успешно используется и в других языках, транслируемых на JavaScript, в частности CoffeeScript и Google Web Toolkit (GWT).

Примечание. dart2js – третий конвертер Dart в JavaScript. Первый назывался dartc, второй – frog. Эти названия встречаются в старой документации и в блогах.

1.4.5. Управление пакетами с помощью pub

Управление пакетами – важная черта любого языка. В Java для этой цели используется Maven, в .NET – NuGet, а в node.js – npm. В Dart есть собственный менеджер пакетов, который называется pub. Pub позволяет разработчику определить метаданные пакета в файле pubspec и опубликовать библиотеки в таких репозиториях, как GitHub.

Когда вам понадобится какая-то библиотека, вы можете с помощью pub загрузить ее вместе со всеми зависимостями с учетом номеров версий. Мы подробнее обсудим pub и приведем соответствующие примеры в главе 5, когда будем говорить о структуре библиотеки, написанной на Dart.

Памятка

- Инструментальная экосистема составляет неотъемлемую часть проекта Dart.
 - Редактор Dart предлагает разработчикам развитые инструментальные средства.
 - Благодаря Dartium разработка на Dart оказывается не сложнее обновления страницы в браузере.
 - Dart спроектирован так, что легко транслируется на JavaScript.
-

1.5. Резюме

На первый взгляд может показаться, что Dart – просто очередной язык программирования. Но если рассматривать его в контексте всей окружающей экосистемы, то окажется, что Dart открывает блистательные перспективы в мире веб-разработки. Современные приложения становятся все более сложными, для их создания необходимы крупные коллективы разработчиков, и Dart вместе со своими инструментами и средами обещает внести некую структуру в слишком уж гибкий мир JavaScript.

Создавать одностраничные приложения, работающие в браузере (к примеру, Google Plus), на Dart гораздо легче, потому что сопровождение объемного клиентского кода перестает быть чрезмерно хрупким. Язык Dart – который можно исполнять непосредственно или транслировать на JavaScript – вкупе с HTML5 – идеальное решение для построения веб-приложений, не нуждающихся во внешних подключаемых модулях.

В следующих главах мы ближе познакомимся с экосистемой Dart, изучим во всех подробностях ядро языка и воспользуемся Dart для разработки одностраничных веб-приложений, ориентированных на современные веб-браузеры с поддержкой HTML5. Прочитав эту книгу, вы сможете создавать на Dart приложения, способные работать на стороне клиента без подключения к сети, загружаемые с файлового сервера Dart и подключающиеся к Dart-серверу для сохранения данных в базе.



Глава 2. Программа «Здравствуй, мир» на Dart

В этой главе

- ☐ Знакомство с набором инструментов для Dart.
- ☐ Создание приложения в редакторе Dart.
- ☐ Инструменты для отладки Dart-приложений.
- ☐ Исполнение приложений в браузере.
- ☐ Развертывание приложений для браузеров, поддерживающих JavaScript.

Dart располагает развитой инструментальной экосистемой, которая позволяет писать, запускать и отлаживать Dart-код в различных средах. В этом отношении Google придерживается подхода «все включено» и намеревается предоставить все инструменты, необходимые для создания сложных приложений. В настоящее время инструментарий для Dart доступен на платформах Windows, Mac и Linux; скачать их можно с сайта dartlang.org, поддерживаемого Google.

Набор инструментов для Dart начинается с виртуальной машины (ВМ) Dart. Она существует в двух ипостасях. Во-первых, это серверная ВМ, которая позволяет выполнять программы из командной строки – как в Java, Python или Ruby; именно этим вариантом мы воспользуемся для запуска простого приложения «Здравствуй, мир». Во-вторых, это ВМ, встроенная в Dartium – специализированную версию браузера Chrome, имеющую доступ к объектной модели документа (DOM). С Dartium мы познакомимся, когда начнем использовать редактор Dart для написания и запуска того же приложения в браузере; при этом Dart-скрипт будет включен в HTML-файл.

Мы будем также пользоваться редактором Dart, который применяется для создания приложений, работающих как в браузере, так и на сервере, и посмотрим, как в нем производится отладка и рефакторинг приложения «Здравствуй, мир». Здесь же мы воспользуемся инструментом `dart2js`, который компилирует все множество исходных файлов и библиотек, написанных на Dart, в единственный JavaScript-файл.

Вы узнаете об инструменте `dartdoc`, который генерирует отформатированную документацию по API в формате HTML, исходя из помещенных в код комментариев. Это именно та программа, которой Google пользуется для генерации документации по API Dart на своем сайте. Мы с ее помощью сгенерируем документацию по приложению «Здравствуй, мир».

Выбирая Dart в качестве платформы для создания веб-приложений, вы получаете не только язык, но и все инструменты, разработанные Google для Google. Скачайте редактор Dart, вместе с которым упакованы все необходимые инструменты, в том числе комплект разработки ПО (SDK) и браузер Dartium, и приступим к делу.

2.1. VM Dart для командных приложений

Существуют версии VM Dart для Windows, Mac и Linux. Это позволяет писать Dart-код, который работает без браузера и имеет доступ ко всем средствам компьютера, включая файловую систему и сетевой ввод-вывод на уровне сокетов. В виртуальной машине применяется асинхронная, событийно-управляемая модель, аналогичная принятой в Node.js (это серверная VM, основанная на движке JavaScript, используемом в Chrome). Событийно-управляемая модель подразумевает, что ваш код исполняется до конца, после чего управление передается циклу обработки событий, который уведомляет вашу программу, когда произойдет что-то интересное, например открытие файла или получение данных из TCP-сокета.

Начнем с простого Dart-скрипта – тривиальной программы «Здравствуй, мир», которая состоит из одной строки, печатающей на консоли строку “Hello World”.

Листинг 2.1. HelloWorld.dart

```
main() => print("Hello World");
```

← Строка печатается на stdout

Однострочная функция `main()` является точкой входа в любое Dart-приложение. Ее можно записать и более традиционно – заключив тело функции в фигурные скобки:

```
main() {  
    print("Hello World");  
}
```

Чтобы выполнить этот код, сохраните его в текстовом файле с расширением `.dart` и передайте этот файл в качестве параметра интерпретатору Dart, который находится в папке `bin\` скачанного SDK Dart. В Windows этот файл называется `dart.exe`. Результат выполнения показан на рис. 2.1.

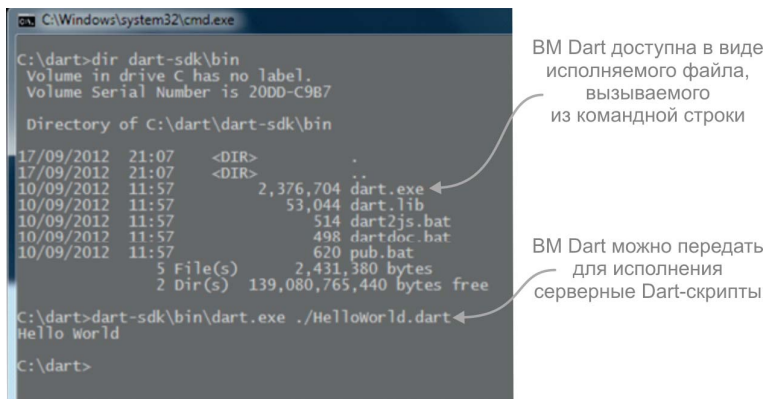


Рис. 2.1. BM Dart в виде командной утилиты

BM Dart можно встроить и в другие приложения, но эта тема выходит за рамки настоящей книги. В качестве примера назовем среду исполнения Dart-скриптов в текстовом редакторе по аналогии со средой исполнения Python, встроенной в такой популярный редактор, как Sublime Text.

Хотя Dart-скрипты можно выполнять из командной строки – именно так запускаются серверные Dart-приложения в производственных системах, – в качестве средства разработки такая среда далека от идеала. По счастью, у нас есть редактор Dart, который позволяет исполнять прямо в редакторе командную ВМ, а заодно и ВМ, размещенную в браузере.

В части IV мы будем детально рассматривать серверное программирование на Dart, в том числе цикл обработки событий и доступ к ресурсам сервера, включая и файлы. Там мы напишем веб-сервер и воспользуемся веб-сокетами HMTL5 для взаимодействия с Dart-кодом, работающим в браузере.

Памятка

- BM Dart доступна в виде двоичного файла, исполняемого на сервере из командной строки.

- Серверная VM Dart имеет полный доступ к серверным API, в том числе файловой системе и сетевым сокетам.

2.2. Программа «Здравствуй, мир» в редакторе Dart

В состав редактора Dart входит немало инструментов, необходимых для разработки и отладки написанного на Dart кода, и число их постоянно растет. Упомянем, в частности, навигацию по коду, автозавершение, рефакторинг, а также предупреждения и сообщения об ошибках. Хотя код на Dart – как и на многих других динамических языках – можно писать практически в любом текстовом редакторе, создатели Dart приложили массу усилий к тому, чтобы предоставить полноценную среду разработки, которой зачастую так не хватает другим альтернативам JavaScript.

Мы начнем с использования редактора Dart для запуска написанного ранее скрипта HelloWorld.dart в серверной VM Dart, но на этот раз запустим ее прямо из редактора. Затем мы модифицируем скрипт, так чтобы его можно было разместить на HTML-странице, и запустим его в браузерной VM Dart, которая встроена в браузер Dartium, поставляемый в комплекте с редактором Dart.

На рис. 2.2 показан редактор Dart, с которым мы будем работать на всем протяжении этой книги.

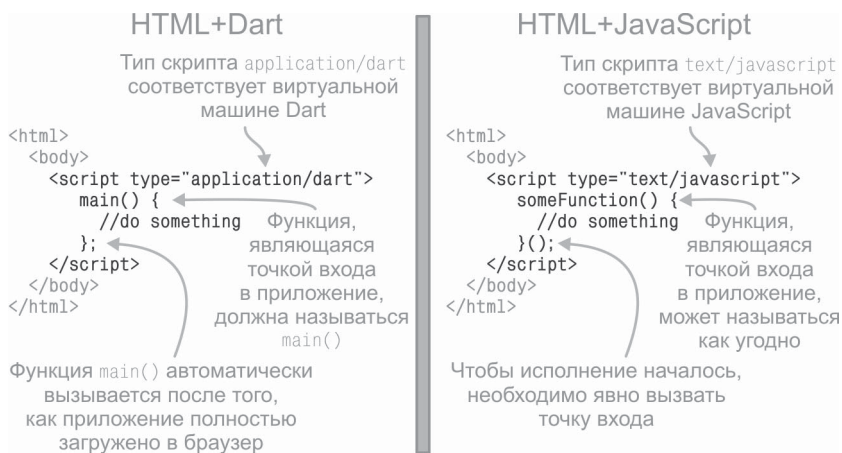


Рис. 2.2. Редактор Dart – облегченный вариант среды Eclipse

2.2.1. Знакомство с инструментами, встроенными в Редактор Dart Editor

Редактор Dart располагает стандартным набором инструментов разработки, в том числе автоматическим завершением имен методов и свойств, рефакторингом, навигацией по коду, проверкой синтаксиса и отладкой. Когда вы работаете с приложением в редакторе Dart, весь код, составляющий приложение, – ваш и внешних библиотек – проверяется на наличие ошибок и предупреждений. Для выполнения статического анализа кода редактор Dart пользуется инструментом `dart_analyzer`. Тот же инструмент можно вызвать и из командной строки, например для включения в систему непрерывной интеграции типа Hudson, CruiseControl или Bamboo.

Примечание. На момент написания этой книги командная версия `dart_analyzer` была доступна только на платформах Mac и Linux. Версия Windows находится в стадии разработки.

Информацию, генерируемую `dart_analyzer`, редактор Dart использует для подсветки строчек кода, в которых обнаружены ошибки или предупреждения (рис. 2.3).

Попытка конкатенировать две строки с помощью оператора `+` приводит к предупреждению. Редактор позволит выполнить такой код, но во время выполнения может быть возбуждено исключение

Откровенная опечатка – ввод `prnt()` вместо `print()` – подсвечивается как ошибка

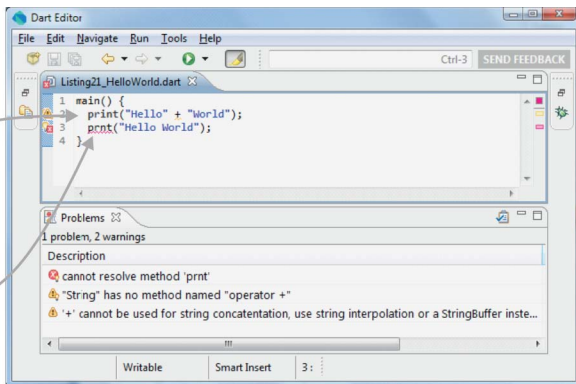
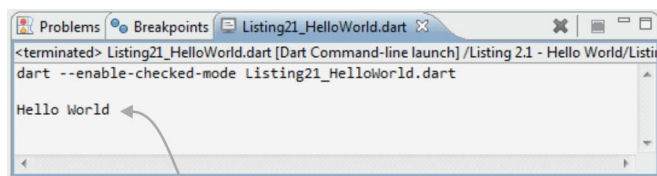


Рис. 2.3. Ошибки и предупреждения в редакторе Dart

Механизм проверки ошибок заодно наделяет редактор Dart некоторыми средствами оперативного исправления. Например, редактор предоставляет возможность создать новую функцию-заглушку `prnt(String string)`, исходя из того, что ему известно об отсутствующей

щем методе `print()`. Предупреждение на рис. 2.3 говорит, что у объекта `String` отсутствует метод `+`. Тем не менее такой код может быть корректен, как мы увидим в главе 7 при рассмотрении функции `noSuchMethod()`, поэтому редактор выдает предупреждение, а не ошибку. Код с предупреждениями выполнить можно, код с ошибками – нельзя.

Информация, которую выводит скрипт «Здравствуй, мир», перенаправляется на консоль редактора Dart, а не печатается в отдельном окне консоли. Редактор Dart пользуется тем же самым двоичным кодом VM Dart, что и при запуске скрипта из командной строки. На рис. 2.4 показано, как выглядит результат работы скрипта в окне консоли редактора.



Вывод VM Dart показывается в окне консоли Редактора Dart

Рис. 2.4. VM Dart печатает на консоли редактора

В редактор Dart включен целый ряд инструментов навигации по коду – вашему собственному или внешних библиотек, полученных из Dart SDK или других источников открытого исходного кода.

Навигация по коду и стандартные библиотеки

Dart – проект с открытым исходным кодом. Этой идеологией пронизан весь язык, и код стандартных библиотек тоже открыт. Поэтому, если вы захотите посмотреть, как выглядит класс `ButtonElement` из `dart:html`, или познакомиться с кодом и комментариями к классу `List` из `dart:core`, то инструменты навигации по коду будут готовы показать весь текст по первому требованию.

Можно перейти из своего кода прямо в место объявления, воспользовавшись командой **Open Declaration (F3)** (Открыть объявление). Эта команда работает как для вашего собственного кода, так и для кода, находящегося в импортированных библиотеках. В частности, она полезна, когда нужно пройти по цепочке вызовов методов из нескольких библиотек, чтобы найти причину ошибки или понять,

как работает код, – особенно если вы работаете в команде и низко-уровневая библиотека написана кем-то другим или если библиотеки загружены из сети.

В редакторе Dart есть также полезные представления **Outline** (Структура) и **Callers** (Вызовы). В окне **Outline** показывается структура исходного файла: функции верхнего уровня, классы и члены классов. В окне **Callers**, ассоциированном с конкретной функцией или методом, видно, из каких мест программы эта функция вызывается. Это аналог команды **Find References** (Найти ссылки) в инструментах для Java и C#. На рис. 2.5 приведен пример.

В окне Outline показана структура исходного файла

В окне Callers показано, кто вызывает данную функцию или метод. В данном случае функция `foo()` вызывается из функции `main()` в строке 3

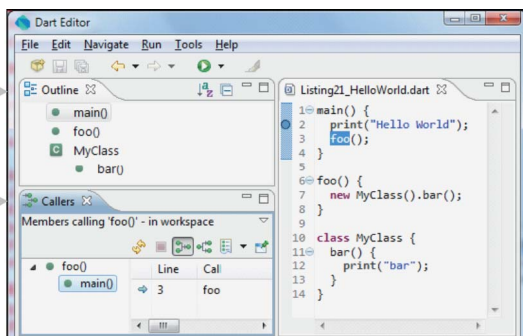


Рис. 2.5. Окна **Callers** и **Outline** в редакторе Dart

Автозавершение кода

Механизм автозавершения кода может и не понадобиться, если вы хорошо знакомы с исходным кодом и библиотеками, но он чрезвычайно полезен при ознакомлении с новой библиотекой и ее возможностями. Он сразу показывает, какие методы доступны в точке программы, на которую указывает курсор. Существует ли метод `Sort` в классе `List`? Есть ли возможность получить из объекта `String` коды символов в виде списка значений типа `char`? Благодаря механизму автозавершения вы сможете узнать об этом, не покидая редактора – собственно, даже не выходя из панели редактирования.

Эта возможность особенно ценна при работе с чужим кодом, написанным другим членом команды или взятым из внешних проектов. Конечно, если вы знаете все методы и их параметры наизусть, то автозавершение, возможно, и ни к чему, но современные каркасы с открытым исходным кодом меняются настолько быстро, что до-

ступ к информации о методах и их параметрах прямо в коде оказывается весьма полезен.

Навигация по коду, автозавершение и информация об ошибках – распространенные инструменты, к которым разработчики привыкли по прошлому опыту, так что их наличие никого не удивляет. Мы упомянули их только потому, что немногие редакторы JavaScript обладают такой функциональностью. Одна из проектных целей Dart – облегчить жизнь разработчикам веб-приложений, и описанные инструменты способствуют ее достижению.

2.2.2. Dart-скрипты и HTML-файлы

Язык Dart спроектирован для исполнения в браузере – непосредственно или после преобразования в JavaScript. Чтобы преобразование в JavaScript было возможно, приложение должно существовать отдельно от HTML-файла, определяющего объемлющую веб-страницу. К счастью, редактор Dart работает именно таким образом: при создании нового проекта в редакторе вам предлагается заодно создать стереотипный HTML, содержащий тег `script`, – для запуска приложения.

В листинге 2.2 показан минимальный HTML-код, необходимый для запуска Dart-приложения непосредственно или после преобразования в JavaScript. Он содержит тег `script` для загрузки скрипта нашего приложения `HelloWorld.dart`. Кроме того, имеется еще один тег `script` для загрузки файла `dart.js`, который определяет, поддерживает ли объемлющий браузер виртуальную машину Dart. Если это не так, то этот файл всюду заменяет тип скрипта `application/dart` на `application/javascript` и добавляет к файлу, указанному в свойстве `src`, суффикс `.js`: например, `HelloWorld.dart` преобразуется в `HelloWorld.dart.js`. Как мы вскоре увидим, JavaScript-версия Dart-приложения создается инструментом `dart2js`.

Листинг 2.2. HTML-файл, способный исполнять Dart-приложения непосредственно или в виде эквивалентного JavaScript-кода

<pre><!DOCTYPE html> <html> <body> <script type="application/dart" src="HelloWorld.dart"></script> <script src="http://dart.googlecode.com/ ↪ svn/branches/bleeding_edge/ ↪ dart/client/dart.js"></script> </body> </html></pre>	Скрипт, ссылающийся на файл с вашим Dart-приложением ← Dart-приложением JavaScript-файл <code>dart.js</code> перенаправляет браузеры, не поддерживающие Dart, на файл <code>HelloWorld.dart.js</code>

Файл `dart.js` содержит JavaScript-код, который проверяет, существует ли в браузере функция `navigator.webkitStartDart()`. Вы можете и сами проверить это условие, если хотите узнать, поддерживает ли браузер Dart.

На рис. 2.6 показано соотношение между объемлющим HTML-файлом и нашим Dart-скриптом «Здравствуй, мир».

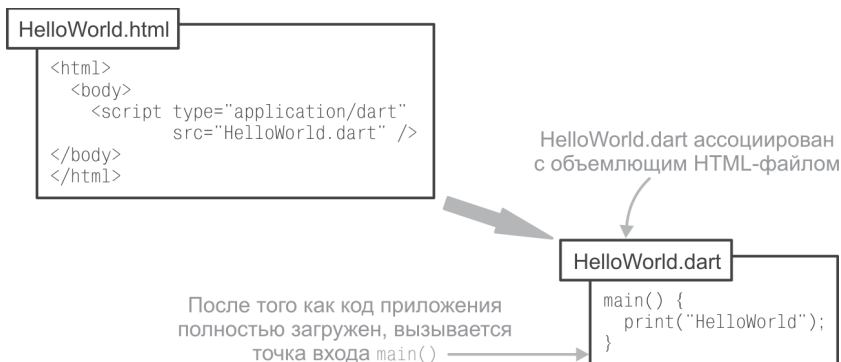


Рис. 2.6. HTML-страница загружает внешние Dart-файлы

2.2.3. Запуск приложения «Здравствуй, мир» в Dartium

Из редактора Dart это приложение можно запустить так же, как мы раньше делали с помощью командной версии VM Dart. Редактор Dart обнаруживает ассоциированный HTML-файл и загружает приложение в браузер Dartium; в качестве веб-сервера при этом используется сервер, встроенный в сам редактор. Этот встроенный сервер перечисляет все Dart-приложения, известные в данный момент редактору (рис. 2.7).

Приложение «Здравствуй, мир» выводит сообщение с помощью функции `print()`. При работе в браузере `print()` печатает на отладочной консоли браузера, как JavaScript-функция `console.log()`. Браузер Dartium умеет взаимодействовать с редактором Dart, поэтому любой текст, появляющийся на отладочной консоли Dartium, переправляется назад – на консоль редактора Dart. Dartium знает также о точках останова, установленных в редакторе Dart, поэтому при достижении точки останова исполнение скрипта в редакторе Dart прерывается. Мы рассмотрим процедуру отладки ниже в этой главе.

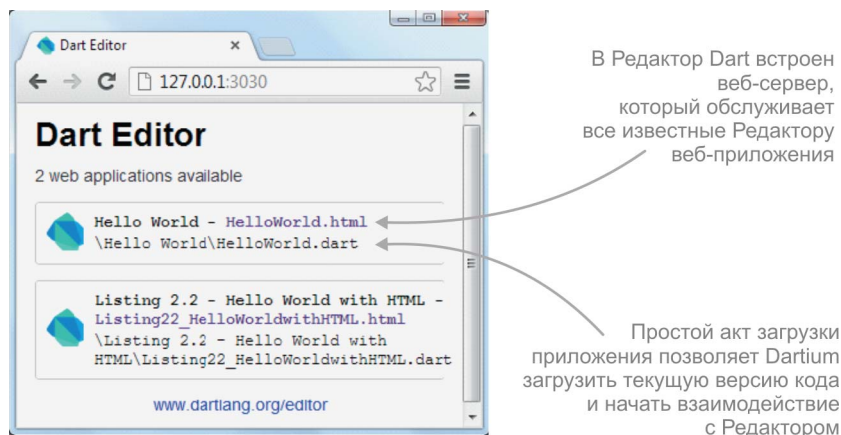


Рис. 2.7. Вид на встроенный в редактор Dart веб-сервер из браузера Dartium

2.2.4. Использование *dart2js* для конвертации в JavaScript

Если пользователь работает с браузером, который не поддерживает Dart непосредственно, то ему придется загрузить JavaScript-версию приложения. По счастью, имеется инструмент *dart2js* (подробнее мы рассмотрим его ниже), который принимает на входе Dart-код и порождает эквивалентный ему .js-файл. Программа *dart2js* удаляет излишний код, поэтому вы можете импортировать сколько угодно внешних Dart-библиотек в уверенности, что в JavaScript перейдет только тот код, который действительно используется в приложении. Для вызова *dart2js* служит команда меню редактора **Tools** ⇒ **Generate JavaScript** (Инструменты ⇒ Генерировать JavaScript).

В следующем листинге показан полный результат трансляции приложения «Здравствуй, мир» с Dart на JavaScript. В него включены код запуска изолятора (единицы исполнения в Dart) и код, эквивалентный функции `print()` из базовой библиотеки Dart. Инструмент *dart2js* создает также файл соответствия исходного кода, которым современные браузеры могут воспользоваться, чтобы показать исходный Dart-код. Поэтому, хотя вы и можете ознакомиться с результирующим JavaScript-кодом, острой необходимости в этом нет.

Листинг 2.3. HelloWorld.dart.js: результат работы dart2js

```
// Generated by dart2js, the Dart to JavaScript compiler.
// The code supports the following hooks:
// dartPrint(message) - if this function is defined it is called
// instead of the Dart [print] method.
// dartMainRunner(main) - if this function is defined, the Dart [main]
// method will not be invoked directly.
// Instead, a closure that will invoke [main] is
// passed to [dartMainRunner].
function Isolate() {}
init();

var $$ = {};
var $ = Isolate.$isolateProperties;

$.Primitives_printString = function(string) {
  if (typeof dartPrint == "function") {
    dartPrint(string);
    return;
  }
  if (typeof console == "object") {
    console.log(string);
    return;
  }
  if (typeof write == "function") {
    write(string);
    write("\n");
  }
};

$.print = function(obj) {
  $.Primitives_printString(obj);
};

$.main = function() {
  return $.print('Hello World');
};

var $ = null;
Isolate = Isolate.$finishIsolateConstructor(Isolate);
var $ = new Isolate();

//
// BEGIN invoke [main].
```

Функция `main()`,
из которой
вызывается `print`

```
//
if (typeof document != 'undefined' && document.readyState != 'complete') {
  document.addEventListener('readystatechange', function () {
    if (document.readyState == 'complete') {
      if (typeof dartMainRunner == 'function') {
        dartMainRunner(function() { $.main(); });
      } else {
        $.main();
      }
    }
  }, false);
} else {
  if (typeof dartMainRunner == 'function') {
    dartMainRunner(function() { $.main(); });
  } else {
    $.main();
  }
}
//
// END invoke [main].
//

function init() {
  Isolate.$isolateProperties = {};
  Isolate.$finishIsolateConstructor = function(oldIsolate) {
    var isolateProperties = oldIsolate.$isolateProperties;
    var isolatePrototype = oldIsolate.prototype;
    var str = "{\n";
    str += "var properties = Isolate.$isolateProperties;\n";
    for (var staticName in isolateProperties) {
      if (Object.prototype.hasOwnProperty.call(isolateProperties, staticName))
      {
        str += "this." + staticName + " = properties." + staticName + ";\n";
      }
    }
    str += "}\n";
    var newIsolate = new Function(str);
    newIsolate.prototype = isolatePrototype;
    isolatePrototype.constructor = newIsolate;
    newIsolate.$isolateProperties = isolateProperties;
    return newIsolate;
  };
}
```

Ассоциированный файл соответствия

//@ sourceMappingURL=HelloWorld.dart.js.map

← исходного кода

Теперь, имея код приложения на языках Dart и JavaScript (с единственным источником – кодом на Dart), мы можем переключать браузер с одного варианта на другой. Если браузер понимает Dart, он загрузит файл с расширением .dart, иначе файл с расширением .js.

Примечание. В отличие от основанного на Java продукта Google Web Toolkit (GWT), который формирует разные варианты JavaScript-файлов для каждого поддерживаемого браузера, dart2js порождает только один файл на JavaScript. Dart спроектирован так, чтобы его можно было транслировать в JavaScript-код, единый для всех современных браузеров.

2.2.5. Генерация документации с помощью dartdoc

В состав инструментария, поставляемого вместе с редактором, входит также программа dartdoc для генерации документации по API. Если добавить специальный комментарий в функцию main(), как показано ниже, то будет сгенерирована документация по API этой функции:

```
/// Это точка входа, она выводит строку
/// Hello World на консоль
main() {
    print("Hello World");
}
```

Документация формируется в том же формате, что документация по API Dart, размещенная на сайте <http://api.dartlang.org>. Сгенерированная документация для приложения «Здравствуй, мир» показана на рис. 2.8.

2.2.6. Отладка Dart-кода с помощью точек останова

Работая совместно, браузер Dartium и редактор Dart обеспечивают функционирование точек останова и пошаговую отладку. Если поставить в программе точку останова и запустить приложение в Dartium, то образуется канал двусторонней связи между редактором и Dartium, который дает возможность редактору прерывать выполнение в точке останова. Тем самым мы получаем ту же функциональность, что в средах разработки на Java или C#. В редакторе

Dart можно определять контрольные переменные, перемещаться по стеку вызовов, производить шаг с обходом или заходом в метод или функцию.

На рис. 2.9 показано, как это выглядит на практике: слева – исходный код, а справа – отладчик со стеком вызовов и локальными переменными.

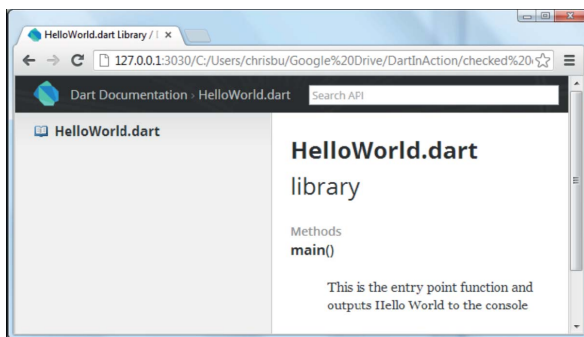


Рис. 2.8. Документация, сгенерированная программой dartdoc для приложения «Здравствуй, мир»

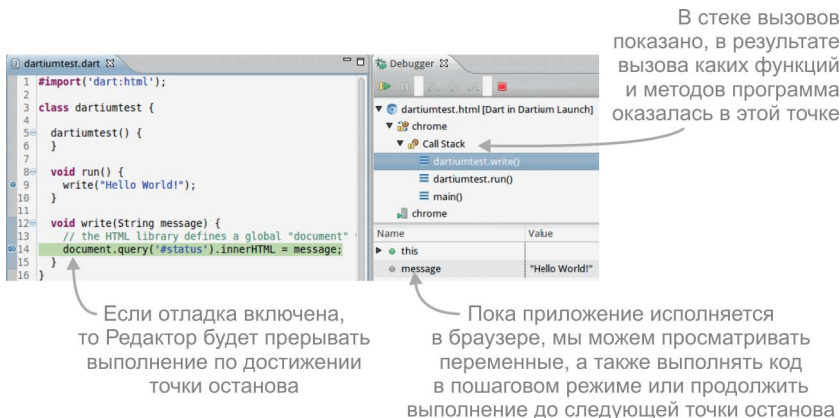


Рис. 2.9. Задание точек останова и отладка в редакторе Dart

Памятка

- Вместе с редактором Dart поставляются Dart SDK и браузер Dartium.
- В редактор встроены различные инструменты навигации и редактирования, в том числе рефакторинг кода и окна для просмотра мест вызова и структуры кода.

- Для статического анализа кода служит инструмент `dart_analyzer`. Он выводит в редактор сообщения об ошибках и предупреждения.
- Инструмент `dartdoc` генерирует документацию по API в формате HTML из включенных в код комментариев.
- Редактор Dart и Dartium взаимодействуют, обеспечивая сквозную интегрированную отладку.

«Живая» отладка веб-приложения – исключительно полезное средство. Возможность расставлять точки останова прямо в редакторе Dart (где можно быстро внести и сохранить изменения, а затем обновить страницу) – отличный способ довести приложение до совершенства.

Итак, мы узнали, как редактировать и запускать приложение в редакторе Dart. Самое время воспользоваться Dart для организации пользовательского интерфейса.

2.3. Импорт библиотек для работы с пользовательским интерфейсом в браузере

В этом разделе мы подробнее рассмотрим, как с помощью библиотеки `dart:html` организуется взаимодействие Dart с моделью DOM браузера. Применяя комбинацию редактора Dart и Dartium для быстрого обновления кода, мы модифицируем HTML-файл, добавив в него один элемент `<div>` с идентификатором `"status"`. Затем мы обратимся к этому `<div>`'у из скрипта «Здравствуй, мир», заменив его содержимое строкой `"Hello World"`. В листинге ниже показан добавленный `<div>` `status`.

Листинг 2.4. `HelloWorld.html`, в который добавлен `<div>` `status`

```
<!DOCTYPE html>
<html>
  <body>
    <div id="status">Waiting for Dart...</div>
    <script type="application/dart" src="HelloWorld.dart"></script>
    <script src="http://dart.googlecode.com/
    svn/branches/bleeding_edge/dart/client/dart.js"></script>
  </body>
</html>
```

Содержимое этого элемента `<div>`
← будет изменено в Dart-приложении

Имея на HTML-странице элемент, мы можем модифицировать Dart-скрипт, так чтобы он обновлял содержимое этого элемента.

Вместо `print` мы включим предложение, которое будет искать элемент `<div>` на странице. Для этого мы воспользуемся встроенной библиотекой `dart:html`, предоставляющей все API для работы с HTML, которые принято ожидать от модели DOM, только в виде, согласованном со стилем Dart; мы часто будем встречаться с библиотекой `dart:html` на страницах этой книги. Если вы привыкли к JavaScript-библиотеке `jQuery`, то `dart:html` покажется вам очень знакомой.

2.3.1. Импорт библиотек Dart

Для импорта библиотеки служит предложение `import`. У него есть три разновидности, которые мы подробнее рассмотрим в главе 5. Первая предназначена для импорта базовых библиотек, в частности `dart:html`. В этом случае употребляется префикс `dart:`, например:

```
import "dart:html";
```

Поскольку эта библиотека является частью Dart SDK, инструменты знают, где ее найти.

Вторая разновидность предложения `import` применяется для импорта сторонних библиотек. В ней используется префикс `package:`, например:

```
import "package:unittest/unittest.dart";
```

Библиотеки, импортированные с префиксом `package:`, разрешаются менеджером пакетов `pub`, который доступен как из командной строки, так и из редактора Dart.

`Pub` может скачивать библиотеки со всеми зависимостями из репозитория, расположенных в Интернете, например с сайта `GitHub` или с собственного репозитория Dart по адресу `pub.dartlang.org`. При таком использовании этот инструмент выполняет те же функции, что `Maven` для Java, `NuGet` для .NET и `npm` для `node.js`. Наиболее употребительны следующие команды:

```
pub install  
pub update
```

Первая команда устанавливает нужные зависимости, вторая обновляет их. Инструмент `pub` появился недавно и, вероятно, в своем развитии обретет дополнительные функции, к числу которых относится генерация заготовок для стереотипных сайтов и каркасов.

Последняя разновидность предложения `import` предназначена для импорта написанных вами или другими лицами библиотек из указанного каталога на локальном диске, например:

```
import "../myLibraries/helloLibrary.dart";
```

В этом случае подразумевается URI папки с библиотеками Dart, которая должна быть доступна инструментам Dart. При конвертации Dart в JavaScript все импортированные библиотеки собираются в единый JavaScript-файл, содержащий только код, который реально используется.

2.3.2. Доступ к элементам DOM с помощью `dart:html`

На рис. 2.10 показан наш Dart-код, в который добавлено предложение `import "dart:html";`, а функция `main()` модифицирована так, чтобы записывать в `<div>` с идентификатором `status` текст "Hello World". Применяемая для этой цели функция `query()` входит в библиотеку `dart:html`.

Функция `query()` аналогична функции `$()` из библиотеки `jQuery`, которая опрашивает модель DOM с применением CSS-селекторов.

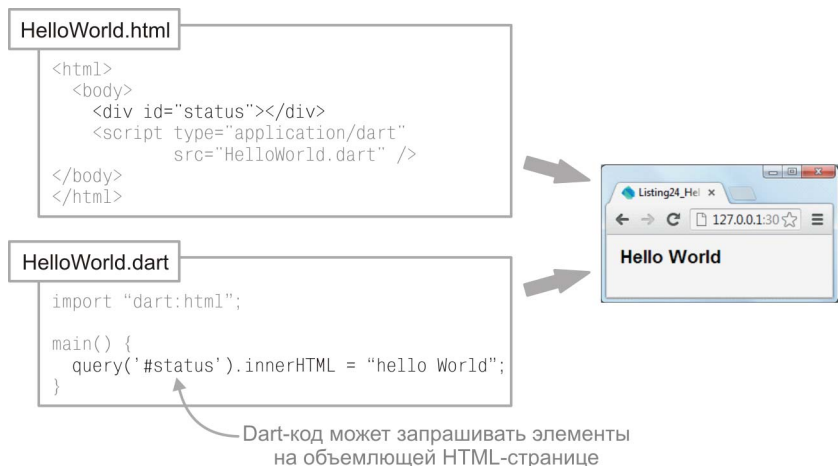


Рис. 2.10. Обновление содержимого элемента `<div>` в браузере

тельных элементов, как кнопки, — с помощью конкретного класса `ButtonElement`. Мы еще вернемся к созданию элементов в следующей главе и — на гораздо более детальном уровне — в третьей части книги, когда будем говорить о создании веб-приложений, работающих на стороне клиента.

В листинге ниже показано модифицированное приложение «Здравствуй, мир», которое конструирует пользовательский интерфейс.

Листинг 2.5. `HelloWorld.dart`: динамическое создание элементов

```
import "dart:html";

main() {
    query("#status").innerHTML = "Hello World"; ← существующий элемент #status

    var button = new ButtonElement(); ← Создается новый элемент button
    button.text = "Click me"; ← Устанавливается свойство кнопки text
    button.onClick.add( (e) {
        var div = new Element.html("<div>I am a div element</div>");
        document.body.elements.add(div);
    });
    document.body.elements.add(button); ← Динамическое добавление кнопки на страницу
}
```

Модифицируется

Обработчик события "on click" создает элементы div из строки с HTML-разметкой

В этом примере мы динамически создали несколько элементов и присоединили обработчик события к кнопке. Функция обработки события — пример анонимной функции, подробнее этот вопрос рассматривается в главе 4.

Памятка

- Библиотека `dart:html` позволяет получить доступ к модели DOM браузера и к отдельным элементам на странице.
- Приложение может динамически создавать элементы и включать их в DOM.
- Менеджер пакетов `pub` позволяет импортировать внешние библиотеки.
- Импортировать библиотеки можно и непосредственно из локальной файловой системы.

2.4. Резюме

В этой главе мы рассмотрели различные инструменты, доступные разработчику, пишущему на языке Dart. Многие языки ставят перед

собой те же цели, что и Dart, но далеко не все располагают столь же богатым набором инструментов.

- ❑ Редактор Dart, браузер Dartium и виртуальная машина Dart предоставляют все необходимое для написания, исполнения и отладки программ на Dart.
- ❑ Программа dart2js конвертирует Dart-код в JavaScript, который может исполняться в современных браузерах.
- ❑ Dart сопровождается библиотеками, которые позволяют удобно взаимодействовать с моделью DOM браузера.

Благодаря сочетанию редактора Dart и Dartium вы можете изучать и экспериментировать с языком Dart и базовыми библиотеками. Поскольку Dart-код можно исполнять непосредственно в браузере Dartium без промежуточного шага компиляции, то продуктивность программиста резко возрастает, так как результат работы можно наблюдать сразу же.

В следующей главе мы приступим к созданию настоящего клиентского Dart-приложения. Мы бегло познакомимся с некоторыми концепциями языка Dart, в том числе с факультативной типизацией и классами, которые в последующих главах изучим гораздо подробнее. Мы также увидим, как воспользоваться каркасом автономного тестирования Dart-программ.



Глава 3. Создание и тестирование Dart-приложения

В этой главе

- ❑ Конструирование пользовательского интерфейса в браузере.
- ❑ Реагирование на пользовательские события.
- ❑ Повторное использование кода с помощью функций.
- ❑ Знакомство с классами в Dart.
- ❑ Написание простых автономных тестов.

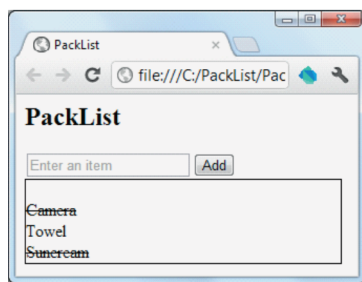
Настало время заняться настоящим программированием на Dart. Пользуясь переменными, функциями и классами, мы напишем простенькое браузерное приложение `PackList`, позволяющее пользователю составить список вещей, которые нужно взять в отпуск. Одна из целей, которые ставились при проектировании Dart, – сделать язык знакомым; в этой главе мы освоимся с переменными, функциями и классами, чтобы в последующих перейти к более интересным вещам.

Пользовательский интерфейс мы будем строить не в виде HTML-разметки, а с помощью производных от `Element` классов из встроенной библиотеки `dart:html`. На момент написания этой книги в составе Dart SDK еще не было библиотеки интерфейсных виджетов, хотя третьими сторонами уже разрабатывались различные библиотеки с открытым исходным кодом. Команда разработчиков Dart намеревается рано или поздно включить библиотеку пользовательского интерфейса в SDK. Ну а пока мы можем сконструировать интерфейс из HTML-элементов прямо в коде. Зная, как это делается, вы легко разберетесь в библиотеке ГИП, когда она появится, потому что уже будете владеть механизмами работы виджетов.

Хотя приложение `PackList` непритязательно и до предела упрощено, реальные одностраничные веб-приложения создаются точно так же. Пользовательский интерфейс у нас будет состоять из поля ввода, кнопки и элемента `<div>`, в котором отображается список вещей. Чтобы приложение могло реагировать на действия пользователя,

мы включим функции обработки событий для добавления новых пунктов в список и пометки упакованных предметов. Наконец, мы напишем класс для представления предмета (и признака его упаковки). При таком дизайне код получится структурированным и допускающим повторное использование.

Программа PackList не нуждается в серверной части – приложение можно запустить в браузере, прочитав файлы прямо с локального диска (хотя можно разместить их и на веб-сервере). PackList работает целиком на стороне клиента; о том, как клиент и сервер могут обмениваться данными, я расскажу в главе 14. Работающее приложение выглядит, как показано на рис. 3.1.



PackList – клиентское приложение, которое позволяет пользователю добавлять в список вещи, которые нужно взять с собой в отпуск, и вычеркивать те, что уже упакованы.

Его можно загрузить с любого веб-сервера в виде статических файлов с расширениями .html и .dart (последние могут быть предварительно конвертированы в JavaScript с помощью программы dart2js), так как никакой обработки на сервере нет

Рис. 3.1. Приложение PackList

К концу этой главы у нас будут готовое одностраничное приложение, работающее в браузере, и набор простых автономных тестов, подтверждающих правильность кода.

3.1. Конструирование пользовательского интерфейса с помощью dart:html

Одностраничное приложение типа PackList конструирует свой пользовательский интерфейс, программно создавая элементы HTML и манипулируя ими. Преимущество такого подхода – в том, что логика отображения пользовательского интерфейса (например, решения о структуре интерфейса в зависимости от состояния пользовательских данных) сосредоточена в браузере, так что ресурсы сервера освобождаются, и он может обслужить больше запросов. В случае одностраничного приложения сервер сначала отправляет

браузеру Dart-код в виде набора статических файлов, а после того как приложение начнет работать, посылает ему данные.

Мы построим приложение `PackList`, отталкиваясь от начального HTML-файла, содержащего Dart-скрипт, и файла с Dart-кодом, который создает HTML-элементы и вставляет их в документ, отображаемый браузером.

3.1.1. Начальный HTML-файл

Любое браузерное Dart-приложение должно быть погружено в HTML-файл, который содержит тег `<script>`, ссылающийся на файл с расширением `.dart`, где находится точка входа в приложение — функция `main()`, которая вызывается автоматически, когда Dart-код будет полностью загружен и готов к работе. Минимальный HTML-файл с тегом `<script>`, который ссылается на Dart-код, выглядит следующим образом:

```
<html>
  <body>
    <script type="application/dart" src="packList.dart"></script> ← Тег <script>
    </body>                                                       загружает .dart-файл
  </html>                                                         и вызывает функцию main()
```

Редактор Dart может создать оба файла автоматически, если запущен мастер нового приложения **New Application Wizard**, а в нем выбран вариант **Web Application** (Веб-приложение). В сгенерированном HTML-файле имеется также тег `<script>`, который загружает файл `packList.dart.js`, содержащий JavaScript-версию приложения на случай, если браузер не понимает языка Dart.

Совет. *Dart isn't running?* Мастер в редакторе Dart создает образец HTML-файла и образец `.dart`-файла. Перед тем как приступить к написанию приложения, удалите пример класс из `.dart`-файла и теги `<h1>` и `<h2>` из `.html`-файла, в противном случае при попытке выполнить приложение в браузере вы увидите сообщение «Dart is not running» (Dart не запущен).

Итак, у нас есть начальный HTML-файл, который содержит тег `<body>` и тег `<script>` для загрузки Dart-приложения, и можно приступить к программному заполнению `<body>` HTML-элементами.

3.1.2. Создание элементов с помощью `dart:html`

При конструировании пользовательского интерфейса в Dart используется библиотека `dart:html`, входящая в состав Dart SDK. Эта библиотека предлагает стандартизованный способ взаимодействия с браузером, абстрагируя различные странности модели DOM (по аналогии с популярной JavaScript-библиотекой `jQuery`). С помощью `dart:html` можно создавать HTML-элементы (кнопки, `<div>`'ы и т. д.) и присоединять их к документу, отображаемому браузером. HTML-элементы представлены классами, производными от `Element`: `DivElement`, `ButtonElement`, и прочими. Программа имеет доступ к свойствам и методам объектов, представляющих HTML-элементы. Элемент может содержать дочерние элементы, обладать идентификатором и стилями, к нему можно присоединять прослушиватели событий.

Совет. На самом деле мы работаем с реализацией интерфейса `Element` (а также `DivElement` и `ButtonElement`), но для программиста это прозрачно – мы получаем экземпляр чего-то, что «выглядит» как `Element`. Подробнее об интерфейсах и их отношениях с классами я буду говорить в главе 6.

В Dart создать HTML-элемент можно двумя способами. В первом случае создается пустой тег, например `<div></div>`, который затем заполняется программно. Во втором случае строится готовый фрагмент разметки, например:

```
<h2 id="title">Pack<em>List</em></h2>
```

Чтобы создать новый объект `Element`, нужно вызвать конструктор класса `Element` из библиотеки `dart:html` с помощью ключевого слова `new`. Созданный объект запоминается в переменной и затем присоединяется к телу документа. Мы можем создать элемент с пустым тегом:

```
var myElement = new Element.tag('div'); ← Имя HTML-тега
```

Или элемент с дочерними элементами и свойствами:

```
var myElement = new Element.html(  
    'h2 id="title">Pack<em>List</em></h2>'); ← HTML-тег
```

У каждого подхода есть свои плюсы, мы рассмотрим их ниже, когда начнем конструировать интерфейс приложения.

В приложении PackList будет заголовок внутри тега `<h2>`, поле для ввода текста, кнопка и элемент `<div>`, где отображается список вещей, которые нужно взять в отпуск. Все эти элементы создаются программно и добавляются в тело документа; структура получающегося в результате HTML-документа показана на рис. 3.2.

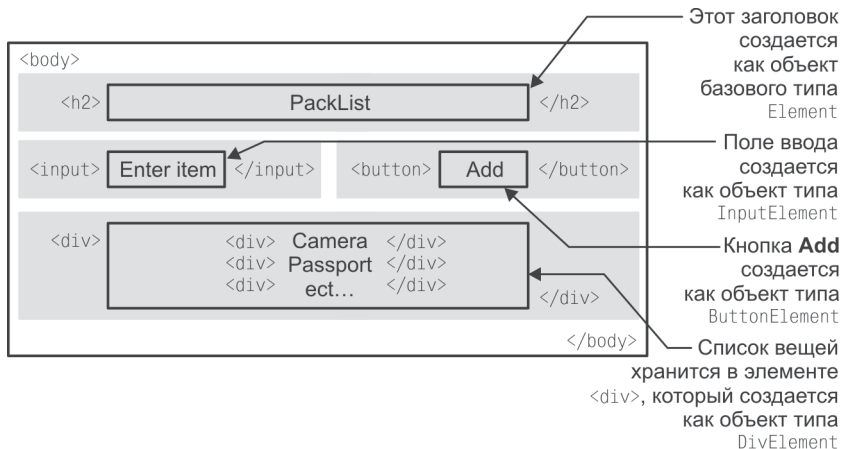


Рис. 3.2. Структура пользовательского интерфейса приложения PackList

3.1.3. Создание экземпляра *Element* из фрагмента HTML

Создавать элементы с помощью конструктора `new Element.html("...html-разметка...")` имеет смысл, когда выполняется одно из следующих условий:

- ❑ заранее известно, как будет выглядеть элемент и его потомки;
- ❑ не требуется сохранять ссылки на дочерние элементы в переменных Dart (тем не менее впоследствии до них можно будет добраться с помощью запроса).

Конструктору `Element.html` можно передать произвольную строку с HTML-разметкой, при условии что результатом является ровно один элемент верхнего уровня.

Недопустимо: содержит два элемента `<p>`
`"<p>Some html</p><p>Another line</p>"` ← верхнего уровня

Допустимо, так как оба элемента `<p>` находятся внутри одного элемента `<div>`
`"<div><p>Some html</p><p>Another line</p></div>"` ←

Объявление строк в Dart

Создать строку в Dart можно несколькими способами. Можно объявить многострочную строку, поставив в начале и в конце три знака кавычки:

```
var myString = """<div>
    <p>a multiline string</p>
</div>""";
```

Такая строка будет отформатирована следующим образом:

```
<div>
    <p>a multiline string</p>
</div>
```

Многострочная строка хранится в точности так, как определена, – со всеми пробелами и знаками перевода строки, иногда это имеет негативные последствия. Например, строка

```
var myString = """<div>
    <p>a multiline string</p>
</div>""";
```

выводится в виде (скорее всего, вы этого не хотели):

```
<div>
    <p>a multiline string</p>
</div>
```

Впрочем, с точки зрения форматирования HTML, не имеет значения, занимает строка одну или несколько строчек, да и читабельность кода не стоит на первом месте. Dart автоматически конкатенирует соседние строковые литералы (то есть две строки, расположенные одна за другой, пусть даже в разных строчках, конкатенируются). Таким образом, в следующих двух переменных хранятся идентичные строки (не многострочные):

```
var myString = "<div>" "<p>a string</p>" "</div>";
var myString = "<div>"
    "<p>a string</p>"
    "</div>";
```

Второе объявление `myString` более наглядно, но на результирующем HTML-коде это не отражается.

Идеальная ситуация для использования конструктора `Element.html` выглядит примерно так:

```
var paragraphContent = "Some about box text";    Создается многострочная строка
Element infoBoxDiv = new Element.html(""  
  <div id='infoBox'>  
    <h3>About PackList</h3>      Значение переменной paragraphContent  
    <p>${paragraphContent}</p>   ← подставляется в HTML-разметку благодаря префиксу $  
  </div>""");
```

Я назвал эту ситуацию идеальной, потому что нам не нужны ссылки на дочерние элементы `<div>`'а `aboutBox` и сам текст относительно статичен. Значение переменной `paragraphContent` подставляется в многострочную строку.

Подстановка переменных в строку

В Dart имеется простой синтаксис для подстановки переменной в строку: `$variableName` или `${expression}`. Это позволяет объявить строку следующим образом:

```
var myValue = 1234;  
var myString = "<p>${myValue}</p>";  
var myOtherString = "<p>${myValue + 1}</p>";
```

Здесь значение переменной `myString` будет `"<p>1234</p>"`, а переменной `myOtherString` — `"<p>1235</p>"`.

Если элемент создан с помощью конструктора `Element.html`, то обращаться ко всем свойствам и методам элемента, сохраненного в переменной, по-прежнему можно, например:

```
infoBoxDiv.children.add(new Element.html("<p>a second paragraph</p>"));    Добавляется еще один элемент в infoBoxDiv  
var id = infoBoxDiv.id;    ← Читается значение <div id='infoBox'>
```

Второй способ порождения HTML-элементов дает конструктор `Element.tag()`, создающий пустой элемент, который можно заполнить из программы.

3.1.4. Создание элементов по имени тега

Иногда требуется задать все свойства динамически на этапе выполнения. В этом случае можно создать пустой элемент, соответствующий указанному тегу, и затем производить с ним различные манипуляции:

```
var itemInput = new Element.tag("input");
```

Здесь создается элемент `InputElement`, которому соответствует такая HTML-разметка:

```
<input></input>
```

Далее можно задать свойства этого элемента, обращаясь к полям объекта `itemInput`:

```
itemInput.id = "txt-item";  
itemInput.placeholder = "Enter an item";
```

Результирующий HTML-код выглядит так:

```
<input id="txt-item" placeholder="Enter an Item"></input>
```

Совет. В свойстве `placeholder` задается текст, который отображается в поле ввода светло-серым цветом и исчезает, как только пользователь начинает вводить в это поле данные.

Какой бы конструктор ни использовать — `Element.tag()` или `Element.html()`, — возвращается объект, который выглядит как `Element`. Иногда, впрочем, требуется обращаться к дополнительным свойствам, специфичным для конкретных типов элементов, например `InputElement` или `ButtonElement`. Благодаря факультативной типизации в Dart исполняемой программе не важно, задан фактический тип элемента или нет, но все же объявлять конкретный тип элемента полезно, чтобы интерпретатор мог проверить тип и выдать предупреждение, если элемент используется недопустимым способом. Например, для создания поля ввода годится любое из следующих трех предложений:

```
var itemInput = new Element.tag("input");  
Element itemInput = new Element.tag("input");  
InputElement itemInput = new Element.tag("input");
```

Но вариант в третьей строке, где явно сказано, что переменная `itemInput` имеет тип `InputElement` (а не, скажем, `ButtonElement` или `DivElement`), сообщает инструментам (и людям, читающим код), что вы действительно имели в виду тип `InputElement`. Ко всему прочему редактор Dart предлагает в списке вариантов автозавершения кода свойства и методы класса `InputElement`.

Для любого элемента, определенного в спецификации HTML, имеется эквивалентный тип в библиотеке `dart:html`. Это относится и к элементам, появившимся только в HTML5, например `CanvasElement`.

Совет. Вместе с Dart поставляется также библиотека `dart:dom`. Она содержит средства прямого доступа к модели DOM браузера и работает так же, как аналогичные средства манипулирования DOM в JavaScript, но ценой утраты единообразного интерфейса `Element`.

Научившись создавать элементы с помощью конструкторов `Element.tag()` и `Element.html()`, посмотрим, как они добавляются в HTML-документ для отображения в браузере.

3.1.5. Добавление элементов в HTML-документ

Видимое содержимое HTML-документа находится внутри тега `<body>`. Именно туда добавляются элементы, созданные средствами библиотеки `dart:html`. На верхнем уровне этой библиотеки определено свойство `document`, у которого, в свою очередь, есть свойство `body`. Программа обращается к свойству `document.body` следующим образом:

```
document.body.children.add(...какой-то элемент...)
```

Совет. У документа верхнего уровня имеется также свойство `document.head`, которое полезно, когда нужно присоединить такие элементы, как `meta` и `keywords`, или название документа к заголовку HTML-страницы.

Объект `document.body` сам по себе является `Element`-ом, а у любого объекта `Element` имеется свойство `children`. Это позволяет создавать иерархии элементов, связанные отношением «родитель–потомок», как показано на рис. 3.3.

В приложении `PackList` мы хотим добавить заглавие внутри тега `<h2>` и для этой цели воспользуемся конструктором `Element.html()`. Кроме того, создадим поле ввода и кнопку – и то, и другое с помощью конструктора `Element.tag()`. В листинге ниже показан соответствующий код, помещенный в функцию `main()`, которая автоматически вызывается в начале работы приложения.

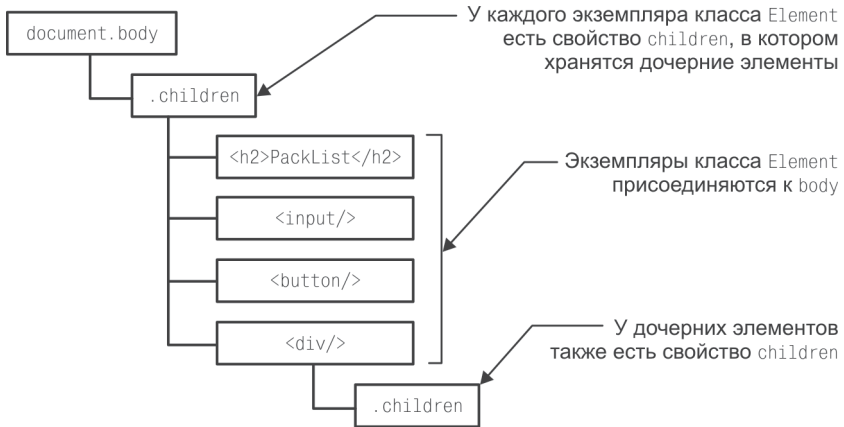


Рис. 3.3. Элемент `body` может содержать много дочерних элементов

Листинг 3.1. Добавление объекта `Element` в тело документа

```
import "dart:html";    ← Импортируется библиотека dart:html

main() {
    var title = new Element.html("<h2>PackList</h2>");    ← элемент, содержащий заглавие
    document.body.children.add(title);    ← Заглавие включается в тело документа

    InputElement itemInput = new Element.tag("input");    | Создается и добавляется
    document.body.children.add(itemInput);                | поле ввода

    ButtonElement addButton = new Element.tag("button");    | Создается и добавляется
    document.body.children.add(addButton);                  | кнопка Add

    DivElement itemContainer = new Element.tag("div");
    document.body.children.add(itemContainer);
}
```

В результате будет сформирована такая HTML-разметка:

```
<body>
  <h2>PackList</h2>
  <input></input>
  <button></button>
</body>
```

Помимо добавления элементов в тело документа, мы должны еще задать некоторые их свойства, например:

```
inputButton.placeholder = "Enter an item";
addButton.text = "Add";
addButton.id = "add-btn";
```

Несущественно, делается это до или после включения элемента в тело, — браузер в любом случае выполнит необходимые обновления страницы, чтобы отразить новое состояние элементов.

Наконец, для завершения пользовательского интерфейса мы добавим элемент `<div>`, в котором будет отображаться список предметов. С помощью свойств объекта `element.style` (см. листинг 3.2) мы можем применить стили непосредственно к элементу `<div>` (на практике используйте для форматирования CSS).

После запуска это приложение порождает пользовательский интерфейс, показанный на рис. 3.4. В листинге ниже приведен полный код, в котором мы создаем все четыре элемента и настраиваем их свойства.

Листинг 3.2. Конструирование пользовательского интерфейса приложения PackList

```
import "dart:html";

main() {
  var title = new Element.html("<h2>PackList</h2>");
  document.body.children.add(title);

 InputElement itemInput = new Element.tag("input");
  itemInput.id = "txt-item";
  itemInput.placeholder = "Enter an item";
  document.body.children.add(itemInput);

  ButtonElement addButton = new Element.tag("button");
  addButton.id = "btn-add";
  addButton.text = "Add";
  document.body.children.add(addButton);

  DivElement itemContainer = new Element.tag("div");
  itemContainer.id = "items";
  itemContainer.style.width = "300px";
  itemContainer.style.border = "1px solid black";
  itemContainer.innerHTML = "&nbsp;";
  document.body.children.add(itemContainer);
}
```

Устанавливаются свойства
InputElement

Устанавливаются свойства
ButtonElement

Устанавливаются
свойства
DivElement

← Содержимым `<div>` будет один
неразрывный пробел (` `)

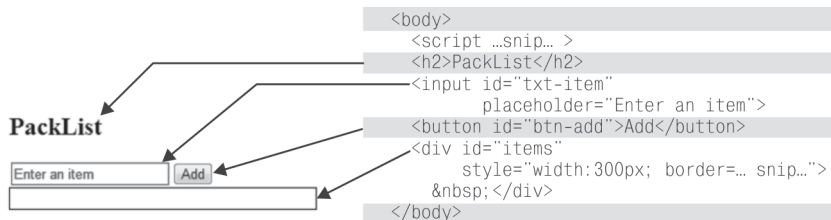


Рис. 3.4. Пользовательский интерфейс приложения PackList

Ради демонстрации мы сделали этот код несколько искусственным. С помощью конструктора `Element.html` можно было бы создать объект `itemContainer` типа `DivElement` короче:

```
DivElement itemContainer = new Element.html('<div id="items"
↪ style="width:300px;border:1px solid black">&nbsp;</div>');
```

Итак, у нас теперь есть пользовательский интерфейс, сконструированный программно из объектов, находящихся в библиотеке `dart:html`. Из таких блоков строится любой пользовательский интерфейс на языке Dart.

Памятка

- Объекты типа `Element` можно создавать двумя способами: `Element.html(...разметка...)` или `Element.tag(...имя тега...)`.
 - В библиотеке `dart:html` определены все элементы, которые понимают современные браузеры.
 - Редактор Dart может предоставить список вариантов автозавершения для свойств (а в документации по API на сайте api.dartlang.org приведена более подробная информация).
 - Элементы становятся в браузере HTML-тегами, а их свойства – атрибутами тегов.
 - У любого элемента (в том числе `body`) имеется свойство `children`, значением которого является список дочерних элементов.
-

Сейчас наше приложение строит только статический пользовательский интерфейс. Оно не реагирует на действия пользователя и не изменяет интерфейс в ответ. Чтобы обработать нажатие кнопки **Add** и добавить новый предмет в список, нам нужны функции-прослушиватели событий. Именно этим мы и займемся в следующем разделе.

3.2. Добавление интерактивности с помощью событий браузера

Чтобы пользовательский интерфейс мог реагировать на действия пользователя, например нажатие кнопки, необходимы прослушиватели событий. В библиотеке `dart:html` в роли прослушивателя выступает функция, принимающая единственный параметр `event`, тип которого реализует интерфейс `Event`. Ниже приведен пример прослушивателя:

```
myEventListenerFunction(Event event) {  
  window.alert("Глянь-ка - событие произошло");  
};
```

Параметр `event` предоставляет дополнительную информацию о событии. Как и `Element` и `ButtonElement`, тип `Event` является универсальным. Конкретные типы событий зависят от сгенерировавшего их элемента. Например, в случае `MouseEvent` параметр `event` содержит флаг, показывающий, какая кнопка мыши была нажата — левая или правая. Но приложению `PackList` знать это необязательно, ему нужно лишь получить уведомление о нажатии кнопки **Add**, поэтому универсального объекта `Event` будет достаточно.

3.2.1. Добавление предмета в список по нажатию кнопки

В приложении `PackList` имеются поле ввода, кнопка и элемент `<div>`, где отображаются предметы, которые нужно взять с собой. Сценарий простой: когда пользователь нажимает кнопку `addButton`, мы берем текст из поля ввода и добавляем его в `itemContainer`. После этого поле `inputBox` следует очистить. Эта последовательность действий изображена на рис. 3.5.

Сделать это можно в прослушивателе нажатия кнопки `addButton`, написав код, который создает новый `<div>` (с CSS-классом `item`), содержащий `packItem`, и добавляет его в контейнер предметов. Нас интересуют две вещи:

- ☐ куда поместить прослушиватель нажатия кнопки `addButton`;
- ☐ сам код прослушивателя.

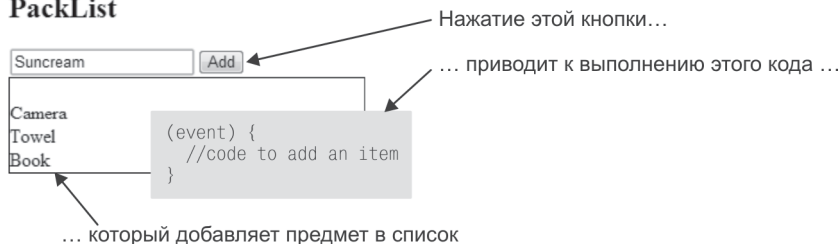
PackList

Рис. 3.5. Приложение PackList позволяет добавлять предметы в список

```
main() {
  // ... код создания элемента ГИП опущен ...
  addButton.onClick.add((event) {
    var packItem = itemInput.value;
    var listElement =
    ↪   new Element.html("<div class='item'>${packItem}</div>");
    itemContainer.children.add(listElement);
    itemInput.value= "";
  });
}
```

Этот код следует поместить в конец функции `main()`. Всякий раз как пользователь нажимает кнопку **Add**, приложение будет добавлять новый предмет в список.

В этом фрагменте много новых синтаксических конструкций, поэтому ненадолго задержимся и посмотрим, что здесь происходит. Сначала мы поговорим о различных способах создания функций в Dart и об определении функции, которая прослушивает события нажатия, а затем рассмотрим, как можно воспользоваться такого рода прослушивателями.

3.2.2. Применение гибкого синтаксиса функций в Dart для обработки событий

Отчасти вследствие наличия факультативной типизации, а отчасти ради синтаксической глазури, позволяющей писать более лаконичные и читабельные программы, в Dart есть несколько способов определения функций. Вы должны научиться распознавать их с первого взгляда.

Примечание. Синтаксической глазурию называется средство языка, позволяющее более изящно выразить некую идею.

Для объявления функций верхнего уровня в Dart существует длинная форма, аналогичная встречающейся в JavaScript, Java или C#, и короткая – с оператором `=>` (лямбда), который может выполнить ровно одно предложение и вернуть его результат.

Длинная форма выглядит так:

```
void myEventListener(Event event) {  
    // что-то сделать  
    window.alert("событие");  
}
```

А коротка (однострочная) – так:

```
void myEventListener(Event event) => window.alert("событие");
```

Как обычно в Dart, типы возвращаемого значения и параметров можно не указывать, на результат работы приложения это не влияет. Необязательно даже указывать в качестве возвращаемого типа `void`.

Синтаксис однострочной функции полезен, когда нужно установить значение чего-то или вызвать другую функцию. Следует отметить, что однострочная функция неявно возвращает значение вычисленного выражения, тогда как в случае длинной записи значение нужно возвращать явно с помощью предложения `return`. Следующие две функции эквивалентны:

```
int addValues(int a, int b) {  
    return a+b;  
}
```

оператор `=>` неявно возвращает значение выражения. Если выражение не порождает значения, то возвращается `null`

```
int addShorthandValues(int a, int b) => a+b;
```

Важно также отметить, что хотя вся информация о типах (возвращаемого значения и параметров) факультативна, сами параметры должны быть поименованы (в отличие от JavaScript, где все параметры необязательны). Кроме того, в отличие от JavaScript, не существует ключевого слова `function` – даже имя функции необязательно. Анонимную (не имеющую имени) функцию можно сохранить

в переменной или передать другой функции в качестве параметра; именно так мы и поступаем в случае обработчика события.

В приложении `PackList` метод в строке `addButton.onClick.add( )` принимает функцию в качестве параметра, и эту функцию можно передать обработчику события, не присваивая ей никакого имени, например:

```
addButton.onClick.add( (event) {  
    // тело функции  
});
```

или

```
addButton.onClick.add( (event) => ...одно предложение... );
```

Можно было бы сначала сохранить функцию в переменной:

```
var myEventListener = (event) {  
    // тело функции  
};  
addButton.onClick.add(myEventListener);
```

или

```
var myEventListener = (event) => ...одно предложение... ;  
addButton.onClick.add(myEventListener);
```

В Dart функции используются для разных целей, в частности для оформления повторно используемого кода, и мы еще будем иметь случай остановиться на них более подробно.

Чтобы помочь вам в идентификации различных вариантов записи функций, на рис. 3.6 приведены эквивалентные функции с различным уровнем полноты информации о типах, в длинной и короткой формах. Результат их работы одинаков (возвращается сумма $a + b$).

Памятка

- У функций есть длинная и короткая синтаксические формы записи.
 - Типы возвращаемого значения и параметров функции факультативны.
 - Анонимные функции можно передавать в качестве параметров и сохранять в переменных.
-

Длинная (многострочная) синтаксическая форма

```
int someFunction(int a, int b) {
  return a+b;
}
```

```
someFunction(int a, int b) {
  return a+b;
}
```

```
someFunction(a, b) {
  return a+b;
}
```

```
var someFunction = (a, b) {
  return a+b;
}
```

```
(a, b) {
  return a+b;
}
```

Короткая (однотрочная) синтаксическая форма

```
int someFunction(int a, int b) => a+b;
```

```
someFunction(int a, int b) => a+b;
```

```
someFunction(a, b) => a+b;
```

```
var someFunction = (a, b) => a+b;
```

```
(a, b) => a+b;
```

Рис. 3.6. Эквивалентные функции в длинной и короткой формах с различным уровнем полноты информации о типах

Научившись создавать функции прослушивания событий, мы можем вернуться к приложению `PackList` и поговорить о некоторых событиях, предоставляемых библиотекой `dart:html`.

3.2.3. Реагирование на события браузера

Библиотека `dart:html` предлагает способ единообразной обработки всех событий, которые браузер может генерировать для элемента. Доступ к перечню событий производится с помощью свойства `element.on`, в котором имеются свойства для всех событий, которые может генерировать элемент `element`. Вот, например, некоторые события, генерируемые кнопкой:

```
addButton.on.click
addButton.on.drag
addButton.on.mouseMove
```

Каждое из этих событий на самом деле представляет собой еще один список (типа `EventListenerList`), содержащий прослушватели. Добавить свой прослушатель в этот список позволяет метод `add()`, которому передается имя функции:

```
addButton.on.click.add(myEventListenerFunction);
```

или сама функция

```
addButton.on.click.add((event) {
  window.alert("Я тоже умею обрабатывать события");
});
```

или сама функция в короткой форме:

```
addButton.on.click.add(event => window.alert("И я!"));
```

Все эти функции выводят диалоговое окно уведомления. Поскольку событие нажатия — это список `EventListenerList`, то никто не запрещает добавить в него все три. При нажатии кнопки они будут вызываться по очереди.

Чтобы удалить функцию `myEventListenerFunction` из списка прослушивателей (у нас есть ссылка на нее по имени), нужно написать:

```
addButton.on.click.remove(myEventListenerFunction);
```

Сейчас в приложении `PackList` для обработки нажатий кнопки используется анонимная функция, но можно возложить на нее дополнительную работу, присвоив ей имя и заставив обрабатывать также события нажатия клавиши. Но для этого придется заняться рефакторингом.

3.2.4. Рефакторинг прослушателя событий для повторного использования

Было бы хорошо, если бы приложение еще умело добавлять предметы в список при нажатии клавиши **Enter**. Добавить прослушатель, который будет реагировать на нажатие **Enter** в текстовом поле, несложно:

```
itemInput.on.keypress.add( (event) {
  if (event.keyCode == 13) {      ← Код клавиши Enter равен 13
    var packItem = itemInput.value;
    var listElement =
    ↪      new Element.html("<div class='item'>${packItem}<div>");
    itemContainer.children.add(listElement);
    itemInput.value= "";
  }
} );
```

Копируем код
из прослушателя
нажатий кнопки

Но мы скопировали код из предложения `if` в обработчике нажатия кнопки `addButton`, а копирование — это плохо. Следует перенести этот код из `main()` в отдельную функцию, которую можно будет использовать повторно:

```
addItem() {
  var packItem = itemInput.value;
  var listElement =
    ↪ new Element.html("<div class='item'>${packItem}<div>");
  itemContainer.children.add(listElement);
  itemInput.value = "";
}
```

Этот код перенесен
из прослушателя
нажатия в функцию
`addItem()`

Теперь можно модифицировать функции прослушивания событий, так чтобы они вызывали новую функцию `addItem()`:

```
addButton.on.click.add( (event) => addItem());  ← кнопки addButton
itemInput.on.keypress.add( (event) {             ← Прослушатель нажатия клавиши в поле itemInput
  if (event.keyCode == 13) {
    addItem();
  }
});
```

Все бы хорошо, только вот функция `addItem()` не работает. Дело в том, что переменные `inputItem` и `itemContainer` объявлены в функции `main()` и в функции `addItem()` не видны.

Чтобы сослаться на эти HTML-элементы, нам понадобится еще одна возможность библиотеки `dart:html` — запрос элементов DOM с помощью CSS-селекторов.

3.2.5. Запрос HTML-элементов в `dart:html`

Мы часто сталкиваемся с ситуацией, когда в программе нет переменной, ссылающейся на какой-то HTML-элемент. Быть может, этот элемент был добавлен конструктором `Element.html()` или вообще создан в другой части приложения. Так или иначе, его надо получить от браузера. По счастью, у элементов из библиотеки `dart:html` есть два полезных метода: `element.query(...)`, который возвращает один элемент, и `element.queryAll(...)`, который возвращает список подходящих дочерних элементов.

В приложении `PackList` нам необходимо получить ссылки на поле ввода `itemInput` (чтобы извлечь из него введенное пользователем

название предмета) и на `<div>` `itemContainer` (чтобы добавить новый предмет в список). Оба этих элемента являются дочерними элементами `body`, как показано на рис. 3.7.

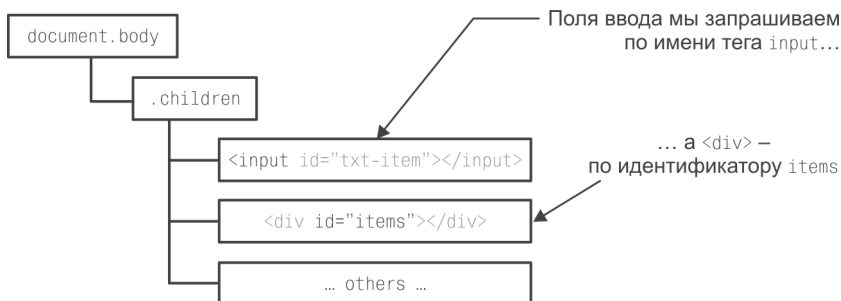


Рис. 3.7. Поле ввода и элемент `<div>` с идентификатором `items` можно получить из тела документа

В примере ниже функция `queryAll()` возвращает все элементы `input`, являющиеся непосредственными потомками `body` (такой элемент всего один), а функция `query()` запрашивает элемент `<div>` с идентификатором `items` (применяя CSS-селектор `#items`).

Чтобы получить ссылки на поле ввода и контейнер предметов, необходимо добавить в функцию `addItem()` следующие строки:

```

addItem() {
  var itemInput = queryAll("input"); ← Запрос всех тегов input, являющихся потомками body
                                     Первый (и единственный) тег input
  InputElement itemInput = itemInputList[0]; ← сохраняется в переменной itemInput

  DivElement itemContainer = query("#items"); ← Запрос элемента с идентификатором
  // ... остальная часть функции опущена ... #items
}

```

В реальной программе мы запрашивали бы оба элемента по `#id`. Знак `#` в CSS-селекторе означает поиск по идентификатору, который никогда не возвращает более одного элемента. Напротив, вызов `itemContainer.queryAll(".item")` возвращает список элементов с CSS-классом `.item` независимо от типа конкретного элемента. В нашем случае это были бы `<div>`'ы, содержащие добавленные пользователем предметы.

В листинге ниже приведен полный исходный код приложения, которое теперь реагирует на события, так что пользователь может добавлять предметы как щелчком мыши, так и нажатием клавиши.

Листинг 3.3. Приложение PackList реагирует на события

```
import 'dart:html';

main() {
  var title = new Element.html("<h2>PackList</h2>");
  document.body.children.add(title);

  InputElement itemInput = new Element.tag("input");
  itemInput.id = "txt-item";
  itemInput.placeholder = "Enter an item";
  itemInput.on.keyPress.add( (event) { ← Функция addItem() вызывается при нажатии
    if (event.keyCode == 13) {          клавиши Enter
      addItem();
    }
  });
  document.body.children.add(itemInput);

  ButtonElement addButton = new Element.tag("button");
  addButton.id = "btn-add";
  addButton.text = "Add";
  addButton.on.click.add((event) => addItem()); ← Функция addItem() вызывается
  document.body.children.add(addButton);        при нажатии кнопки

  DivElement itemContainer = new Element.tag("div");
  itemContainer.id = "items";
  itemContainer.style.width = "300px";
  itemContainer.style.border = "1px solid black";
  itemContainer.innerHTML = "&nbsp;";
  document.body.children.add(itemContainer);
}

addItem() { ← Вынесенная функция addItem()
  var itemInputList = queryAll("input");
  InputElement itemInput = itemInputList[0];
  DivElement itemContainer = query("#items");
  var itemText = itemInput.value;
  var listElement = new Element.html("<div
  ↪      class='item'>${itemText}<div>");
  itemContainer.children.add(listElement);
  itemInput.value = "";
}
```

Итак, теперь у нас есть клиентское приложение, позволяющее пользователю формировать список предметов, которые нужно взять

с собой в отпуск. В нем используется ряд элементов из библиотеки `dart:html` и функция для запроса элементов; эта функция вызывается, когда пользователь щелкает мышью или нажимает клавишу **Enter**. Очередной шаг – дать возможность вычеркивать из списка уже упакованные предметы. В следующем разделе мы напишем класс `PackItem`, который будет инкапсулировать эту функциональность и сделает код более пригодным для повторного использования.

Памятка

- Прослушиватель событий в `dart:html` – это функция, принимающая единственный параметр `event`.
 - Одно событие могут прослушивать несколько прослушивателей.
 - `dart:html` позволяет запрашивать единственный элемент с помощью функции `query()`, которой передается CSS-селектор.
 - Для запроса нескольких элементов применяется функция `queryAll()`.
-

3.3. Инкапсуляция структуры и функциональности с помощью классов

Последнее, что мы добавим в приложение `PackList`, – возможность вычеркивать из списка уже упакованные элементы. У пользователя должна быть возможность изменить состояние предмета в списке – упакован или не упакован – щелчком по нему мышью.

Сделать это проще всего, добавив прослушиватель события щелчка к каждому элементу списка. Когда пользователь щелкает по предмету, функция-прослушиватель добавляет CSS-стиль, в котором определено перечеркивание.

Для этого мы подвергнем приложение рефакторингу, добавив простой класс `PackItem`, который делает следующее:

- ❑ инкапсулирует функциональность предмета в списке, реализуя переключение между состояниями «упакован» и «не упакован»;
- ❑ хранит данные о предмете (название и состояние).

На рис. 3.8 показана новая последовательность событий.

Чтобы поддержать перечеркивание, мы должны добавить CSS-стили в HTML-файл (можно поместить их в начало секции `<body>`):

```
<body>
```

```
  <style type="text/css">
```

```
    .item { cursor:pointer; } ←
```

Для всех предметов в списке курсор имеет форму указателя,

сообщающую пользователю, что по нему можно щелкнуть

```
.packed { text-decoration:line-through; } ← Если для предмета добавлен CSS-класс
</style>                                     .packed, то его название перечеркивается
... И Т. Д. ...
```

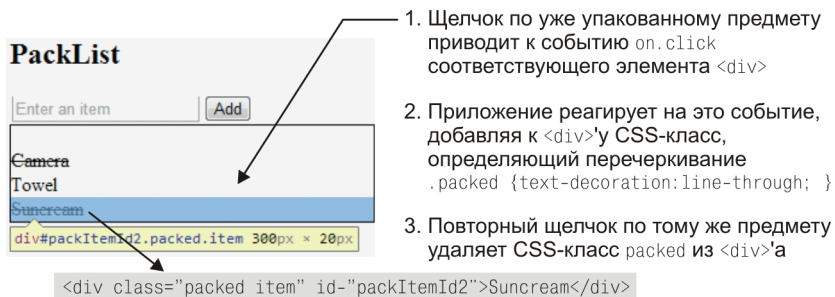


Рис. 3.8. Приложение реагирует на щелчок мышь
по упакованному предмету, перечеркивая его название

Теперь, имея в объемлющем HTML-файле необходимые стили, можно приступить к созданию класса `PackItem`. Поместите следующий код выше или ниже функции `main()` (но не внутри нее):

```
class PackItem {
}
```

3.3.1. Классы в Dart не таят неожиданностей

То, как реализована одна из проектных целей Dart – не вызывать недоумения у программистов, – наглядно проявляется в синтаксисе классов. Если вам доводилось работать с классами в C# или Java, то структура класса в Dart покажется очень знакомой. Разумеется, Dart, оставаясь языком, основанным на классах, с одиночным наследованием классов и множественным наследованием интерфейсов, добавляет кое-какие удобные дополнительные черты.

Одна из них – факультативность классов (в отличие от C# и Java). Функции необязательно должны быть членами класса; например, функции `main()` и `addItem()` определены на верхнем уровне и не являются частью какого-либо класса. Кроме того, в отличие от Java, в одном Dart-файле может быть сколько угодно классов – ограничений на количество нет, но тем не менее для лучшей организации

кода имеет смысл заводить несколько файлов; эта тема обсуждается в главе 5.

Несмотря на все отличия, у классов в Dart есть конструкторы, методы и свойства, которые могут быть открытыми или закрытыми, а также специальный синтаксис для методов чтения и установки свойств. Мы начнем с рассмотрения конструктора, создающего объекты `PackItem`.

3.3.2. Конструирование класса `PackItem`

Основная задача класса `PackItem` — хранить название предмета `itemText`, переданное функцией `addItem()`. Сделать это можно, как показано в листинге 3.4, где определен класс со свойством `itemText` и конструктором, который инициализирует это свойство переданным значением. Нам понадобится также элемент пользовательского интерфейса, ассоциированный с объектом. Обратите внимание на сокращенную нотацию для инициализации свойства `itemText` в конструкторе (конструктор — это функция с тем же именем, что у самого класса).

Листинг 3.4. Базовая структура класса

```
class PackItem {  
  var itemText;  ← Свойство itemText  
  var uiElement; ← Элемент ГИП, ассоциированный с объектом (пока неинициализированный)  
  
  PackItem(this.itemText) {  
    // тело конструктора пусто  
  }  
}
```

Конструктор принимает единственный параметр `itemText` (которым инициализируется свойство `itemText`)

Предупреждение. В общем случае не рекомендуется помещать в один класс поведение пользовательского интерфейса и бизнес-логику, как в этом примере, где данные и элемент ГИП находятся в одном классе. Это усложняет автономное тестирование приложения и является примером тесно связанного дизайна. Было бы лучше завести два класса: один отвечал бы за работу с пользовательским интерфейсом, а другой — за хранение и манипулирование данными объекта. Между этими классами можно было бы организовать слабую связь; о том, как это сделать, я расскажу ниже, при более подробном рассмотрении клиентских приложений в Dart.

Новый экземпляр этого класса создается следующим образом:

```
var somePackItem = new PackItem("Крем от загара");
```

Это то же самое ключевое слово `new`, которым мы пользовались при создании объектов `Element` из библиотеки `dart:html`. Оно вызывает конструктор класса `PackItem`, передавая ему название предмета (взятое из поля ввода).

В Dart учтен опыт использования конструкторов в C# и Java. Один из усвоенных уроков состоит в том, что конструктор чаще всего применяется для инициализации свойств объекта – в данном случае свойства `itemText`. В Dart разрешено записывать конструктор в привычной (программисту на Java) форме:

```
PackItem(itemText) {  
  this.itemText = itemText;  
}
```

Но в Dart имеется и сокращенная нотация, позволяющая указать в списке параметров, что единственное назначение конструктора – инициализация свойства. Такой конструктор записывается следующим образом:

```
PackItem(this.itemText) { }
```

Этот синтаксис удобен сам по себе, но разрешается также смешивать стандартные параметры и инициализаторы:

```
PackItem(this.itemText, color, quantity) {  
  this.color = color;  
  this.quantity = quantity;  
}
```

Теперь, имея простой класс, мы можем переписать метод `addItem()`, как показано в листинге 3.5. Новая версия конструирует объект `PackItem`, передавая ему название предмета, и с помощью свойства `uiElement` добавляет новый предмет в список `itemContainer`. Свойство `uiElement` в конечном итоге должно будет содержать ссылку на `<div>`, но пока оно оставлено неконкретизированным – объявлено с помощью ключевого слова `var`.

Листинг 3.5. Рефакторинг метода `addItem` с целью использования нового класса

```
addItem() {  
  var itemInputList = queryAll("input");
```

```

InputElement itemInput = itemInputList[0];
DivElement itemContainer = query("#items");

var packItem = new PackItem(itemInput.value);
itemContainer.children.add(packItem.uiElement);
itemInput.value = "";
}

```

С помощью конструктора создается
← новый экземпляр PackItem
← Получаем свойство uiElement
объекта и добавляем новый HTML-
элемент в контейнер itemContainer

Хотя рефакторинг произведен корректно, осталась еще одна немаловажная проблема: свойство `uiElement` еще не инициализировано и, следовательно, равно `null`. Это можно исправить, заменив поле методом чтения (дополнительная модификация функции `addItem()` при этом не требуется).

3.3.3. Инкапсуляция функциональности с помощью методов чтения и установки

Поля и методы чтения/установки в Dart взаимозаменяемы, вызывающий код при этом менять не нужно. Это означает, что код

```
itemContainer.children.add(packItem.uiElement);
```

понятия не имеет, получает ли он значение от метода чтения или непосредственно из поля. Имени метода чтения (установки) должно предшествовать ключевое слово `get` (или `set`). Метод чтения не принимает параметров, а метод установки принимает единственный параметр, как показано ниже:

```

class PackItem {
  var _uiElement;    ← Свойство uiElement переименовано в _uiElement
  DivElement get uiElement => _uiElement;  ← В методе чтения применяется короткая
                                              форма для возврата _uiElement
  set uiElement(DivElement value) => _uiElement = value; ← В методе установки также
                                                              применяется короткая
                                                              форма для присваивания
                                                              значения свойству _uiElement
}

```

Вероятно, вы обратили внимание, что мы переименовали свойство `uiElement` в `_uiElement` (добавив в начале подчеркик). У этого изменения есть глубокий смысл, о котором я расскажу в главе 5, когда мы будем говорить о библиотеках и закрытых членах; пока можете считать, что так по соглашению обозначается закрытое свойство.

Совет. Не забывайте, что благодаря факультативной типизации задавать информацию о типе необязательно. Но пользователям вашего класса было бы полезно видеть типы открытых свойств и методов. Информация о типе предназначена для документирования и инструментальных средств.

Использовать наш класс можно следующим образом:

<pre>var packItem = new PackItem("Крем от загара"); packItem.uiElement = new Element.tag("div"); itemContainer.children.add(packItem.uiElement);</pre>	Задает значение <code>uiElement</code> ← с помощью метода установки ← Получает значение <code>uiElement</code> с помощью метода чтения

Поскольку методы чтения и установки взаимозаменяемы с полями, проектировщик библиотеки может начать с простых полей:

```
class PackItem {
  var uiElement;
}
```

а затем, если потребуется расширить функциональность, заменить их методами чтения и установки; менять код вызывающей программы при этом не придется.

Предупреждение. Хороший проектировщик заботится о том, чтобы методы чтения и установки выполнялись быстро. Пользователь должен думать, что просто читает свойство, пусть даже на самом деле в методе чтения или установки производится отложенная инициализация или еще какая-то обработка.

Реализация свойств, допускающих только чтение

Свойство `uiElement` в классе `PackItem` должно допускать только чтение, то есть модифицировать его вправе лишь сам класс. Поскольку функция `addItem()` сейчас читает свойство `uiElement`, то мы можем заменить его методом чтения, не предоставив метода установки (это и означает, что свойство можно читать, но нельзя изменить). Мы воспользуемся методом чтения для «отложенной инициализации» закрытого свойства `_uiElement`, как показано в листинге ниже. При первом вызове метод чтения создает элемент `<div>` и запоминает его в свойстве `_uiElement`. При последующих вызовах он просто возвращает уже готовое значение `_uiElement`.

Листинг 3.6. Добавление метода чтения uiElement

```

class PackItem {
  // остальной код опущен
  var _uiElement; ← Свойство uiElement переименовано в _uiElement

  DivElement get uiElement {
    if (_uiElement == null) { ← Если закрытое свойство _uiElement еще не инициализировано...

      _uiElement = new Element.tag("div"); ← ...инициализировать его элементом <div>

      _uiElement.classes.add("item"); ← Добавить CSS-класс .item, чтобы курсор
                                   принимал форму указателя
      _uiElement.text = this.itemText; ← Установить отображаемый текст
    }

    return _uiElement; ← Вернуть _uiElement вызывающей программе
  }
}

```

Теперь приложение показывает пользователю курсор в форме указателя, подсказывающего пользователю, что по названию предмета в списке можно щелкнуть. Пока что щелчок мышью не изменяет состояния предмета, но мы приближаемся к этому. Следующий наш шаг — сохранить в классе `PackItem` данные о состоянии в виде булевского свойства `isPacked` и добавить прослушиватель события щелчка, который изменит значение `isPacked` на противоположное, как показано на рис. 3.9.

Изменение состояния в методе установки

Для представления свойства `isPacked` мы снова воспользуемся методами чтения и установки. При изменении значения `isPacked` мы должны произвести определенное действие (обновить пользовательский интерфейс), для чего достаточно завести еще один CSS-класс — `.packed`, применяемый к элементу, запомненному в свойстве `uiElement`. В классе `.packed` определен стиль `text-decoration: linethrough`.

Начнем с добавления закрытого свойства `_isPacked` и методов его чтения и установки, как показано в листинге ниже.

Листинг 3.7. Добавление свойства _isPacked и методов его чтения и установки

```

class PackItem {
  // остальной код опущен
  var _isPacked = false; ← Закрытое свойство _isPacked инициализируется значением false
                        ← (а не подразумеваемым по умолчанию значением null)

```

`bool get isPacked => _isPacked;` ← Метод чтения в короткой форме

`set isPacked(value) {` ← Метод установки в длинной форме

```

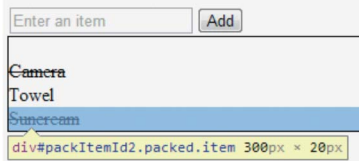
_isPacked = value;
if (_isPacked == true) {
    uiElement.classes.add("packed");
}
else {
    uiElement.classes.remove("packed");
}
}
}

```

Добавление
и удаление
CSS-класса
.packed

Последнее, что осталось сделать, – добавить прослушиватель щелчка по элементу `uiElement`, который будет переключать состояние между «упаковано» и «не упаковано». Элемент `uiElement`, как

PackList



- Щелчок мышью по упакованному предмету приводит к возникновению события `on.click` для элемента `<div>`

Прослушиватель события выполнит предложение `this.isPacked = ! this.isPacked;`, которое меняет значение свойства `isPacked` на противоположное

- Приложение отреагирует на это, добавив CSS-класс `.packed` в `<div>` предмета, в результате чего предмет получит стиль `{text-decoration:line-through; }`

Метод установки `isPacked` добавляет или удаляет CSS-класс `.packed`

- Повторный щелчок по предмету удаляет CSS-класс `packed` из элемента `<div>`

```

<div class="packed item">Suncream</div>
<div class="item">Suncream</div>

```

```

set isPacked(value) {
    _isPacked = value;
    if (_isPacked == true) {
        uiElement.classes.add("packed");
    }
    else {
        uiElement.classes.remove("packed");
    }
}

```

Рис. 3.9. При изменении свойства `isPacked` добавляется или удаляется CSS-класс `.packed`

любой `<div>`, может реагировать на события `on.click`, поэтому прослушиватель щелчка нужно добавить в метод чтения. Полужирным шрифтом выделен код самого прослушивателя, записанного в короткой форме, – он изменяет значение `isPacked` с `true` на `false` и обратно:

```
DivElement get uiElement {  
  if (_uiElement == null) {  
    _uiElement = new Element.tag("div");  
    _uiElement.classes.add("item");  
    _uiElement.text = this.itemText;  
    _uiElement.on.click.add( (event) => isPacked = !isPacked);  
  }  
  return _uiElement;  
}
```

Оператор **!** (не) меняет логическое значение `isPacked` на противоположное. В результате вызывается метод установки `isPacked`

И после нанесения этого последнего штриха мы получили работающее клиентское приложение, которое можно конвертировать в JavaScript и развернуть на любом веб-сервере в виде набора статических файлов, которые понимают все современные браузеры. Приложение состоит из одного Dart-класса с конструктором и методами чтения и установки.

3.4. Автономное тестирование программы

Тестирование написанного вами кода – стандартный этап разработки программного обеспечения. Инструменты Dart умеют выполнять статический анализ кода, принимая во внимание имеющуюся в нем информацию о типах, но возможность многократно прогонять автономные тесты в браузере и на сервере помогает контролировать качество программы. В Dart комплект автономных тестов – это отдельное приложение, которое разрабатывается параллельно с основным. Впоследствии это приложение можно запустить в браузере, который выполнит каждый тест и выведет результаты.

Пора уже перейти к коду. Добавьте в свою папку исходного кода новое приложение рядом с файлами `PackList.dart` и `PackList.html`. Файл `PackListTest.html`, содержащий автономные тесты, простой и короткий – в нем есть только ссылка на приложение `PackListTest.dart` (см. листинг ниже).

Листинг 3.8. PackListTest.html: начальный HTML для автономных тестов

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>PackListTest</title>
  </head>
  <body>
    <script type="application/dart" src="PackListTest.dart"></script>
  </body>
</html>
```

Ссылка на файл PackListTest.dart, содержащий точку входа в приложение

Имея HTML-файл для тестового приложения, мы можем начать добавлять код в файл PackListTest.dart. В него мы импортируем само тестируемое приложение PackList.dart, а также каркас автономного тестирования Dart и конфигурационный скрипт, предоставляемый этим каркасом. Конфигурационный скрипт связывает библиотеку автономного тестирования с набором функций вывода, которые отправляют результаты либо в окно браузера, либо на консоль сервера.

Ограничения на стороне клиента и сервера

Хотя программа, написанная на Dart, может исполняться как в браузере (на стороне клиента), так и на сервере, некоторые библиотеки предназначены только для клиента, а некоторые – только для сервера. Например, библиотека `dart:html` взаимодействует с браузерной моделью DOM и потому доступна, только когда Dart исполняется в браузере. На сервере она работать не будет, потому что никакой модели DOM там нет. Тестирование программы, которая импортирует `dart:html`, на сервере невозможно, так как серверная виртуальная машина не содержит браузерной модели DOM для манипуляций HTML-разметкой.

В листинге 3.9 приведен минимальный код, необходимый для того, чтобы запустить клиентский комплект тестов. При импорте библиотеки PackList.dart необходимо указать префикс в команде `import`, потому что иначе возникнет конфликт между двумя функциями `main()` верхнего уровня – из приложения PackList и из приложения PackListTest. Префикс в команде `import` позволяет эти функции различить (мы еще вернемся к этой теме в главе 5). Далее следует импортировать каркас автономного тестирования с помощью команды `pub install` из командной строки или из меню редактора Dart Editor.

Листинг 3.9. PackListTest.dart: стереотипный код комплекта клиентских тестов

```
import "package:unittest/unittest.dart";  ← Импортируется каркас автономного тестирования

import "package:unittest/html_config.dart";  ← Импортируется клиентский
                                              конфигурационный файл
// import "package:unittest/vm_config.dart";  ← При тестировании в серверной VM нужно
                                              было бы импортировать конфигурационный
                                              файл VM

import "PackList.dart" as packListApp;      ← Импортируется тестируемая библиотека

main() {
  useHtmlConfiguration();                  ← Настраивается клиентский HTML

  // useVmConfiguration();                 ← Сюда добавляются тесты

  // todo: добавить сюда тесты            ← При тестировании в серверной VM нужно было бы
}                                           настроить VM
```

Чтобы активировать управление пакетами, необходимо указать в файле `pubspec.yaml`, созданном редактором, зависимость, которая говорит о том, что мы собираемся использовать автономное тестирование. В главе 5 мы обсудим менеджер пакетов `pub` и синтаксис файла `pubspec` более подробно. А пока добавьте в файл `pubspec.yaml` своего приложения раздел зависимостей, показанный ниже:

```
dependencies:
  unittest: any
```

Затем выполните команду `pub install` из меню редактора, чтобы включить указанную зависимость от каркаса автономного тестирования в проект.

3.4.1. Создание автономных тестов

Теперь у нас имеется вся инфраструктура для написания автономных тестов. Тестом называется созданная вами функция, которая передается функции `test()` из библиотеки `unittest`, являющейся частью каркаса автономного тестирования. О том, как передавать функции, мы поговорим в следующей главе, а пока на рис. 3.10 продемонстрирован синтаксис создания теста.

Имя теста передается
в первом аргументе
функции test()

Вторым аргументом является
анонимная функция,
обертывающая сам код теста

```
test( "test name", () {
    // test code goes here
} );
```

Код автономного теста
находится внутри
анонимной функции

Рис. 3.10. Код, необходимый для создания автономного теста

3.4.2. Определение ожидаемых результатов теста

В автономном тесте может быть любой корректный Dart-код. Обычно автономный тест проверяет, совпадают ли результаты с ожиданиями, – например, что некоторое значение не равно null или что некоторому свойству объекта присвоено правильное значение. Для этого используется функция `expect()` (также из библиотеки `unittest`), которая принимает два параметра: фактическое значение, вычисленное в тесте, и сравнитель, который позволяет сопоставить это значение с ожидаемым. В листинге 3.10 приведен простой пример, содержащий два теста. В первом ожидается, что созданный экземпляр `PackItem` отличен от null, а во втором проверяется, что значение `itemText`, присвоенное в конструкторе, правильно возвращается в виде свойства объекта.

Листинг 3.10. Проверка ожидаемых результатов в автономных тестах

```
// стереотипные команды import
main() {
    useHtmlConfiguration();

    test("PackItem constructor", () {
        var item = new PackItem("Полотенце");
        Expect.isNotNull(item);
    });

    test("PackItem itemText property", () {
        var item = new PackItem("Полотенце");
        expect(item.itemText, equals("Полотенце"));
    });
}
```

← Определение автономного теста

← Стандартный Dart-код для создания `PackItem`

← Ожидается, что новый объект не равен null (проверяется с помощью статического метода `isNotNull`)

Во втором тесте создается еще один объект `PackItem`

← Ожидается, что свойству `itemText` правильно присвоено значение «Полотенце», для чего используется встроенная функция `equals()`

Результат работы этой программы показан на рис. 3.11. В окне браузера отображаются результаты каждого теста плюс трассировка стека для тестов, которые не прошли.

Автоматизация прогона тестов

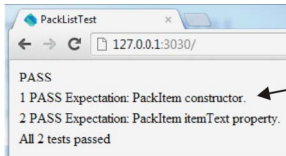
Прогон автономных тестов в браузере можно автоматизировать с помощью стороннего каркаса тестирования на стороне клиента, например Selenium (www.seleniumhq.org), который можно запускать на сервере непрерывной интеграции. Selenium может перейти на страницу автономных тестов приложения в браузере. Вы можете настроить Selenium, так чтобы он сообщал об успехе или неудаче в зависимости от того, что отображается в браузере. В Dart имеется также интерфейс к веб-драйверу Selenium, позволяющий писать скрипты для Selenium непосредственно на Dart. Подробнее об API веб-драйвера см. на сайте api.dartlang.org.

Таблица 3.1. Готовые сравнители для автономных тестов

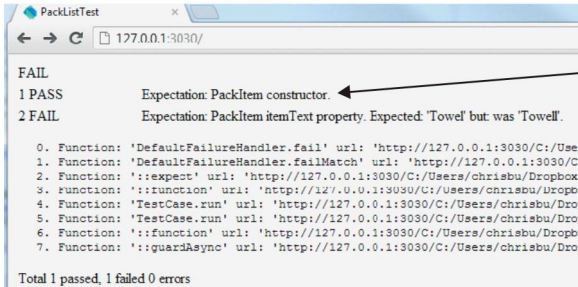
Сравнитель	Описание
isNull isNotNull	Ожидает, что фактическое значение равно или не равно <code>null</code>
isTrue isFalse	Ожидает, что фактическое значение равно <code>true</code> или <code>false</code>
same(expected)	Ожидает, что фактический объект совпадает с ожидаемым
equals(expected)	Ожидает, что фактический объект равен ожидаемому
isEmpty contains(expected)	Ожидает, что фактический список пуст или содержит ожидаемый объект
throws	Ожидает, что возбуждено какое-то исключение
throwsNullPointerException	Ожидает, что возбуждено конкретное исключение (существуют сравнители и для других исключений базовых типов)
anything	Ожидает произвольное значение

3.4.3. Создание пользовательского сравнителя

Чтобы создать свой сравнитель, нужно расширить базовый класс `BaseMatcher`. Подробно вопрос о расширении классов мы будем рассматривать в главах 6 и 7, но уже сейчас скажем, что в классе, расширяющем базовый сравнитель, нужно реализовать две функции: `matches()` и `describe()`. Функция `matches()` возвращает булевское значение, показывающее, были ли удовлетворены ожидания. В листинге



Результат каждого теста выводится в окно браузера (или на консоль сервера в случае прогона тестов на сервере)



Если тест не прошел, то для упрощения диагностики ошибки выводится также трассировка стека

Рис. 3.11. Отображение результатов автономных тестов в браузере

ниже приведен код сравнителя, который проверяет, что у двух объектов `PackItem` значение свойства `itemText` одинаково.

Листинг 3.11. Пользовательский сравнитель объектов `PackItem`

```
// стереотипный код опущен
class CustomMatcher extends BaseMatcher { ← CustomMatcher расширяет класс BaseMatcher
  PackItem _expected;

  CustomMatcher(this._expected); ← Конструктор принимает ожидаемое значение

  bool matches(PackItem actual) {
    if (_expected.itemText == actual.itemText) {
      return true;
    }
    else {
      return false;
    }
  }

  Description describe(Description description) { ← Описание для отображения
    description.add("itemText");                  в пользовательском интерфейсе
  }
}

main() {
```

Функция `matches()` возвращает `true`, если ожидаемое значение соответствует фактическому


```
useHtmlConfiguration();

test("PackItem custom", () {
  var packItem1 = new PackItem("Полотенце");
  var packItem2 = new PackItem("Полотенце");
  expect(packItem2, new CustomMatcher(packItem1)); ← тесте
});
}
```

В этом разделе мы бегло познакомились с автономным тестированием в Dart. Автономные тесты, прогоняемые после любого изменения кода, дают уверенность, что написанный вами код работает так, как ожидается.

Памятка

- Автономные тесты можно запускать как в браузере, так и на сервере.
 - Браузерные автономные тесты могут импортировать библиотеки, в которых используется `dart:html`.
 - Серверные автономные тесты не должны импортировать библиотеки, в которых используется `dart:html`.
 - Для проверки того, что фактический результат теста совпадает с ожидаемым, в автономных тестах используется функция `expect()` в сочетании со сравнителем.
 - Расширяя класс `BaseMatcher`, можно создавать собственные сравнители.
-

3.5. Резюме

Теперь вы знаете о таких синтаксических конструкциях, как классы и короткая форма функции; они будут часто встречаться на страницах этой книги. В этой главе мы затронули различные темы, которые в дальнейшем будем рассматривать более детально. Подведем некоторые итоги.

- Библиотека `dart:html` предоставляет элегантный способ взаимодействия с браузером за счет использования классов, свойств и списков Dart для манипулирования элементами модели DOM.
- В библиотеке `dart:html` определены события, для которых можно написать функции-прослушиватели. Такая функция принимает единственный параметр `event`. Одно событие могут прослушивать несколько функций.
- В Dart для функций имеется короткая форма записи для возврата результата вычисления одного выражения.

- ❑ Классы в Dart аналогичны классам в Java и C#: в них могут быть методы и свойства. Для определения методов чтения и установки свойств предусмотрены специальные ключевые слова `get` и `set`.
- ❑ Методы чтения и установки взаимозаменяемы со свойствами, то есть вызывающая программа не знает, к чему именно обращается: к самому свойству или к методу его чтения либо установки.
- ❑ Мы рассмотрели автономные тесты в Dart и узнали, что их можно прогонять в браузере или на сервере. Автономный тест сравнивает ожидаемый результат с фактическим, применяя для этой цели встроенный или написанный пользователем сравнитель.

В реальной программе мы, наверное, добавили бы возможность сохранять данные между сеансами работы в браузере, например в его локальном хранилище, а также возможность отправлять данные на сервер. Эти вопросы будут обсуждаться в частях III и IV.

После того как мы познакомились с основами разработки веб-приложений в среде Dart, настало время сменить курс и изучить язык Dart глубже. Рассматриваемые концепции применимы к программам, работающим как в браузере, так и на сервере. В следующей главе мы рассмотрим функции – кирпичики, из которых строится любое приложение. По ходу дела вы узнаете, что функции могут быть и переменными.



ЧАСТЬ II. ЯЗЫК DART

Из этой, самой длинной части книги вы узнаете о концепциях и средствах языка, предназначенных для структурирования приложений.

В главе 4 рассматриваются функции и замыкания – основные средства языка Dart, которые будут новыми для программистов, привыкших к языкам, основанным на классах, в частности Java и C#. Функции составляют самую суть Dart, причем анонимные функции используются повсеместно, например для обработки событий браузера и других систем.

В главе 5 вы узнаете, как структурировать Dart-приложение путем разбиения проекта на несколько библиотек. Хотя язык Dart основан на классах, наименьшей единицей ограничения доступа является не класс, а библиотека; вы увидите, как это можно с пользой применить.

В главах 6 и 7 речь пойдет о классах и интерфейсах и о том, как факультативная система типов позволяет реализовать в Dart динамическую типизацию (*duck typing*). Мы рассмотрим типы *dynamic* и *Object* и узнаем, как строить развитые иерархии наследования типов.

Глава 8 опирается на две предыдущие главы; в ней показано, как можно перегружать стандартные операторы и создавать обобщенные универсальные классы. Здесь же обсуждается, как эта техника применяется к построению обобщенных коллекций и как можно использовать обобщенные типы в своих приложениях.

Добравшись до главы 9, вы уже будете достаточно знать о синтаксисе Dart, чтобы снова вернуться к функциям. На этот раз мы рассмотрим, как они применяются в асинхронном программировании в качестве функций обратного вызова в сочетании с типом *Future*.

К концу этой части книги вы будете обладать знаниями о принципах и устройстве языка Dart и сможете эффективно использовать API, предоставляемые экосистемой Dart, для создания веб-приложений.



Глава 4. Функции и замыкания

В этой главе

- ❑ Объявление функции и типа возвращаемого значения.
- ❑ Обязательные и необязательные параметры.
- ❑ Сохранение функций в переменных.
- ❑ Использование функций в качестве замыканий.

Dart во многих отношениях похож на языки Java и C#, но в том, что касается функций, он больше напоминает JavaScript, чем языки с более строгой системой типов. В Dart все является объектами, в том числе функции, а это означает, что функцию можно сохранить в переменной и передавать из одного места приложения в другое точно так же, как `String`, `int` или любой другой объект. Таким образом, функции – это *полноценные* (first-class) объекты языка, как и все прочие типы, а не какие-то граждане второго сорта.

Мы начнем с изучения синтаксических вариантов записи функций, в том числе короткой и длинной нотаций, а также различных форм, зависящих от объема сообщаемой программистом факультативной информации. Мы будем использовать функции как переменные – передавать их в виде параметров и возвращать в качестве значений.

После того как вы освоитесь с использованием функций как полноценных объектов, которые можно сохранять в переменных, мы покажем, как определять типы функций для обеспечения строгой типизации. Эти определения помогут системе удостовериться, что функции, передаваемые из одного места приложения в другое, отвечают намерениям разработчика.

Наконец, мы рассмотрим замыкания, которые возникают, когда в функции используются переменные, определенные вне ее области видимости. Если эта функция (сохраненная в переменной) передается в другую часть приложения, то говорят о *замыканиях*. Это непростая тема, и – кстати – замыкания активно используются в JavaScript для эмуляции методов чтения и установки, а также

для реализации закрытых членов класса – в Dart эти механизмы уже встроены.

Примечание. Вообще говоря, все сказанное о функциях применимо и к методам, которые представляют собой особый вид функций, ассоциированных с классом. Классы мы будем подробно изучать в главе 6.

4.1. Функции в Dart

Компьютерная программа – это описание шагов, которые компьютер должен выполнить в определенном порядке, причем иногда выход одного шага является входом другого. В повседневной жизни мы постоянно используем функции с входными данными и выходными результатами. Например, готовя бетонную смесь на своем садовом участке (я часто ловлю себя на том, что больше возвожу в саду какие-то конструкции, чем выращиваю растения), я следую инструкциям по приготовлению качественного универсального бетона. Функция на каждом шаге инструкции принимает входные данные и порождает результаты. Названия функций выделены ниже полужирным шрифтом.

1. **Отмерить** нужное количество цемента (по объему).
2. **Отмерить** количество песка, по объему вдвое больше, чем цемента.
3. **Отмерить** количество щебня, по объему втрое больше, чем цемента.
4. **Смешать** цемент с песком, получив тем самым растворную смесь.
5. **Смешать** растворную смесь со щебнем, получив сухую бетонную смесь.
6. **Смешать** бетонную смесь с водой, получив влажную бетонную смесь.
7. **Уложить** бетон, прежде чем он схватился.

Функции `measure()` (Отмерить) и `mix()` (Смешать) используются несколько раз, получая на входе результат предыдущего шага и порождая новый результат. Смешение двух ингредиентов, например цемента и песка, дает новый ингредиент (растворную смесь), который можно использовать на следующих шагах процесса. Существует также функция `lay()` (Уложить), использованная только один раз. Отправная точка – начальный объем цемента – зависит от работы; в качестве приблизительной единицы измерения я обычно использую мешок.

Эти функции можно записать на языке Dart, как показано в листинге 4.1. Мы опустили классы, производные от `Ingredient`, которые функции возвращают в качестве значений, потому что для изложения они несущественны (но приведены в исходном коде к этой книге). Набор инструкций по изготовлению бетона составляет содержание функции `main()` – первой функции, исполняемой любым Dart-приложением.

Листинг 4.1. Процесс смешивания бетона на языке Dart

```
Ingredient mix(Ingredient item1, Ingredient item2) {
    return item1 + item2;
}
```

← Функции `mix()` и `measureQty()` возвращают ингредиенты


```
Ingredient measureQty(Ingredient ingredient,
                      int numberOfCementBags,
                      int proportion) {
    return ingredient * (numberOfCementBags * proportion);
}
```

Функция `measureQty()` возвращает новый ингредиент, вычисляя число, кратное количеству мешков цемента


```
void lay(ConcreteMix concreteMix) {
    // опущена, реализация не требуется
}
```



```
main() {
    Ingredient cement = new Cement();
    cement.bags = 2;
    print(cement.bags);

    Ingredient sand = measureQty (new Sand(), cement.bags, 2);
    Ingredient gravel = measureQty (new Gravel(), cement.bags, 3);

    Ingredient mortar = mix(cement, sand);
    Ingredient dryConcrete = mix(mortar, gravel);

    ConcreteMix wetConcrete = new ConcreteMix(dryConcrete, new Water());
    lay(wetConcrete);
}
```

← Задаем начальное количество мешков цемента

Вычисляем количество других ингредиентов, исходя из количества мешков цемента

Смешиваем ингредиенты, подавая выход предыдущей функции на вход последующей

Функции в Dart объявляются так же, как в Java или C#, – у них есть тип возвращаемого значения, имя и список параметров. В отличие от JavaScript, ключевое слово `function`, означающее, что это именно функция, не нужно. В отличие от Java и C#, типы парамет-

ров и возвращаемого значения необязательны, поскольку вся система типов в Dart факультативна.

Теперь, познакомившись с одним способом определения функций в Dart, рассмотрим некоторые другие, обращая внимание на факультативность типизации и наличие длинной и короткой нотаций. В главе 3 мы уже вкратце рассматривали длинный и короткий синтаксис записи функций в Dart: короткая форма позволяет определять однострочные функции, автоматически возвращающие результат вычисления выражения в этой единственной строке. Факультативность типизации означает, что типы возвращаемого значения и параметров можно не указывать. На рис. 4.1 показаны различные сочетания информации о типах и синтаксической нотации для функции `mix()`.

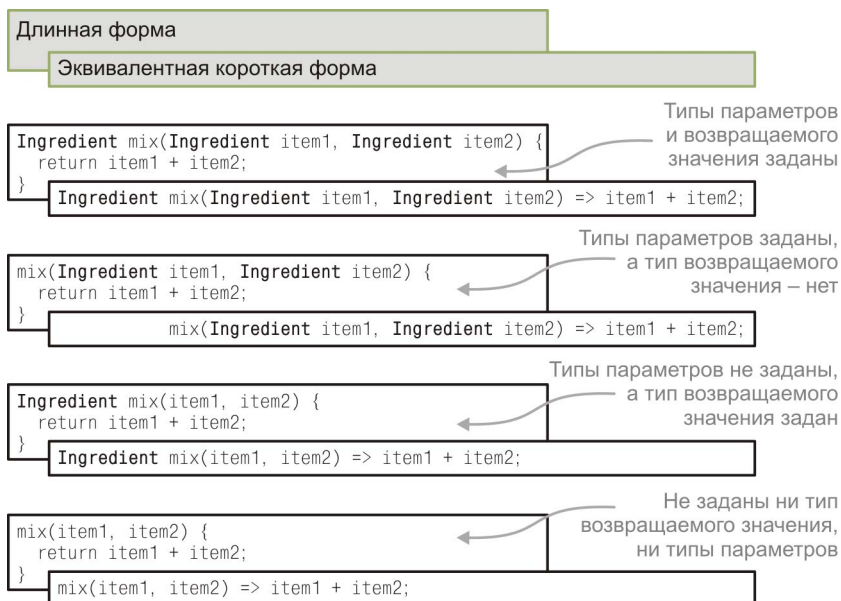


Рис. 4.1. Длинная и короткая формы записи функции `mix` в Dart

Короткая форма применима только для однострочных функций, тогда как длинная – для любых. Короткая форма удобна, когда код должен быть лаконичным и ясным. В следующем разделе мы увидим, что функции, представленные в короткой форме, автоматически возвращают результат вычисления выражения в строке.

4.1.1. Тип возвращаемого функцией значения и ключевое слово `return`

Все функции в Dart возвращают значение. Для однострочных функций в короткой форме записи таким значением является результат вычисления выражения в строке. Для функций в длинной форме возвращается значение `null`, если явно не указано противное.

Функция в короткой форме автоматически возвращает результат вычисления выражения в своей единственной строке; например, короткая форма функции `mix(item1, item2)` возвращает сумму `item1 + item2`, хотя явно это нигде не указано. Запись

`function_name()` => выражение;

можно произнести вслух как «вернуть результат выражения». С функциями в короткой форме так обстоит дело всегда, даже если выражение не порождает никакого значения, например сводится к вызову функции, не возвращающей значения. Поскольку по умолчанию возвращается значение `null`, то в отсутствие чего-либо другая однострочная функция этот `null` и возвращает.

С другой стороны, функции, записанные в длинной форме, по умолчанию возвращают `null`, если явно – с помощью ключевого слова `return` – не возвращается другое значение. Например, в длинной форме функции `mix()` мы пишем:

```
return item1 + item2;
```

Когда встречается `return`, результат сразу же возвращается вызывающей функции. Если в длинной форме записи функции ключевое слово `return` не используется, то функция возвращает `null`. На рис. 4.2 показано, как использовать `return` в длинной форме и вычисляемое выражение – в короткой. Показана также типичная ошибка – программист забыл написать `return`.

Задание типа возвращаемого значения

Тип возвращаемого значения можно указать перед именем функции. Поскольку типизация в Dart факультативна, задавать тип возвращаемого значения необязательно; в этом случае будет возвращен тип `dynamic`, как в примере ниже:

```
Ingredient mix(item1, item2) {...опущено...}
```

← Для функции можно указать тип возвращаемого значения

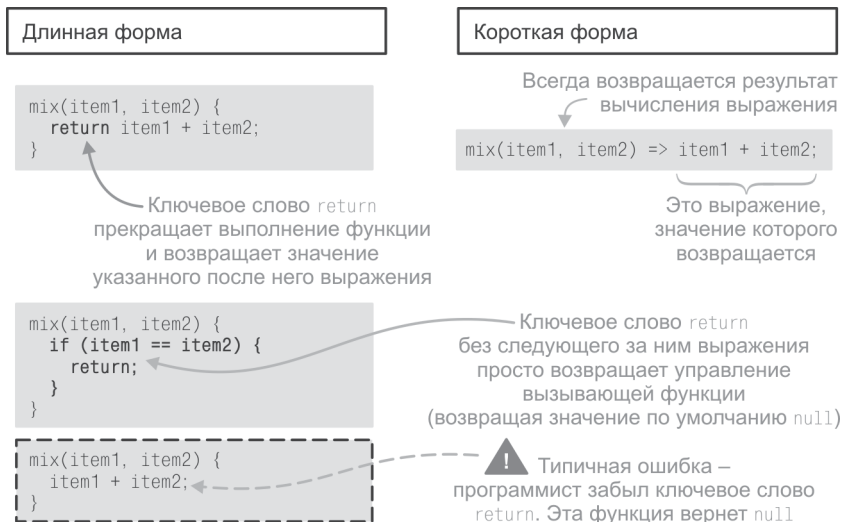


Рис. 4.2. В длинной форме для возврата значения необходимо ключевое слово `return`, тогда как в короткой автоматически возвращается результат вычисления выражения

<pre>mix(item1, item2) {...опущено...} dynamic mix(item1, item2) {...опущено...}</pre>	Тип возвращаемого значения не задан, поэтому возвращается тип <code>dynamic</code>

Если тип возвращаемого значения указан, то инструменты могут осуществить три вида проверки кода. Во-первых, будет выдано предупреждение, если в длинной форме записи функции отсутствует ключевое слово `return`. Во-вторых, проверяется, что фактически возвращенный тип совпадает с ожидаемым; в данном случае функция должна вернуть объект типа `Ingredient`. Третий способ контроля применяется, когда функция вызывается. Поскольку система типов в Dart является заодно и средством документирования, то проверяется, что вызывающая программа корректно использует указанный явно тип возвращаемого значения. Например, результат функции `mix()` – объект типа `Ingredient` – можно записать в переменную типа `ConcreteMix`, которую затем попытаться передать функции `lay`. Если тип возвращаемого значения не указан, то такой код допустим, но работать будет неправильно. Если же документировано, что ваша функция возвращает значение типа `Ingredient`, то инструментальные средства могут сообщить об ошибке:

Инструмент предупредит, что
объект типа `Ingredient` нельзя
присваивать объекту типа

```
Ingredient mixture = mix(new Sand(), new Gravel()); ← ConcreteMix  
lay(mixture);    ← Песка с щебенкой недостаточно для создания прочной конструкции
```

Функция `mix()` явно возвращает значение типа `Ingredient`, однако функция `lay()` ожидает получить объект типа `ConcreteMix`. Не имея явной информации о типе, Dart разрешил бы выполнить этот код (и, скорее всего, дело закончилось бы ошибкой в другом месте).

Использование типа `void` для указания отсутствия возвращаемого значения

Но при таком поведении возникает новая проблема: как сказать, что функция вообще не возвращает значения? На самом деле в Dart все функции возвращают какое-то значение, но ключевое слово `void` позволяет сообщить модулю проверки типов, что вызывающая программа не должна ожидать никакого значения от вашей функции.

Допустим, мы собираемся поручить приготовление бетона небольшой бетономешалке. Для этого можно написать функцию, которая запускает мешалку и не возвращает значения:

```
startMixer(speed) {  
    Mixer.start(speed);  
}
```

Поскольку в Dart все функции что-то возвращают, то этот код автоматически вернет вызывающей программе значение `null`, подразумеваемое по умолчанию. Пользователь так объявленной функции `startMixer()` не может сказать, собирался ли ее автор вернуть `null` или просто забыл написать `return` перед вызовом `Mixer.start()` (что, возможно, является ошибкой в программе). Если попытаться сохранить результат `startMixer` в переменной, то она получит значение `null`:

```
var runningMixer = startMixer("slow"); ← Переменная runningMixer содержит null, потому  
                                         что функция startMixer() не возвращает значения  
                                         явно
```

Но, указав в качестве типа возвращаемого значения ключевое слово `void`, вы сообщаете пользователю функции, что не собирались ничего возвращать. В этом случае у модуля проверки типов будет достаточно

информации, чтобы выдать предупреждение при попытке как-то использовать возвращаемое значение. Функция по-прежнему возвращает `null`, однако вас предупредят о том, что использовать это значение не следует. Объявление функции можно модифицировать следующим образом:

```
void startMixer(speed) { ← Ключевое слово void явно говорит о том, что никакое значение
    Mixer.start(speed);   не возвращается
}
```

Если в качестве возвращаемого типа указан `void`, то проверяется также, что в функции нигде не используется слово `return` с указанием возвращаемого значения.

4.1.2. Передача функции данных с помощью параметров

Функции в Dart не только возвращают значения, но и принимают параметры. В отличие от JavaScript, имена всех параметров необходимо перечислить в определении функции. В JavaScript список параметров необязателен, потому что переданные функции аргументы можно получить динамически с помощью встроенной переменной `arguments`. Dart же, являясь более структурированным языком, требует, чтобы все параметры были определены – как в Java и C#. В Dart не поддерживается перегрузка функций (и методов), зато есть необязательные параметры.

Совет. Между длинной и короткой формами записи функции нет отличий в части определения параметров, поэтому все примеры в этом разделе приведены только для длинной формы, но в равной мере относятся и к короткой.

Функция `measureQty(ingredient, numberOfCementBags, proportion)` принимает три параметра: `ingredient`, `numberOfCementBags` и `proportion`. Она может использовать их точно так же, как переменные, определенные в ее области видимости. Ниже показаны типизированная и нетипизированная сигнатуры функции:

```
measureQty(Ingredient ingredient,
           int numberOfCementBags,
           int proportion) {
    // ...опущено...
}
```

В типизированной версии
задана информация о типах

```
calculateQty (ingredient,
              numberOfCementBags,
              proportion) {
  // ...опущено...
}
```

```
calculateQty (dynamic ingredient,
              dynamic numberOfCementBags,
              dynamic proportion) {
  // ...опущено...
}
```

Нетипизированная версия эквивалентна версии с типом `dynamic`

Указывая для формальных параметров функции типы, например `Ingredient` и `int`, вы тем самым объявляете, что типы фактического и формального параметров должны быть связаны отношением *являются*. Это позволяет модулю проверки типов удостовериться в том, что вызывающая программа передает функции `measureQty()` аргументы допустимых типов. Вызывающая программа должна передавать аргументы в том порядке, в котором они перечислены в объявлении функции; в данном случае первым должен быть передан аргумент `ingredient`, вторым — `numberOfCementBags` и последним — `proportion`:

```
var sand = measureQty(new Sand(), cement.bags, 2);
```

Аргументы передаются по ссылке

При вызове функции `measureQty()` каждый аргумент, например ингредиент песок, передается как ссылка на значение. Поэтому функция может изменять свойства ингредиента (скажем, оставшееся количество), но не сам ингредиент (см. рис. 4.3).

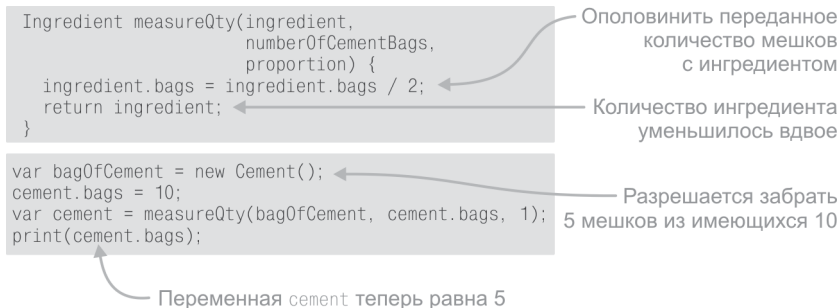


Рис. 4.3. При передаче объекта по ссылке можно изменять свойства объекта, но не сам объект

Например, что случится, если в функции `measureQty()` мы создадим новый мешок песка (возьмем со склада), поняв, что на вход было передано недостаточно мешков? Оказывается, что исходный ингредиент останется нетронутым. В листинге ниже эта мысль продемонстрирована на примере.

Листинг 4.2. Доказательство того, что параметры передаются по ссылке

```
Ingredient measureQty(ingredient, numberOfCementBags, proportion) {
    if (ingredient.bags == 0) { ← Функция определяет, что передано недостаточное
                                количество ингредиента
        ingredient = new Ingredient(); ← Вместо переданного ингредиента создаем новый...
        ingredient.bags = numberOfCementBags * proportion; ← ...но замена действует только
        return ingredient;                                     в области видимости функции.
    }                                                         Исходный мешок
}                                                            по-прежнему пуст

main() {
    var emptyBagOfCement = new Cement();
    emptyBagOfCement.bags = 0;

    var cement = measureQty(emptyBagOfCement, 1, 1); ← передаем его
    print(emptyBagOfCement.bags); ← Исходный объект emptyBagOfCement не изменился
}
```

Сказанное остается в силе независимо от того, модифицируется объект, например `ingredient`, или примитивный тип, например `int`, потому что в Dart все является объектом (как было сказано в предыдущей главе). Таким образом, изменяя внутри функции переданную ссылку на объект, мы всего лишь теряем старую ссылку и создаем новую. При выходе из функции новая ссылка будет потеряна, а у вызывающей программы остается та же ссылка на объект, которая и была раньше.

Необязательные позиционные параметры

У функций в Dart могут быть необязательные параметры, имеющие значения по умолчанию. В объявлении функции можно указать, какие параметры разрешено передавать вызывающей программе. Но если вызывающая программа не передаст их, то функция возьмет значения по умолчанию.

Для отмеривания количества ингредиента, пропорционального количеству мешков цемента, мы вызываем функцию `measureQty()`,

которая возвращает новый ингредиент в количестве, зависящем от числа мешков цемента и коэффициента пропорциональности. Иногда соотношение между числом мешков цемента и нового ингредиента 1:1. Чтобы упростить этот случай без использования необязательных параметров, мы могли бы модифицировать функцию: если переданный параметр `numberOfCementBags` или `proportion` равен `null`, то считать его равным 1:

```
measureQty(ingredient, int numberOfCementBags, int proportion) {  
  if (numberOfCementBags == null) numberOfCementBags = 1;  
  if (proportion == null) proportion = 1;  
  return ingredient * (numberOfCementBags * proportion);  
}
```

Но вызывающая программа должна указать количество мешков и коэффициент пропорциональности. Это верно даже в том случае, когда вызывающая программа хочет передать стандартное значение, подразумеваемое по умолчанию, например коэффициент пропорциональности, равный 1. Вызывающий код должен выглядеть примерно так:

```
measureQty(new Sand(), null, null);  
measureQty(new Sand(), 1, null);  
measureQty(new Sand(), null, 1);  
measureQty(new Sand(), 1, 1);
```

Такие аргументы называются позиционными; их позиция в списке аргументов существенна. Например, в третьем аргументе всегда передается коэффициент пропорциональности, а в первом – ингредиент.

Было бы лучше, если бы вызывающая программа могла передать только необходимые значения, например ингредиент и коэффициент пропорциональности, опустив количество мешков, если оно не нужно. Именно для этой цели в Dart существуют необязательные параметры. Необязательные параметры должны располагаться после всех позиционных. Блок необязательных параметров заключается в квадратные скобки, а параметры в нем, как обычно, разделяются запятыми. Например, в рассмотренной выше функции необязательные параметры можно было бы использовать следующим образом (выделено полужирным шрифтом):

```
measureQty(ingredient, [int numberOfCementBags, int proportion]) {  
    // ... опущено...  
}
```

Теперь вызывающая программа может передавать значения параметров `numberOfCementBags` и `proportion` только в случае, когда это действительно нужно.

Фактические значения необязательных параметров передаются в порядке объявления последних, например:

<pre>measureQty(new Sand(), 2, 1); measureQty(new Sand(), 2); measureQty(new Sand());</pre>	Необязательные параметры можно опускать, если они не нужны

Разумеется, и в этом случае переданные параметры инициализируются значением `null`, то есть `measureQty()` должна по-прежнему проверять параметры на `null` и по умолчанию брать значение 1. По счастью, значение по умолчанию можно указать в объявлении необязательного параметра:

<pre>measureQty(ingredient, [int numberOfCementBags=1, int proportion=1]) { return ingredient * (numberOfCementBags * proportion); }</pre>	Для необязательных параметров заданы значения по умолчанию

Теперь вызывающая программа может передавать или не передавать параметры, а ваша функция не обязана выполнять проверку на `null`. Как уже было сказано, обязательные параметры должны быть переданы все и в правильном порядке. Для необязательных параметров это уже не так. Вызывающая программа может использовать необязательные параметры двумя способами.

- ❑ Вообще не передавать значений. Тогда функция возьмет значение по умолчанию или `null`, если таковое не задано.
- ❑ Передать значения некоторых необязательных параметров. Переданные значения сопоставляются с параметрами в порядке объявления слева направо. Параметры, на которые не хватило переданных значений, принимают значения по умолчанию или `null`.

Необязательные именованные параметры

Альтернативой необязательным позиционным параметрам являются необязательные именованные параметры. В этом случае вы-

зывающая программа может указывать параметры, для которых передает значения, в любом порядке. Как и раньше, обязательные параметры должны предшествовать необязательным, но теперь необязательные параметры заключаются в фигурные скобки с указанием значений по умолчанию:

```
measureQty(ingredient, {int numberOfCementBags:1, int proportion:1}) {  
    return ingredient * (numberOfCementBags * proportion);  
}
```

Необязательные именованные
параметры заключаются
в фигурные скобки

Отметим, что, в отличие от необязательных позиционных параметров, значение по умолчанию отделяется от имени параметра двоеточием (:).

Теперь вызывающая программа может вызвать функцию `measureQty()`, передав обязательный аргумент `ingredient` и, быть может, необязательные аргументы `numberOfCementBags` и `proportion` в любом порядке:

```
measureQty(new Sand(), numberOfCementBags: 2, proportion: 1);  
measureQty(new Sand(), numberOfCementBags: 2);  
measureQty(new Sand(), proportion: 1);  
measureQty(new Sand());
```

Необязательные
именованные
параметры передают-
ся с указанием имени

В отличие от необязательных позиционных параметров, вызывающая программа должна указывать имена передаваемых именованных параметров. Следовательно, такая запись недопустима:

```
measureQty(new Sand(), 2, 1);
```

поскольку необязательные параметры в ней не поименованы.

Примечание. В случае обязательных позиционных параметров вызывающая программа не обязана знать имена параметров. Напротив, необязательные именованные параметры составляют часть API функции, поэтому к придумыванию хороших имен для них следует относиться так же серьезно, как к придумыванию имен функций и методов. Изменение имени необязательного параметра после выпуска кода в обращение может отразиться на его пользователях так же, как изменение имени функции.

Нельзя задавать в объявлении одной функции как позиционные, так и именованные необязательные параметры. Выбирайте тот

способ, который лучше отвечает предполагаемому использованию функции.

На рис. 4.4 показаны различные способы вызвать функцию `measureQty()`.



Рис. 4.4. Различные способы передачи значений необязательных позиционных и именованных параметров функции

Памятка

- Функции, записанные в короткой форме, автоматически возвращают результат вычисления выражения в однострочном теле функции.
- Функции, записанные в длинной форме, должны возвращать значение с помощью ключевого слова `return`; в противном случае автоматически возвращается `null`.
- Чтобы сообщить модулю проверки типов, что вы не собирались возвращать значение, нужно указать в качестве возвращаемого типа `void`.
- Информация о параметрах типов факультативна.
- Необязательные позиционные параметры можно перечислить через запятую, заключив в квадратные скобки и поместив после объявления обязательных параметров.
- Вызывающая программа может ссылаться на необязательный параметр по имени с помощью синтаксиса *имя:значение*.

Теперь вы знаете, как выглядит функция, как она вызывается, как задать тип возвращаемого значения и как объявить обязательные и необязательные параметры. Самое время посмотреть, что еще можно

делать с функциями. Точнее, мы поговорим о том, как запоминать функции в переменных и передавать их в виде аргументов другим функциям.

4.2. Функции как полноценные объекты

Говоря, что функция является *полноценной*, мы имеем в виду, что ее можно запомнить в переменной и передать из одного места приложения в другое. Для полноценных функций не существует какого-то особого синтаксиса, в языке Dart все функции являются полноценными. Чтобы обратиться к функции как к объекту (а не просто вызвать ее), нужно указать ее имя без скобок, в которых обычно передаются параметры. В таком случае мы получаем доступ к объекту-функции.

Рассмотрим функцию `Ingredient mix(item1, item2)`, о которой шла речь выше в этой главе. Чтобы ее вызвать, нужно написать имя функции и список фактических аргументов в круглых скобках: `mix(sand, cement);`. Если же написать только ее имя, опустив скобки и аргументы, то мы получим ссылку на объект-функцию, который можно использовать точно так же, как `String` или `int`. Запомнив объект-функцию в переменной, мы сможем воспользоваться им снова и снова, как показано в следующем фрагменте:

```
var mortar = mix(sand, cement);  ← Функция mix вызывается, а возвращенное ей значение
                                запоминается в переменной mortar
var mixFunction = mix;          ← В переменной mixFunction запоминается сама функция
                                функция — это одновременно Object
var dryConcrete = mixFunction(mortar, gravel); ← и Function, так что оба предложения
                                                печатают true
print(mix is Object);           | Функция, хранящаяся в переменной mixFunction, вызывается,
print(mix is Function);         | чтобы приготовить еще немного сухой смеси
```

Из этого примера видно, что функция `mix()` и переменная `mixFunction` связаны отношением *является* как с типом `Object` (напомним, все *является* объектом), так и с классом `Function`, который представляет тип функции.

И тут открывается интересная возможность. Если функцию можно сохранить в переменной, то так ли нужно сначала объявлять ее в области видимости верхнего уровня? Вовсе нет, вполне допустимо

объявить функцию по месту (в теле другой функции) и запомнить в переменной, не засоряя область видимости верхнего уровня в библиотеке. На самом деле есть три способа объявить функцию по месту и один – в области видимости верхнего уровня. Все они показаны на рис. 4.5, где объявления функций выделены жирным шрифтом. Ниже мы подробно рассмотрим каждый из этих примеров.

Функция `mix1()` объявлена в области видимости верхнего уровня, то есть вне функции или класса

Скрипт или библиотечный Dart-файл

```

mix1(item1, item2) {
  return item1 + item2;
}

main() {
  mix2(item1, item2) {
    return item1 + item2;
  }

  var mix3 = (item1, item2) {
    return item1 + item2;
  };

  var mix4 = mixer(item1, item2) {
    return item1 + item2;
  };

  mix1(new Sand(), new Gravel());
  mix2(new Sand(), new Gravel());
  mix3(new Sand(), new Gravel());
  mix4(new Sand(), new Gravel());
  // mixer(new Sand(), new Gravel());
  print(mix1);
  print(mix2);
  print(mix3);
  print(mix4);
}

```

Функция `main()` открывает новую область видимости, в которой можно объявлять переменные и функции

Функция `mix2()` определена внутри области видимости функции. Синтаксис при этом такой же, как в случае `mix1()`

⚠ Обратите внимание на отсутствие ; после объявления

Эта функция объявлена анонимно и присвоена переменной `mix3`. Обратите внимание, что предложение присваивания завершается знаком ;

Эта функция – `mixer()` – объявлена и присвоена переменной `mix4`. Функция `mixer()` не существует вне области видимости предложения присваивания. Обратите внимание, что и на этот раз предложение присваивания завершается знаком ;

⚠ Поскольку `mixer()` не существует вне своей собственной области видимости, то попытка вызвать ее в этом месте приведет к ошибке. Функция `mixer()` присвоена переменной `mix4`

Теперь у нас есть ссылки на все четыре функции, и их можно использовать в приложении

Рис. 4.5. Четыре разных способа объявить функцию. `mix1()` объявлена в области видимости верхнего уровня, а остальные три – в теле другой функции

Области видимости функции и библиотеки

Самой широкой областью видимости в Dart является библиотека, и все функции, не погруженные в какой-то другой блок, например в определение другой функции или класса (в последнем случае они называются методами), находятся в области видимости библиотеки. Говорят, что они определены на верхнем уровне библиотеки.

Разрешается объявлять одни функции внутри других. Объявленная таким образом внутренняя функция находится в области видимости внешней функции, как и любая объявленная там же переменная – скажем, типа `String` или `int`. Обратиться к функции в области видимости другой функции можно лишь из блока, в котором она объявлена, если только не передать ее другой функции в качестве параметра или не вернуть в качестве значения (и здесь наблюдается полная аналогия с другими переменными). Примеры такого рода мы увидим, когда будем обсуждать замыкания.

Мы уже пользовались областью видимости верхнего уровня библиотеки для объявления функций, подобных `mix1()`; такие функции называются *библиотечными*. Функции трех остальных видов – объявленные в теле объемлющей функции – называются *локальными* и заслуживают более пристального внимания. Эта часть Dart выглядит обманчиво простой, но, как и замыкания, может оказаться куда сложнее, чем представляется на первый взгляд.

4.2.1. Объявления локальных функций

Локальными называются функции, объявленные в теле другой функции. В отличие от библиотечных функций, на локальную функцию нельзя явно сослаться вне той функции, в которой она объявлена. Однако ее можно передать другой функции и вернуть в качестве значения той функции, где она объявлена. Несколько возвращенных таким образом локальных функций можно запомнить в списке и в каком-то другом месте вызывать по очереди.

В листинге 4.3. определена и используется функция `combineIngredients()`, которая принимает в качестве параметров функцию смешивания и два ингредиента. Поскольку функция смешивания произвольна, то `combineIngredients()` позволяет реализовать любую стратегию – хоть лопатой, хоть в бетономешалке. Я неоднократно буду возвращаться к функции `combineIngredients()` в этом разделе.

Листинг 4.3. Функция, принимающая другую функцию в качестве параметра

Ingredient combineIngredients(mixFunc, item1, item2) { ← Первый параметр –
это функция смешивания.

```
return mixFunc(item1, item2);
```

← Функция `mixFunc()` смешивает два ингредиента и возвращает результат

Остальные два параметра – смешиваемые ингредиенты

```
}
```

Теперь, поняв, как можно использовать объект-функцию, хранящуюся в переменной, рассмотрим все три способа объявления локальных функций, начав с самого простого. Во всех примерах ниже используется длинная форма записи, но те же самые правила применимы и к короткой форме.

Объявление простой локальной функции

Синтаксис объявления простой локальной функции, показанный на рис. 4.6, ничем не отличается от синтаксиса объявления библиотечных функций в области видимости верхнего уровня, разве что само объявление находится внутри другой функции. Имя функции одновременно является ссылкой на нее, как в случае `mix2()`; а поскольку имя функции принадлежит ее области видимости, то функцию можно вызывать рекурсивно (подробнее о рекурсивных функциях мы будем говорить ниже в этой главе).

Функция `mix2()` определена в области видимости другой функции. Синтаксис такой же, как для `mix1()`

```
mix2(item1, item2) {
    return item1 + item2;
}
```

⚠ Обратите внимание на отсутствие ; после объявления

Рис. 4.6. Синтаксис объявления простой локальной функции

В объявлении простой локальной функции завершающая точка с запятой не нужна, потому что закрывающая фигурная скобка сама является завершителем – как и при объявлении функции в области видимости верхнего уровня. Это существенный момент, потому что в двух других способах объявления – когда функция явно присваивается переменной – точка с запятой после закрывающей скобки необходима.

В объявлении функции можно добавить информацию о типах – так же, как для библиотечных функций. В листинге ниже внутри функ-

ции `main()` определена строго типизированная функция `mix2()`, которая передается в качестве параметра функции `combineIngredients()`.

Листинг 4.4. Внешняя функция `main()` использует внутреннюю функцию `mix2()`

```
main() {
  mix2(item1, item2) {
    return item1 + item2;
  }

  var cement = new Cement();
  var sand = new Sand();
  var mortar = combineIngredients(mix2, cement, sand);
  var dryCement = mix2(mortar, new Gravel());
}
```

Локальная функция `mix2()`,
из которой образован
объект-функция

Объект-функцию `mix2` можно
передавать другим функциям
← как обычную переменную
Функцию `mix2()` можно вызвать
и напрямую в той области видимости,
где она объявлена

Имя `mix2` существует в области видимости самой функции `mix2()`. А раз так, то функция `mix2()` может вызывать саму себя (рекурсивно) или ссылаться на соответствующий ей же объект-функцию.

Рекурсия

Рекурсивной называется функция, которая вызывает саму себя. Такой прием часто применяется в функциях сортировки и при обходе деревьев. Рассмотрим следующую рекурсивную функцию `stir()`, которая вызывает сама себя, пока счетчик перемешиваний (`stirCount`) меньше 10:

```
stir(ingredient, stirCount) {
  print("Перемешивается $ingredient");
  if (stirCount < 10) {
    stirCount++;
    stir(ingredient, stirCount);
  }
}
```

В Интернете достаточно ресурсов, посвященных использованию рекурсии в информатике, – она находит применение отнюдь не только в Dart. Но чтобы эта техника работала, важно, чтобы имя функции было доступно в ее собственной области видимости. Иначе говоря, функция `stir()` должна быть видна в точке, где она вызывается. Имена простых локальных функций, равно как и функций в области видимости библиотеки, принадлежат их собственной области видимости, поэтому такие функции можно вызывать рекурсивно.

Объявление анонимной функции

При объявлении анонимной функции имя не указывается (см. рис. 4.7). Как и функцию, объявленную любым другим способом, ее

можно присвоить переменной, передать другой функции в качестве параметра или вернуть в качестве значения функции. Но рекурсивно вызвать ее невозможно, потому что она не имеет имени, по которому к ней можно было бы обратиться в ее собственной области видимости.

```
var mix3 = (item1, item2) {  
    return item1 + item2;  
};
```

Эта функция объявлена анонимно и присвоена переменной `mix3`. Обратите внимание, что предложение присваивания завершается знаком `;`;

Рис. 4.7. Объявление анонимной функции

В длинной форме записи анонимной функции необходима завершающая точка с запятой, потому что мы присваиваем значение переменной (`mix3`), а предложение присваивания всегда должно завершаться точкой с запятой.

Анонимная функция начинается открывающей круглой скобкой и продолжается до конца тела функции. В короткой форме анонимную функцию можно записать так:

```
() => null;
```

Функцию, записанную в таком виде, вообще невозможно вызвать, потому что не существует имени, по которому на нее можно было бы сослаться. Тем не менее это корректная анонимная функция, которая не принимает параметров и возвращает значение `null`.

Анонимные функции часто передаются напрямую другой функции в качестве параметра или сохраняются в списке. Оба способа работают, потому что ссылка на объект-функцию сохраняется либо в параметре, у которого есть имя, либо в элементе списка. В листинге 4.5 эта идея продемонстрирована в действии: мы создаем список анонимных функций для обработки дополнительных ингредиентов, а затем по очереди вызываем каждую функцию. Анонимные функции выделены полужирным шрифтом, причем показана как длинная, так и короткая форма. Анонимные функции передаются в качестве параметра методу `List.add()` и в дальнейшем могут быть извлечены из списка (как `String`, `int` или любой другой объект).

Листинг 4.5. Сохранение анонимных функций в списке

```
main() {  
    List taskList = new List();
```

```
taskList.add( (item) => item.pour() );
```

← В список добавляется анонимная функция
в короткой форме

```
taskList.add( (item) {
    item.level();
    item.store();
} );
```

В список добавляется
анонимная функция
в длинной форме

```
var aggregate = new Aggregate();

foreach(task in taskList) {
    task(aggregate);
}
```

← Обход списка
← По очереди вызывается каждая функция

Этот подход часто применяется к обработчикам событий браузера. Мы можем определить несколько обработчиков одного и того же события, например нажатия кнопки; для этого нужно передать функцию-обработчик методу `button.onClick.add()`, который сохраняет ее в списке функций, вызываемых по нажатии кнопки. Никто не мешает применять эту технику к функциям, имеющим имя, но часто – в особенности для однострочных функций – проще и понятнее передать анонимную функцию, как в следующем обращении к функции `combineIngredients`, которая принимает функцию в качестве параметра. Анонимная функция выделена полужирным шрифтом.

```
combineIngredients( (item1, item2) => item1 + item2, sand, gravel);
```

Мы, как и раньше, можем задать типы параметров, но не тип возвращаемого значения, поскольку Dart рассматривает любой текст левее открывающей скобки как имя функции (таким образом, функция окажется именованной, а не анонимной). Ниже определена функция с именем `Ingredient`, а не анонимная функция, возвращающая ингредиент:

```
Ingredient (item) => item.openBag();
```

Эту проблему можно разрешить, воспользовавшись третьим и последним способом объявления функций: присваиванием.

Объявление именованной функции в присваивании

Третий способ объявления функции представляет собой гибрид двух предыдущих: объявляется именованная функция, которая сра-

зу же присваивается переменной. Поскольку это присваивание, то, как и в предыдущем примере, в конце предложения должна стоять точка с запятой (рис. 4.8).

```
var mix4 = mixer(item1, item2) {
    return item1 + item2;
};
```

Функция `mixer()` объявляется и присваивается переменной `mix4`. Функция `mixer()` не существует вне области видимости присваивания. Обратите также внимание, что предложение присваивания завершается знаком `;`

Рис. 4.8. Объявление именованной функции в предложении присваивания

Достоинство данного подхода заключается в том, что мы можем объявить возвращаемый тип, а имя функции оказывается в ее области видимости, что позволяет вызывать ее рекурсивно. В приведенном выше примере имя функции `mixer()` находится в ее же области видимости, причем доступно оно *только* в области видимости этой функции. Обратиться к ней извне можно лишь по имени `mix4`.

Ниже показано, как можно переписать функцию `mix4` в виде, допускающем рекурсию и возможность проверить возвращаемый тип при передаче ее в виде аргумента функции `combineIngredients()`.

Листинг 4.6. Рекурсивная, строго типизированная, именованная функция

<pre>main() { var mix4 = Ingredient mixer(Ingredient item1, Ingredient item2) { if (item1 is Sand) { return mixer(item2, item1); } else { return item1 + item2; } } var sand = new Sand(); var gravel = new Gravel(); combineIngredients(mix4, sand, gravel); }</pre>	В объявлении функции указано имя, а также типы параметров и возвращаемого значения, сама функция присвоена переменной <code>mix4</code> ← Имя функции <code>mixer()</code> находится в ее области видимости, что позволяет вызывать ее рекурсивно Передается переменная <code>mix4</code>, являющаяся ссылкой на функцию <code>mixer()</code>

Имя `mixer()` по существу мало кому нужно. Оно доступно только внутри области видимости функции, а больше нигде не видно. Ес-

ли мы объявляем функцию `mixer()` непосредственно при передаче другой функции в виде аргумента, то сослаться на нее по имени `mixer()` не можем нигде, кроме как внутри нее самой. Следующий пример (рис. 4.9) выглядит почти так же, как объявление простой локальной функции, с которого мы начали рассмотрение, но имеет одно тонкое отличие: объявленная функция неявно присваивается параметру функции `combineIngredients()`.

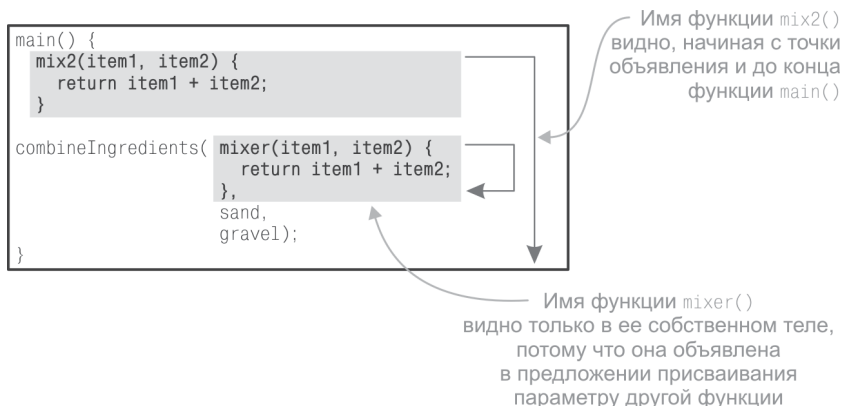


Рис. 4.9. Именованная функция, объявленная как аргумент другой функции, может ссылаться на саму себя по имени. Ни в каком другом месте это имя недоступно

Мы рассмотрели объявление функций и присваивание их переменным и параметрам функций, но как быть с системой типов в Dart? Откуда нам знать, что функция `combineIngredients()` принимает другую функцию в качестве первого параметра? К счастью, Dart имеет средства для строгой типизации функций – в виде ключевого слова `typedef`, к обсуждению которого я и перейду.

4.2.2. Определение строгого типа функции

До сих пор мы сохраняли объекты-функции в динамически типизированных переменных и передавали их другим функциям в виде динамических параметров. У этого подхода имеется одна проблема: факультативная система типизации в Dart позволяет задать тип переменной или параметра функции, но как сказать, что этот тип – функция? Мы уже видели, что функция *является* одновременно

`Object` и `Function`, поэтому можно было бы воспользоваться этими типами, как показано в листинге ниже.

Листинг 4.7. Тип `Function` строго определяет тип переменной или параметра

```
Ingredient combineIngredients(Function mixFunc, item1, item2) {
    return mixFunc(item1, item2);
}
```

Параметр `mixFunc` имеет строгий тип `Function`

```
main() {
    Function mix = (item1, item2) {
        return item1 + item2;
    }
```

Функция сохраняется в переменной `mix` со строгим типом `Function`

```
var sand = new Sand();
var gravel = new Gravel();
combineIngredients(mix, sand, gravel);
}
```

При передаче `mix` в функцию `combineIngredients` модуль проверки типов может убедиться, что первый параметр — действительно функция

Используя объект-функцию, запомненный в переменной, мы на самом деле работаем с экземпляром класса `Function`. Но не все экземпляры функций одинаковы. Функция `mix()` отличается от `measureQty()`, а та отличается от `lay()`. Нам необходим способ, который позволил бы строго типизировать первый параметр функции `combineIngredients()`, сообщив, что это именно функция `mix()`, а не какая-то другая.

В Dart есть два способа решить эту задачу. Первый проще, но несколько многословен: указать сигнатуру функции прямо в определении параметра, как показано на рис. 4.10. Тем самым мы объявляем, что параметр функции должен иметь вполне определенную сигнатуру. Но представьте, что есть 10 функций, принимающих `mix()` в качестве параметра; тогда ее сигнатуру пришлось бы выписать 10 раз. По счастью, Dart позволяет объявить сигнатуру функции отдельно с помощью ключевого слова `typedef`, описывающего тип функции. В предложении `typedef` объявляется только сигнатура функции, а не сама функция или объект-функция. Использовать `typedef` разрешено лишь в области видимости библиотеки, а не внутри функции. В следующем листинге показано, как с помощью `typedef` определить сигнатуру функции, которой можно воспользоваться при объявлении параметра `mixFunc` функции `combineIngredients()`.

1. Объявление `combineIngredients()`.

Можно объявить, что параметр `mixFunc()` может принимать только функцию с указанной сигнатурой

```
Ingredient combineIngredients(
    Ingredient mixFunc(Ingredient, Ingredient),
    item1,
    item2) {
    print("mixing $item1 with $item2 using $mixFunc");
    return mixFunc(item1, item2);
}
```

В списке параметров указана функция с именем `mixFunc()` и два параметра типа `dynamic`

Параметр `mixFunc()` объявлен как функция, принимающая два параметра типа `Ingredient` и возвращающая значение типа `Ingredient`

2. Вызов `combineIngredients()`.

Модуль проверки типов может удостовериться, что передана функция с правильной сигнатурой

Возвращаемый тип Типы параметров функции

```
Ingredient mix(Ingredient item1, Ingredient item2) {
    return item1 + item2;
}

var sand = new Sand();
var gravel = new Gravel();

combineIngredients(mix, sand, gravel);
```

У этой функции `mix()` сигнатура допускает передачу в функцию `combineIngredients()`

```
Ingredient mixFunc(Ingredient, Ingredient),
```

Сигнатура этой функции `mix()` соответствует типу, поэтому ее можно передать в функцию `combineIngredients()`

Инструменты проверки типов могут удостовериться, что сигнатура функции `mixFunc()` отвечает ожиданиям

Рис. 4.10. Можно объявить, что в параметре разрешено передавать только функцию с указанной сигнатурой

Листинг 4.8. Использование `typedef` для объявления сигнатуры функции

```
typedef Ingredient MixFunc(Ingredient, Ingredient); ← typedef объявляет тип функции MixFunc

Ingredient combineIngredients(MixFunc mixFunc, item1, item2) { ←
    return mixFunc(item1, item2);
}                                     Тип MixFunc используется для задания строгого типа параметра
```

Благодаря `typedef` мы можем создать аббревиатуру для сигнатуры конкретной функции и использовать ее в объявлениях переменных и параметров. Это позволит модулю проверки типов убедиться, что

из одного места приложения в другое передаются объекты правильных типов.

Итак, мы познакомились с различными способами объявления функций в локальной области видимости другой функции. Теперь пришло время заняться замыканиями, то есть явлением, которое возникает, когда функция ссылается на переменную, объявленную вне ее собственной области видимости. Замыкание – мощный механизм функционального программирования.

Памятка

- Указывая только имя функции, без параметров в круглых скобках, мы получаем соответствующий ей объект-функцию.
 - Простые локальные функции, объявленные так же, как функции в области видимости верхнего уровня библиотеки, могут ссылаться на самих себя по имени; для них разрешено задавать полную информацию о типе параметров и возвращаемого значения.
 - У анонимных функций нет имени, они не могут рекурсивно вызывать себя и сообщать информацию о возвращаемом типе. Однако это удобная сокращенная запись для помещения функций в списки и для передачи их другим функциям в качестве параметров.
 - Вместо анонимной функции можно использовать именованную, тогда мы получаем возможность рекурсивного вызова и задания информации о типе, однако имя доступно только в области видимости самой функции.
 - Для объявления сигнатуры функции можно использовать ключевое слово `typedef`, тогда средство проверки типов будет иметь возможность проверять корректность использования объектов-функций.
-

4.3. Замыкания

Замыкания – особый способ использования функций. Передавая объект-функцию из одного места приложения в другое, программист часто создает замыкание, даже не подозревая об этом. Замыкания активно применяются в JavaScript с целью имитации различных конструкций, характерных для языков на основе классов: методов чтения и установки и закрытых свойств. Для этого создаются функции, единственное назначение которых – вернуть другую функцию. Однако в Dart для таких конструкций есть поддержка в самом языке, поэтому вряд ли замыкания понадобятся вам для этой цели при написании нового кода. Но вполне вероятно, что многие программы будут переноситься с JavaScript на Dart, поэтому концепция замыкания, схожая в Dart и в JavaScript, окажется полезной.

Объявленная функция не начинает исполняться немедленно, а хранится в переменной в виде объекта-функции – точно так же,

как значение типа `String` или `int`. И в теле функции могут использоваться переменные, объявленные ранее, как показано во фрагменте ниже:

```
main() {  
  var cement = new Cement();  ← Объявляется локальная переменная cement  
  
  mix(item1, item2) {          ← Объявляется локальная функция mix()  
    return cement + item1 + item2;  ← Переменную cement можно использовать  
  }                                даже внутри локальной функции  
}
```

Здесь мы создали функцию `mix()`, которая смешивает два ингредиента с цементом. Но вместо того чтобы каждый раз передавать переменную `cement` в отдельном аргументе, мы объявили ее предварительно и используем внутри функции.

Вызывая функцию `mix()` с параметрами `sand` и `gravel`, мы по-прежнему имеем доступ к переменной `cement`, которая была объявлена вне области видимости функции `mix()`. Эту функцию можно затем передать в `combineIngredients()`, и та радостно смешивает остальные два ингредиента с `cement`, даже не зная о существовании `cement`. На рис. 4.11 показано, как это происходит.

Это и есть замыкание – функция, сохраняющая ссылки на переменные, которые были ей видны в момент объявления.

Откуда взялось название «замыкание»?

Термин *замыкание* (closure) происходит от слова *close-over* (накрывать). Можно считать, что функция «накрывает» все нелокальные переменные, которые находились в ее области видимости в точке объявления.

Как мы только что видели, один из способов использования замыкания – скрыть некоторые детали реализации (в данном случае переменную `cement`) от функции, которая пользуется замыканием.

Замыкание образуется и тогда, когда одна функция возвращает другую. Допустим, что имеется функция `getShovel()`, которая возвращает лопату. Эту лопату можно использовать для перемешивания ингредиентов, но, как показано в листинге 4.9, на лопате налипла грязь (`stickyMud`). Когда функция `getShovel()` возвращает функцию `mix()`, та сохраняет ссылку на переменную `stickyMud`, и налипшая грязь смешивается с ингредиентами, несмотря на то что функция `getShovel()` давно завершилась.

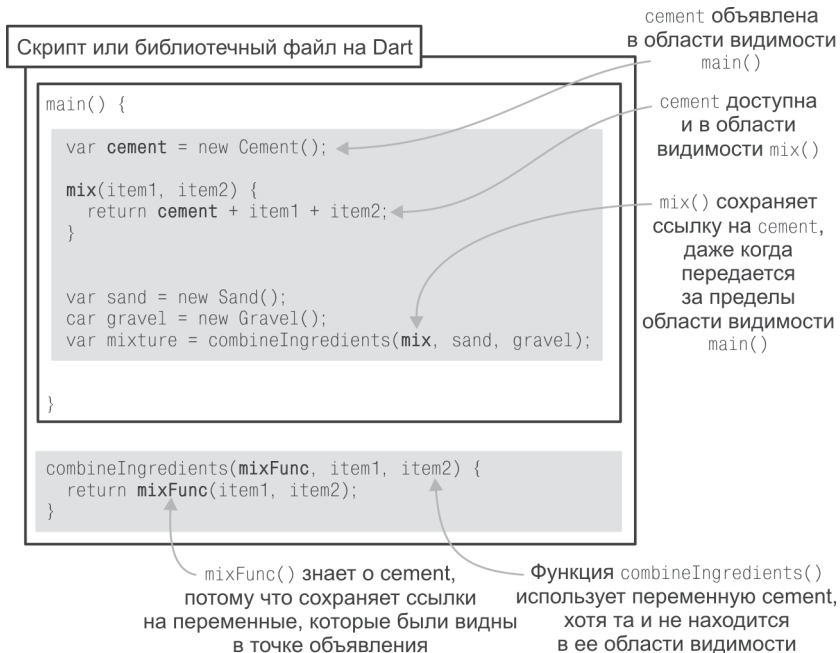


Рис. 4.11. Функция `mix()` сохраняет ссылку на переменную `cement` даже тогда, когда вызывается вне области видимости функции `main()`, где эта переменная объявлена

Листинг 4.9. Создание замыкания с помощью возврата функции

```

getShovel() {
    var stickyMud = new Mud(); ← Создается грязь на лопате

    var mix = (item1, item2) {
        return stickyMud + item1 + item2;
    }

    return mix; ← Возвращается mix()
}

main() {
    var mixFunc = getShovel(); ← Вызывается функция getShovel(), которая возвращает
                               ← функцию mix(), сохранившую ссылку на stickyMud

    var sand = new Sand();
    var cement = new Cement();
}

```

Объявляется функция `mix()`, в которой переменная `stickyMud` используется

```
var muddyMortar = mixFunc(sand, cement);  
}
```

Благодаря функции `mix()` в смеси
← оказываются не только цемент с песком,
но и еще кое-что

Замыкания могут возникать непреднамеренно, потому что переменные в области видимости объемлющей функции – вещь совершенно естественная. Всякий раз, возвращая одну функцию из другой или передавая ее в качестве параметра, вы имеете дело с замыканием.

Памятка

- Функция, в которой используются переменные, объявленные вне ее собственной области видимости, имеет все шансы стать замыканием.
 - Функция становится замыканием, когда передается за пределы области видимости, в которой объявлена, – в виде параметра вызываемой функции либо в виде возвращаемого значения.
-

4.4. Резюме

В этой главе мы рассмотрели, как объявляются функции – в короткой и длинной форме. Функция, записанная в короткой форме, неявно возвращает значение выражения в единственной строке ее тела. Если же используется длинная форма, то для возврата значения нужно явно включить ключевое слово `return`.

В Dart любая функция возвращает какое-то значение – `null`, если не было указано ничего другого, – однако с помощью ключевого слова `void` можно сообщить инструментальным средствам о том, что возвращать значение не предполагалось.

Функцию можно сохранять в переменной, указывая имя функции без круглых скобок. При этом мы получаем объект-функцию, который можно передавать из одного места программы в другое, как любую переменную. Объект-функцию можно вернуть из функции или передать в другую функцию, где его можно вызвать как обычную функцию. Объект-функция *является* экземпляром класса `Function`.

Чтобы объявить тип переменной или параметра, в котором хранится объект-функция, воспользуйтесь ключевым словом `typedef` для определения именованной сигнатуры функции (`typedef` может встречаться только в области видимости верхнего уровня библиотек). Это даст инструментам возможность проверить правильность вашего кода. Имя сигнатуры можно использовать так же, как любой другой тип.

Мы также рассмотрели замыкания, которые создаются, когда в теле функции используются переменные, объявленные в объемлющей области видимости, и эта функция передается в другую часть программы. Замыкания можно использовать для сокрытия деталей реализации, о которых принимающая функция не должна знать.

Разобравшись с функциями, мы в следующей главе познакомимся с механизмами библиотек и ограничения доступа в Dart. Это важно, потому что имена библиотечных функций и классов самым непосредственным образом отражаются на их доступности. Обе концепции тесно взаимосвязаны, и в этом вопросе следует навести ясность, до того как переходить к классам и интерфейсам.



Глава 5. Библиотеки и ограничение доступа

В этой главе

- ❑ Организация кода в виде набора повторно используемых пакетов.
- ❑ Импорт библиотек в приложение.
- ❑ Соккрытие функциональности за счет ограничения доступа к библиотеке.
- ❑ Использование нескольких исходных файлов.
- ❑ Скрипты как запускаемые библиотеки.

В большинстве языков программирования имеется тот или иной способ организации библиотек, позволяющий разбивать код на несколько файлов и предоставлять его в распоряжение других разработчиков. Это способствует модульности и повторной использованию кода и позволяет нескольким разработчикам трудиться над разными частями программы за счет сокрытия внутренней реализации библиотеки. Dart в этом смысле не исключение, однако в Dart механизм библиотек еще и тесно связан с ограничением доступа – прежде всего к классам и функциям. В Java и C# ограничение доступа реализуется на уровне класса, а в Dart – на уровне библиотеки. Потому-то мы и обсуждаем библиотеки и ограничение доступа так рано.

В этой главе вы научитесь создавать и использовать библиотеки в Dart и узнаете, как они связаны с моделью ограничения доступа, которая позволяет скрывать внутренние детали реализации библиотеки. Библиотека – это наименьшая единица развертывания, в ней может находиться всего один класс или функция, а может и целое приложение. На практике все приложения, кроме совсем уж тривиальных, разбиваются на несколько библиотек, потому что при этом обеспечиваются хорошая, слабо связанная архитектура, возможность повторного использования и тестопригодность. По ходу этой главы мы построим простую библиотеку протоколирования, которую вы сможете импортировать в собственные программы.

При создании повторно используемого кода часто бывает желательно скрыть детали реализации, к которым пользователи не должны обращаться в обход опубликованного и согласованного интерфейса, например внутреннее состояние класса. В Dart есть возможность опубликовать библиотеку для коллег или просто в веб, сделав видимой конечным пользователям лишь некоторую ее часть. При таком подходе внутреннюю реализацию библиотеки можно изменять, не затрагивая интересов пользователей. Эта модель отличается от принятой в языке Java и C#, где ограничение доступа реализовано на уровне класса.

Почему Dart отошел от модели ограничения доступа на уровне класса?

Это одна из частей Dart, на которую JavaScript и разработка для веб вообще оказали особенно сильное влияние. В JavaScript нет понятия закрытости, если не считать некоторых соглашений, в том числе возврата замыканий из функций. Поэтому модель ограничения доступа в Dart следует считать шагом вперед, по сравнению JavaScript, а не сравнивать с более традиционными языками на основе классов, в частности Java и C#.

Факультативная типизация в Dart позволяет включать в места, где пользователи взаимодействуют с библиотекой, документацию в виде информации о типах параметров и возвращаемых значений функций или о свойствах классов, не требуя строгой типизации повсеместно. Как уже отмечалось ранее, работа приложения не зависит от того, есть информация о типах или нет, но ее наличие помогает другим разработчикам и инструментальным средствам.

В главе 3 мы уже видели, как импортируется встроенная библиотека `dart:html` с помощью ключевого слова `import`, и, как скоро выяснится, импортировать собственную библиотеку ничуть не сложнее.

5.1. Определение и импорт библиотеки

В этом разделе мы создадим библиотеку протоколирования, которую назовем `loglib`. Это будет простое средство вывода сообщений различного уровня – отладочных, информационных и предупредительных – на консоль браузера. Такого рода библиотеки существуют для большинства языков, например `nLog` для .NET и `log4j` для Java. В Dart для протоколирования проще всего использовать функцию `print()`, вокруг нее и строится библиотека `loglib`.

Для экспериментов с библиотекой необходим какой-нибудь код, который будет ее вызывать, например приложение `PackList` из главы 3, в котором есть только функция `main()`. Поскольку в `PackList` уже имеется простая функциональность, допускающая протоколирование, то это идеальное приложение для тестирования нашей библиотеки `loglib`. В нем есть следующие места, куда можно с пользой включить протоколирование:

- ❑ конструирование ГИП (отладочный уровень);
- ❑ добавление предмета (информационный уровень);
- ❑ добавление предмета без названия (предупредительный уровень);
- ❑ смена состояния предмета – упакован или не упакован (информационный уровень).

Этот план подсказывает набор функций, который должна предоставлять библиотека; они показаны на рис. 5.1.

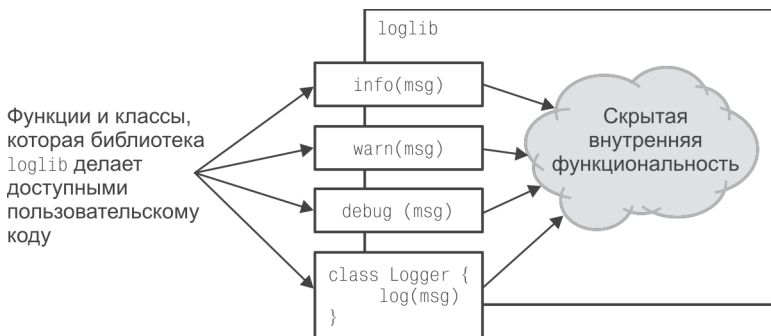


Рис. 5.1. Функции и классы библиотеки `loglib`, доступные внешней программе

Библиотека `loglib` будет содержать три функции верхнего уровня – `info()`, `warn()`, `debug()` – и один класс `Logger`. Функцией *верхнего уровня* называется функция, существующая вне класса (например, функций `main()`). Библиотека может содержать как функции, так и классы, причем функция необязательно должна быть членом какого-то класса (как в C# или Java). Никто не запрещает создавать библиотеки, состоящие из одних лишь функций. И наоборот – можно создать библиотеку, содержащую только классы.

Код разбивается на библиотеки, для того чтобы предоставить сторонней программе упакованную и абстрагированную функциональ-

ность. Используя существующую библиотеку, например `dart:html`, вы не знаете, *как* она создает HTML-элементы и генерирует события браузера, – важно лишь, что она это *делает*. Разумная организация кода в виде библиотек позволяет упаковывать их как единое целое и создавать новые версии, предлагая опубликованный интерфейс к функциональности библиотеки.

5.1.1. Определение библиотеки с помощью ключевого слова *library*

Библиотека в Dart представляет собой файл с расширением `.dart`, в начале которого находится объявление `library` имя_библиотеки;. Объявление `library` означает, что это корневой файл библиотеки, оно должно предшествовать всем объявлениям функций и классов (рис. 5.2).

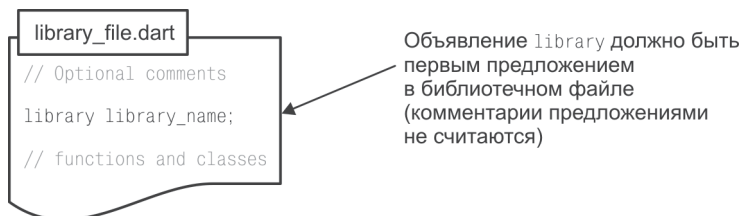


Рис. 5.2. `library` – первое предложение в библиотеке

Смысл именования библиотеки – в том, чтобы идентифицировать повторно используемый блок кода. Имя библиотеки должно быть записано строчными буквами, слова в нем разделяются знаками подчеркивания. Имя библиотеки не обязано совпадать с именем файла, хотя по соглашению имена файлов с Dart-кодом образуются точно так же: строчными буквами и с подчеркиками. В отличие от Java, в Dart не существует жестких связей между именем файла и структурой папки. На рис. 5.3 показаны допустимые и недопустимые имена библиотек.

Библиотеку можно не только называть как угодно, но и помещать в любую папку; ограничений на количество библиотек в одной папке тоже нет. Чтобы определить библиотеку `loglib`, нам понадобится один файл `loglib.dart`, содержащий такую строку:

```
library loglib; ← Эта строка определяет, что файл является библиотекой
```



Рис. 5.3. Допустимые и недопустимые имена библиотек

Редактор Dart без труда идентифицирует библиотечные .dart-файлы (отличая их от небиблотечных файлов) и показывает их имена в окне файлов полужирным шрифтом (рис. 5.4).

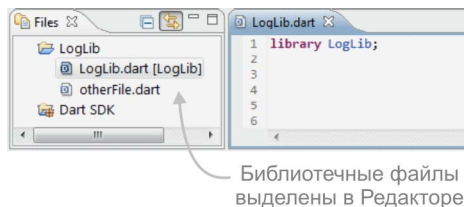


Рис. 5.4. Редактор Dart выделяет имена библиотек полужирным шрифтом

Определив библиотечный файл, мы можем приступить к написанию кода. Пока что функции протоколирования верхнего уровня будут вызывать встроенную функцию `print()`, которая выведет сообщение на консоль браузера, пользуясь короткой нотацией, которую мы рассматривали в главе 4. В классе `Logger` будет функция, которая делает то же самое (листинг 5.1).

Листинг 5.1. Функции и классы в файле loglib.dart

```
library loglib;
```

```
debug(msg) => print("DEBUG: $msg");  
warn(msg) => print("WARN: $msg");  
info(msg) => print("INFO: $msg");
```

Функции верхнего уровня
(не обязательно являющиеся
членами класса)

```
class Logger {  
  log(msg) => print("LOG: $msg");  
}
```

Класс Logger
содержит
функцию log()

Сейчас библиотека loglib не дает практически ничего сверх встроенной функции print(), но постепенно мы изменим это положение.

Совет. Встроенная функция print() отправляет сообщение на stdout (если работает в составе серверного скрипта) или на отладочную консоль браузера (доступна из меню **Инструменты** ⇒ **Консоль JavaScript** в Chrome/Dartium или **Инструменты** ⇒ **Веб-разработка** ⇒ **Веб-консоль** в Firefox).

Теперь, когда у нас есть библиотека с работоспособными функциями и классами, пора применить их в деле. Воспользуемся нашей библиотекой из стороннего приложения.

5.1.2. Импорт библиотек

Для импортирования библиотеки loglib нужно воспользоваться предложением import "loglib.dart"; при условии что файл loglib.dart находится в той же папке, что и приложение. Предложение import – мощное средство, несмотря на обманчивую простоту: оно может ссылаться на библиотеку по URI-адресу в самой файловой системе или в веб (по протоколу HTTP). Все показанные ниже предложения import допустимы:

```
import "../libs/loglib/loglib.dart";  
import "http://www.mysite.com/loglib.dart";  
import "file:///c:/loglib/loglib.dart";  
import "package:/loglib/loglib.dart";
```

Предупреждение. Задание абсолютного имени файла с указанием конкретной папки на конкретном диске противоречит идее повторного использования; это означает, что разработчик, который пожелает вос-


```
main() {  
  // todo: вывести отладочное сообщение  
  // ...остальная часть файла опущена...
```

Импорт одной библиотеки в другую

Библиотека также может импортировать другие библиотеки. В библиотеку `loglib` можно было бы при необходимости импортировать сколько угодно библиотек, лишь бы все предложения `import` располагались между объявлением `library` и самим кодом, как показано на рис. 5.6.

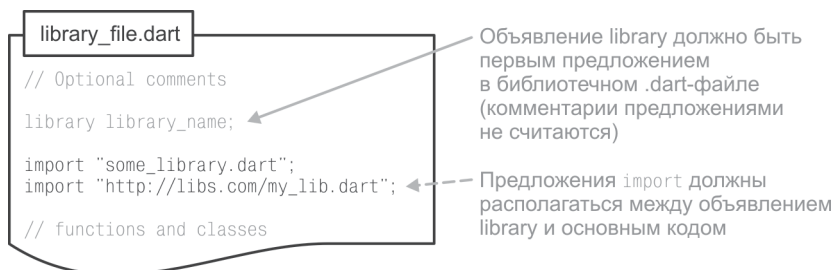


Рис. 5.6. Предложение `import` должно предшествовать основному коду

Если бы библиотека `loglib` импортировала библиотеку `dart:html`, то последняя была бы доступна только библиотеке `loglib`. Чтобы воспользоваться `dart:html`, приложению пришлось бы еще раз включить предложение `import` — уже в своем коде. Импортированные файлы локальны для той библиотеки, где встречаются предложения `import`.

Допускаются также циклические ссылки между импортируемыми библиотеками. Например, библиотека А может импортировать библиотеку В, а библиотека В — импортировать библиотеку А. Так как в Dart библиотеки целиком загружаются до начала работы приложения, то компилятор может обнаружить и разрешить такие зависимости.

Использование импортированных функций верхнего уровня

Первое, что мы хотели сделать с помощью библиотеки `loglib`, — добавить вывод отладочного сообщения перед конструированием пользовательского интерфейса. Лучше всего поместить этот код в начало функции `main()`, там, где оставлен комментарий `todo` (рис. 5.7). При

запуске PackList на консоли браузера будет напечатано сообщение. На рис. 5.7 показана связь между вызовом метода `debug()` из приложения PackList и библиотекой `loglib`.

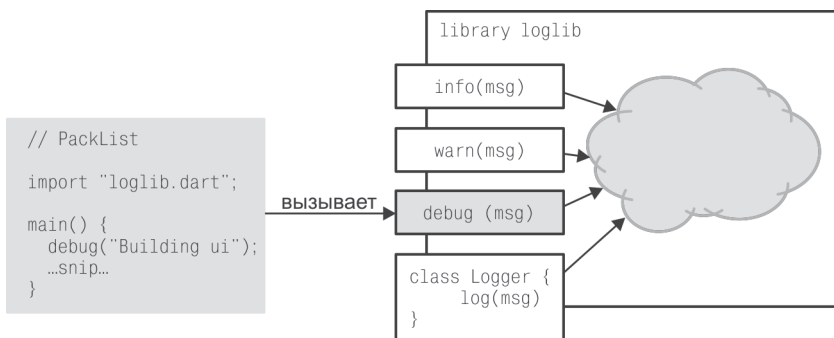


Рис. 5.7. PackList вызывает функцию верхнего уровня из библиотеки `loglib`

Dart спроектирован так, чтобы его можно транслировать на JavaScript для исполнения в браузерах, не поддерживающих Dart непосредственно. В JavaScript понятия библиотеки нет, поэтому Dart-приложение с несколькими библиотеками после конвертации становится одним JavaScript-файлом, в котором каждая импортированная библиотека занимает отдельную секцию, помеченную комментарием (см. листинг 5.3). Отметим, что в результирующем JavaScript-файле в секции `loglib` находится только функция `debug(msg)`. Поскольку никакими другими функциями или классами мы не пользовались, то конвертор их и не включил.

Листинг 5.3. JavaScript-файл, сгенерированный из приложения PackList и библиотеки `loglib`

```

...опущено...
// ***** Library loglib *****
// ***** Code for top level *****
function debug(msg) {
  return print$(("DEBUG: " + msg));
}
// ***** Library C:\DartInAction\PackList *****
// ***** Code for PackItem *****
// ***** Code for top level *****

```

В секции `loglib` присутствует только функция `debug()`

```
function main() {
  debug("Starting building UI");
  ...опущено...
```

Но и она включена лишь потому,
что вызывается из приложения

Теперь можно продолжить реализацию намеченного плана в приложении PackList. Осталось запротоколировать следующие действия:

- ❑ добавление предмета (информационный уровень);
- ❑ добавление предмета без названия (предупредительный уровень);
- ❑ смена состояния предмета – упакован или не упакован (информационный уровень).

Для этого нужно добавить в класс PackItem показанные ниже предложения. Во-первых, мы проверяем длину itemText в конструкторе и выводим сообщение функцией info или warning; во-вторых, добавляем еще один обработчик события, который вызывает функцию info при любом щелчке по предмету в пользовательском интерфейсе.

Листинг 5.4. Добавление вызовов функций протоколирования в приложение PackList

```
class PackItem {
  // ...опущено...

  PackItem(this.itemText) {
    if (itemText.length == 0) {
      warn("Пользователь добавил предмет без названия");
    }
    else {
      info("Пользователь добавил предмет $itemText");
    }
  }

  DivElement get uiElement {
    if (_uiElement == null) {
      _uiElement = new Element.tag("div");
      _uiElement.classes.add("item");
      _uiElement.text = this.itemText;
      _uiElement.on.click.add( (event) => isPacked = !isPacked);
      _uiElement.on.click.add( (event) => info("Предмет изменен");
    }
    return _uiElement;
  }
  // ...опущено...
}
```

Добавляется
протоколирование
в конструктор PackItem

Существующий
обработчик

← события

Добавляется второй обработчик события, который
выводит в протокол сообщение «Предмет изменен»

Использование импортированных классов верхнего уровня

Сейчас наше приложение PackList пользуется функциями из открытого интерфейса библиотеки. Но в библиотеке есть еще класс `Logger`, про который мы забыли. Добавим еще одно сообщение – по завершении конструирования интерфейса, но на этот раз воспользуемся классом `Logger` (просто для примера). Создаем экземпляр этого класса с помощью ключевого слова `new`, будто этот класс объявлен прямо в файле `PackList`.

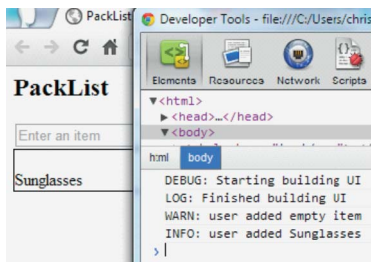
Листинг 5.5. В файле `packlist.dart` используется класс `Logger`, импортированный из `loglib`

```
import "../loglib/loglib.dart";

main() {
  debug("Начинается конструирование ГИП");
  // ...код конструирования ГИП опущен
  var logger = new Logger();
  logger.log("Закончено конструирование ГИП");
  // ...опущено...
}
```

Создается новый экземпляр класса `Logger`,
 ← импортированного из `loglib`
 ← Вызывается метод `log()` объекта `Logger`

Итак, мы подключили библиотеку `loglib` к приложению `PackList`. Теперь оно выводит различные сообщения в начале работы и после действий пользователя. На рис. 5.8 показаны несколько таких сообщений на консоли разработчика.



Сообщения, выведенные функциями протоколирования из библиотеки `loglib`, отображаются на консоли разработчика

Рис. 5.8. Сообщения, выведенные приложением `PackList`

Использование префикса библиотеки для предотвращения конфликта имен

Теперь мы можем предоставить работающую библиотеку в распоряжение других разработчиков, опубликовав ее интерфейс. К сожалению, ничто не мешает программисту использовать также другую библиотеку, в которой тоже есть класс `Logger` и функция верхнего уровня `info()`. Тут-то и приходят на выручку префиксы импорта.

Префикс в предложении `import` позволяет импортировать несколько библиотек, содержащих классы и функции верхнего уровня с одинаковыми именами. Представьте, что разработчик хочет воспользоваться гипотетической библиотекой `WebLogger`, которая умеет посылать сообщения расположенному где-то в сети серверу и тоже включает функцию `info()`.

Dart не будет знать, какую функцию `info()` вы имели в виду (рис. 5.9).

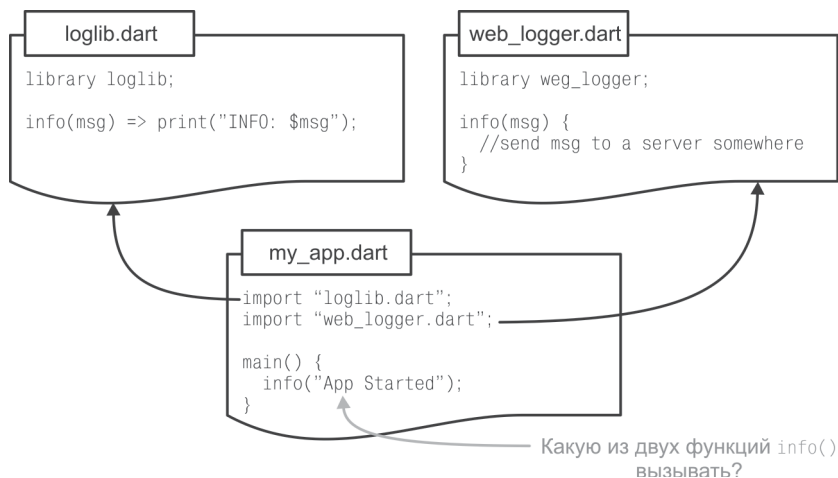


Рис. 5.9. Иногда разные импортированные библиотеки содержат функции с одинаковыми именами, поэтому необходим какой-то механизм разрешения конфликтов

По счастью, в Dart имеется ключевое слово `as`, являющееся частью предложения `import`. Благодаря ему удастся разрешать конфликты имен. За словом `as` должен следовать префикс, который становится частью ссылки на имя из данной библиотеки (см. рис. 5.10).



Рис. 5.10. Можно при импорте библиотеки объявить префикс, с которым программа будет обращаться к библиотечным функциям

Если при импорте некоторой библиотеки префикс определен, то его необходимо указывать при обращении к любой функции или классу из этой библиотеки. В принципе, префикс можно указывать во всех предложениях `import`, но, поступая так, вы загромождаете свой код, так как при обращении ко всем функциям и классам приходится писать префиксы. Лучше применять прагматический подход: добавлять префиксы, только когда это улучшает восприятие кода или предотвращает конфликты имен.

Сейчас библиотека протоколирования `loglib` предоставляет абсолютно все свои функции и классы пользователям. От приложения, импортирующего библиотеку, ничто не скрыто – вся функциональность опубликована. В следующем разделе мы сделаем некоторые элементы закрытыми, запретив к ним доступ извне библиотеки.

Памятка

- Предложение `library имя_библиотеки;` должно быть первым в библиотеке.
 - При импорте библиотеки можно также указывать путь: `import "uri/to/lib.dart";`.
 - Предложения `library` и `import` должны предшествовать всему остальному коду.
 - Для предотвращения конфликтов имен из разных библиотек можно использовать префиксы импорта.
-

5.2. Соккрытие функциональности путем ограничения доступа к частям библиотеки

В любой библиотеке могут существовать внутренние детали реализации, которые автор не хотел бы раскрывать конечным пользователям. Сейчас в библиотеке `Logger` имеются лишь очень простые функции вывода сообщений на консоль браузера. Но предположим, что понадобилось добавить еще одну возможность – отправлять сообщения удаленному серверу. Мы не хотим, чтобы пользователи вызывали код протоколирования на сервер напрямую; он должен вызываться внутри библиотеки из функций, составляющих ее открытый интерфейс. Если просто взять и объявить в библиотеке классы и функции, то они будут доступны любому пользователю. Но на наше счастье, Dart считает закрытыми элементы, имена которых начинаются со знака подчеркивания (`_`).

Примечание. В этой главе слово *закрытый* (`private`) употребляется в смысле, принятом в ООП, то есть относится к языковой конструкции, позволяющей скрывать внутреннее устройство одной системы от другой. Закрытие кода библиотеки от любопытных глаз – совсем другая задача, которую можно решить путем запутывания и минимизации кода; средства для этого сейчас разрабатываются командой Dart. Кроме того, закрытость (или ограничение доступа) никоим образом не связана с безопасностью – весь импортированный библиотечный код работает на тех же правах, что и главное приложение. В главе 15 мы увидим, как создаются изоляторы, которые в какой-то мере обеспечивают безопасность.

Как видно по рис. 5.11, ограничение доступа реализовано на уровне библиотеки, а не класса. Все объявленное закрытым доступно только внутри самой библиотеки, обратиться к нему по имени извне невозможно.

Закрытыми можно делать функции и классы верхнего уровня, а в классах – свойства и методы. Для этого достаточно добавить подчеркик в начало имени. При обращении к таким закрытым элементам подчеркик также необходим, как в примере ниже:

```
info(msg) {  
  print("INFO: $msg");  
  _logToServer("INFO",msg);  ← Из функции info библиотеки loglib  
}  
                             вызывается закрытая функция _logToServer
```

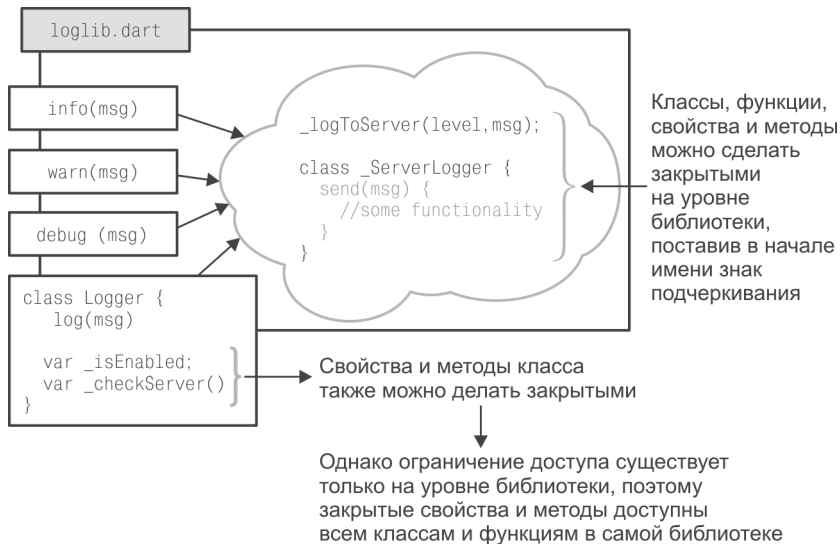


Рис. 5.11. Для ограничения доступа к классу, функции, свойству или методу нужно поставить в начале имени знак подчеркивания, тогда доступ к этому элементу будет возможен только изнутри самой библиотеки

Стоит ли создавать языковой механизм на базе соглашения об именовании?

Знак подчеркивания в начале имени – широко употребительное (хотя и не общепринятое) соглашение для обозначения закрытости, особенно в языках, где встроенного понятия закрытости нет (как в JavaScript и Python). Dart сделал следующий шаг в этом направлении, превратив соглашение в часть языка.

Это решение вызвало оживленные дебаты в сообществе Dart и, пожалуй, стало одним из самых серьезных разногласий. С одной стороны, для обеспечения закрытости от программиста не требуется почти никаких усилий, и читателю кода сразу видно, что именно закрыто, – не нужно искать объявление в другом месте. С другой стороны, это негативно отражается на легкости восприятия кода. Что вы скажете о таком предложении:

```
var someValue = new MyClass()._getValue()._doAction(_withProperty);
```

Есть и еще одно возражение против подчёрка – если мы захотим сделать открытый элемент закрытым (или наоборот), то придется переименовывать все его вхождения. У этого аргумента есть и другая сторона: если мы переименовываем закрытый элемент, делая его открытым, то изменение не коснется ничего, кроме самой библиотеки (так как раньше пользователи не могли обращаться к закрытому элементу). Если же, наоборот, открытый элемент делается закрытым, то возникают куда более серьезные

проблемы, чем переименование (например, если таким образом станет недоступной некая функция, то все пользовательские программы, в которых она использовалась, перестанут работать).

Следует запомнить два правила:

- ❑ код, находящийся в библиотеке, может без ограничений обращаться к любому коду в той же библиотеке;
- ❑ код, находящийся вне библиотеки, может обращаться только к открытому коду в этой библиотеке.

Особенно эти правила важны для классов, в которых есть как открытые, так и закрытые свойства и методы.

5.2.1. Ограничение доступа в классах

Ограничение доступа к членам класса в Dart работает иначе, чем в C# и Java. Первое из сформулированных выше правил означает, что два разных класса имеют доступ к закрытым свойствам и методам друг друга (по аналогии с закрытостью в пакете в Java).

В библиотеке `loglib` у нас сейчас есть класс `Logger`. Допустим, требуется знать, включено или выключено протоколирование; заведем для этого внутреннее свойство `_isEnabled`. Другие классы, находящиеся в той же библиотеке, что и `Logger`, могут обращаться к этому свойству напрямую, но пользователям библиотеки оно недоступно. Прочие части приложения не должны ничего знать о внутреннем устройстве класса `Logger`, им достаточно того, что он *работает*. Эта связь показана на рис. 5.12.

Подчерк в начале имени позволяет включить в библиотеку весьма развитую функциональность, гарантируя в то же время, что пользователям она будет доступна только через четко определенный и согласованный интерфейс классов, методов и функций верхнего уровня.

Доступ к закрытым полям с помощью методов чтения и установки

Как было сказано в главе 3, методы чтения и установки позволяют обращаться к полям. Их тоже можно сделать открытыми или закрыть, добавив подчерк в начало имени. Если вы хотите, чтобы внешние пользователи могли только читать свойство `_isEnabled`, включите в класс открытый метод чтения. Если включить еще и открытый метод установки, то свойство можно будет изменять.

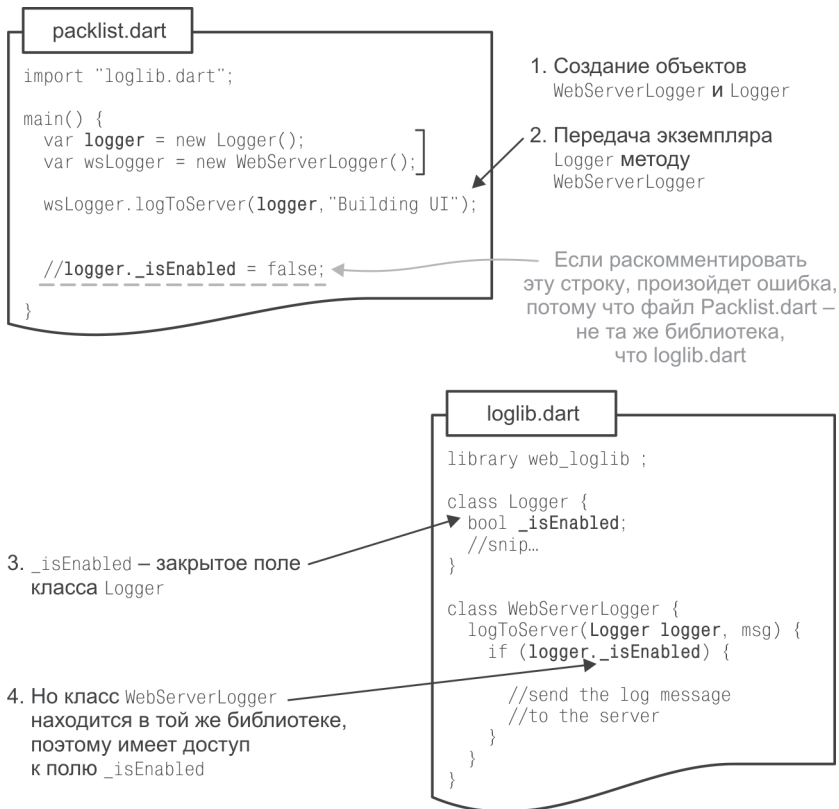


Рис. 5.12. Класс `WebServerLogger` имеет доступ к закрытым свойствам класса `Logger`, потому что они находятся в одной и той же библиотеке

Интересно отметить, что совершенно законно определять свойства, допускающие только чтение или только запись. На рис. 5.13 показано, как библиотека может показывать или скрывать свойства класса с помощью методов чтения и установки.

Использование закрытых методов

Помимо закрытых полей, обернутых методами чтения и установки, в классе можно определить также закрытые методы – добавив в начало имени знак подчеркивания. Это полезно, например, потому, что позволяет перерабатывать длинные методы, разбивая их на более короткие закрытые, к которым нет доступа извне библиотеки.

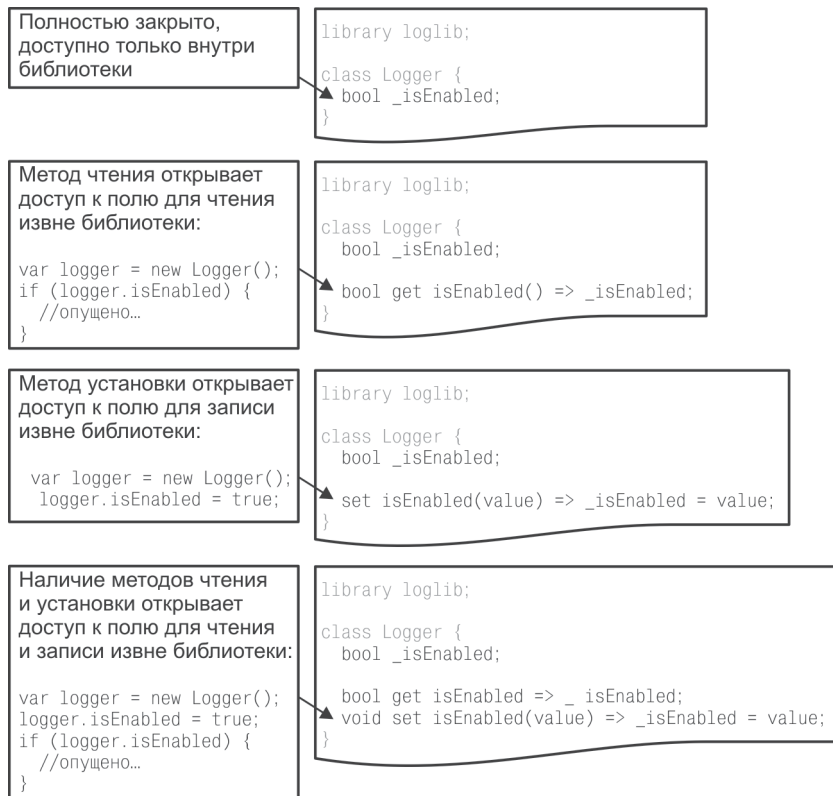


Рис. 5.13. Применение методов чтения и установки для предоставления внешним пользователям различного уровня доступа к полям класса

На рис. 5.14 показана типичная операция: блок кода выносится из длинного метода в новый закрытый метод `_sendToServer()`. Метод `_sendToServer()` невозможно вызвать извне библиотеки, но исходный метод `log()` работает, как и раньше, а внешние пользователи даже не подозревают об изменении.

Закрытые классы

Можно создавать не только закрытые свойства и методы классов, но и закрытые классы – естественно, начав имя с подчеркика. Таковым является класс `_ServerLogger` в листинге 5.6. Закрытые классы полезны тем, что их можно создать только внутри библиотеки. Извне

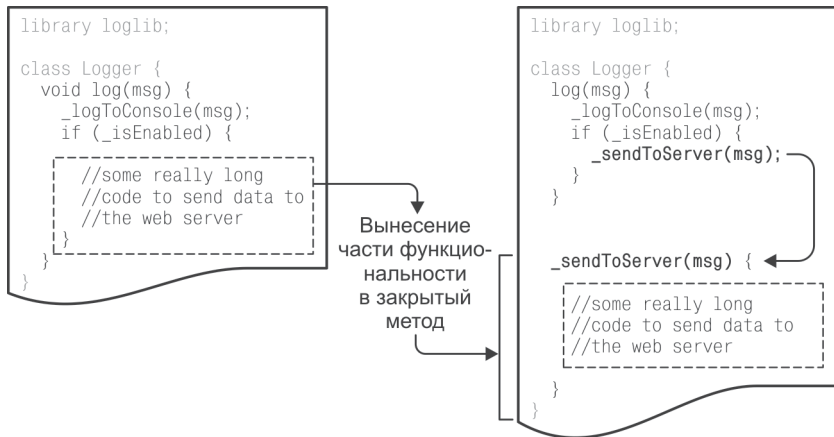


Рис. 5.14. Чтобы сделать код более удобным для чтения и сопровождения, можно вынести блок в отдельный закрытый метод того же класса, недоступный извне библиотеки

создать экземпляр закрытого класса с помощью ключевого слова `new` невозможно.

У закрытых классов могут быть открытые методы и свойства. В классе `_ServerLogger` имеются закрытое свойство `_serverIp` и открытое свойство `serverName`.

Интересный вопрос: зачем закрытому классу (который доступен только внутри библиотеки) открытые методы и свойства? Пока мы находимся внутри одной библиотеки, проводить различие между открытыми и закрытыми членами класса бессмысленно, а если класс закрыт, то как к нему обратиться извне? В листинге ниже показано, как такое может случиться: функция `getServerLogger()` возвращает экземпляр закрытого класса `_ServerLogger`.

Листинг 5.6. Библиотека `mixed_loglib.dart` содержит открытые и закрытые классы

```
library mixed_loglib;
```

Класс `Logger` открыт, на него можно ссылаться

```
class Logger {
```

← напрямую из внешней программы

```
  _ServerLogger getServerLogger() {
    return new _ServerLogger();
  }
```

Метод открытого класса `Logger` вправе вернуть экземпляр закрытого класса `_ServerLogger`, который вызывающая программа может использовать, но не может создать

```
}
```

<pre>class _ServerLogger { var serverName; var _serverIp; }</pre>	Закрытый класс <code>_ServerLogger</code> содержит открытое и закрытое свойства; для самой библиотеки это безразлично, так как класс в целом закрыт

Хотя напрямую обращаться к закрытым классам извне библиотеки невозможно, открытый библиотечный метод или функция вправе вернуть экземпляр закрытого класса. Вообще-то, так поступать не стоит, но благодаря факультативной типизации Dart способен обработать этот случай.

Совет. Допустимо возвращать из библиотеки открытый класс, но к такому классу обычно обращаются через его открытый неявный интерфейс, а не по имени. К этой идее я еще вернусь в следующей главе.

Вызывающая программа видит класс `Logger` (и может создать его экземпляр), но ничего не знает о закрытом классе `_ServerLogger`. Она может вызвать функцию `getServerLogger()` и использовать полученный от нее экземпляр `_ServerLogger` при условии, что не ссылается на него по имени, то есть сохраняет объект в нетипизированной переменной, как показано в следующем примере:

<pre>Logger logger = new Logger(); var privateInstance = logger.getServerLogger();</pre>	Результат разрешается сохранить ← только в переменной динамического типа

Хотя обращаться к классу `_ServerLogger` по имени нельзя, но коль скоро у вас уже есть экземпляр этого класса, то можно обращаться к его открытым свойствам, и инструменты слова поперек не скажут. Однако автозавершения вы не получите, так как не можете предоставить инструментам информацию о типе. Попытавшись получить доступ к свойству `privateInstance._serverIp`, вы получите ошибку `noSuchMethod`, потому что обращаетесь к закрытому свойству извне библиотеки. Но доступ к свойству `privateInstance.serverName` возможен, потому что оно не является закрытым. Написание библиотек, которые предполагается использовать именно таким образом, следует расценивать как порочную практику, если только она не дополняется открытыми интерфейсами, потому что у пользователя нет никакого способа узнать, как работать с закрытыми классами (помимо изучения исходного кода).

5.2.2. Использование закрытых функций в библиотеках

Функции на верхнем уровне библиотеки, как и классы, могут быть открытыми и закрытыми. Наличие подчеркива в начале имени функции делает ее закрытой, то есть обратиться к ней извне библиотеки невозможно. Это бывает полезно, когда нужна закрытая служебная функция без ассоциированных данных о состоянии, то есть делать ее методом класса нет смысла.

В листинге ниже показана закрытая функция, которая вызывает-ся открытыми функциями `info()`, `warn()` и `debug()`, а также из класса `Logger`.

Листинг 5.7. Закрытая функция в библиотеке `loglib.dart`

```
library loglib;

_logMsg(msg) { ← Новая закрытая функция _logMsg доступна только внутри библиотеки
  _ServerLogger serverLogger = new _ServerLogger(); ← Каждый раз для вывода
  serverLogger.send(msg);                             сообщения создается новый
}                                                       экземпляр класса (только
                                                       для примера)

info(msg) => _logMsg("INFO $msg"); | Открытые функции
warn(msg) => _logMsg("DEBUG $msg"); | теперь вызывают
debug(msg) => _logMsg("WARN $msg"); | _logMsg

class _ServerLogger {
  // ...опущено...
}

class Logger {
  log(msg) => _logMsg(msg); ← Класс Logger теперь тоже вызывает _logMsg
}
```

Закрытая функция верхнего уровня `_logMsg()` доступна только внутри библиотеки `loglib` – внешние пользователи не могут к ней обратиться. В примере выше при каждом вызове создается экземпляр закрытого класса `_ServerLogger`, но это сделано только для примера – вполне хватило бы и простой функции `print()`.

Памятка

- Соглашение о префиксе `_`, обозначающем закрытость, распространяется на функции, классы, методы и свойства.

- Закрытый код доступен только внутри библиотеки, где он находится.
 - Код, не помеченный как закрытый, доступен внешним пользователям библиотеки.
-

Создание повторно используемых библиотек со скрытой внутренней функциональностью – стандартная практика разработки многих приложений, и в Dart этот механизм поддержан за счет превращения соглашения о знаке подчеркивания в языковое средство.

Итак, теперь вы знаете, как разбить приложение на библиотеки, допускающие повторное использование. Однако сама библиотека может состоять из многих тысяч строк кода. Держать весь этот код в одном файле нерационально. К счастью, в Dart имеется способ разбить библиотеку на еще более мелкие файлы с исходным кодом.

5.3. Организация исходного кода библиотеки

Теперь библиотека `loglib` содержит открытые и закрытые классы и функции. Если продолжить добавление функциональности, то очень скоро в библиотеке станет трудно ориентироваться, даже с помощью инструментов. Но еще хуже то, что при коллективной разработке любой крупный рефакторинг файла может вызвать сложности у членов команды, работающих над одной и той же библиотекой.

На наше счастье, Dart позволяет разбить библиотеку на несколько исходных файлов. Внешние пользователи об этом не знают – им безразлично, состоит библиотека из одного или сотни файлов, каждый из которых содержит один или несколько классов или функций.

В этом разделе мы разобьем файл `loglib.dart`, который сейчас содержит два класса и четыре функции (рис. 5.15), на несколько исходных файлов.

Мы поместим классы и функции в два разных файла, а для их связывания воспользуемся файлом библиотеки `loglib.dart`. Конечная цель – получить три файла, показанные на рис. 5.16.

Это лишь один из многих способов разбить библиотеку на части. Можно было бы поместить каждый класс и каждую функцию в отдельный файл или оставить в одном файле все открытые функции и класс, а в другой перенести закрытые.



Рис. 5.15. Файл loglib.dart уже содержит несколько классов и функций, и их количество может увеличиваться

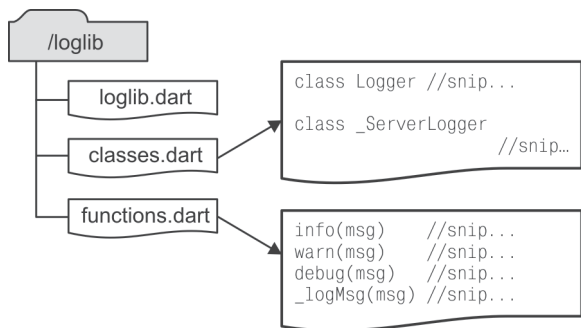


Рис. 5.16. Конечная цель – разбить библиотеку на три файла

Совет. Часто библиотека насчитывает несколько функционально слабо связанных частей, каждая из которых состоит из нескольких классов. В таком случае рекомендуется помещать каждую функциональную часть в отдельный исходный файл.

5.3.1. Ключевые слова *part* и *part of*

В Dart имеется ключевое слово `part`, предназначенное для разбиения библиотечного кода на несколько файлов. Оно используется в том же файле, что и слово `library`, и должно сопровождаться относительным путем к другому исходному файлу, входящему в состав библиотеки, например: `part "functions.dart";`. Можете создать новые

пустые текстовые файлы `classes.dart` и `functions.dart` и переместить в них, соответственно, классы и функции. В листинге ниже приведен полный текст файла `functions.dart`.

Листинг 5.8. Полный текст исходного файла `functions.dart`

`part of loglib;` ← Обозначает, что это часть библиотеки `loglib`

```
_logMsg(msg) {  
  print(msg);  
  _ServerLogger serverLogger = new _ServerLogger();  
  serverLogger.send(msg);  
}  
  
info(msg) => _logMsg("INFO $msg");  
warn(msg) => _logMsg("DEBUG $msg");  
debug(msg) => _logMsg("WARN $msg");
```

Мы называем такой файл *частичным*, или просто *частью*. Его разрешается использовать только в контексте библиотеки, сам по себе он не дает ничего. Важно понимать, что частичный файл содержит фрагмент кода, который *мог бы* остаться в оригинальном библиотечном файле, но вынесен оттуда для удобства разработчика. С точки зрения видимости классов и функций или преобразования кода в JavaScript, безразлично, состоит библиотека из одного файла или из нескольких.

После переноса функций и классов в отдельные файлы импортируем их, как показано на рис. 5.17.

За ключевым словом `part` в предложении `part "classes.dart";` должен идти путь к файлу относительно файла, указанного в объявлении `library`; например, можно было бы поместить все связанные исходные файлы в подпапку той папки, где находится основной библиотечный файл `loglib.dart`:

```
part "sourceFiles/classes.dart";
```

Теперь файл `loglib.dart` содержит только предложения `library`, `import` и `part`.

Такая организация, безусловно, допустима, вы постоянно будете с ней сталкиваться при изучении встроенных библиотек Dart. Напомним (см. раздел 5.1), что библиотека может импортировать другие библиотеки. Если бы это было нужно в библиотеке `loglib`, то в файле появились бы также предложения `import`:

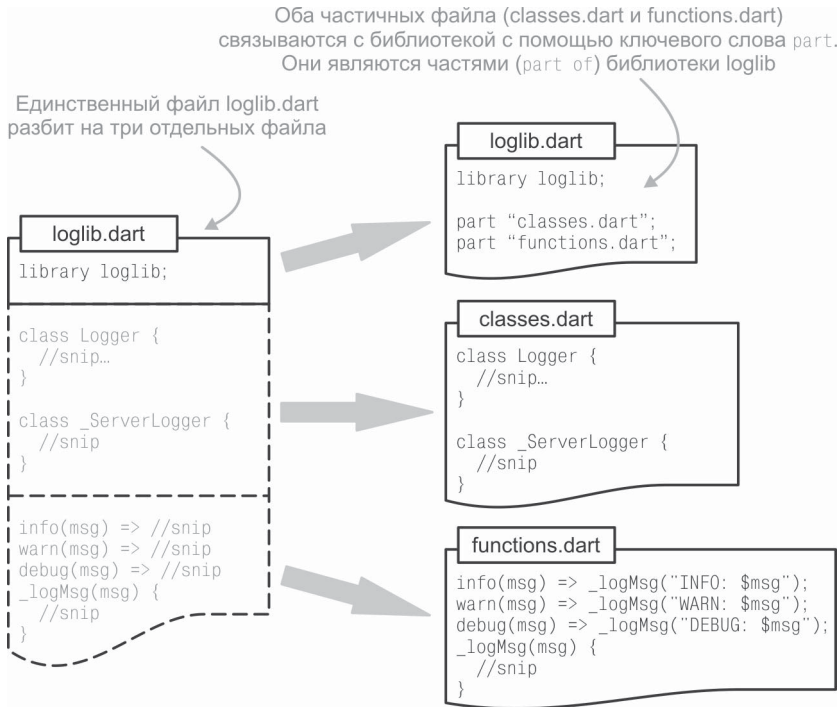


Рис. 5.17. Разбиение одного библиотечного файла на несколько исходных

```
library loglib;
```

```
import "other_library.dart"; ← В библиотеку loglib импортируется внешняя библиотека
```

```
part "classes.dart";
part "functions.dart";
```

Порядок этих предложений важен, как следует по рис. 5.18.

Все классы и функции из внешних библиотек, в частности `import "dart:html";`, становятся доступными всем частичным файлам, входящим в состав библиотеки `loglib`. Таким образом, отношения между классами и функциями, принадлежащими библиотеке, остаются неизменными, хотя они и распределены между несколькими исходными файлами.

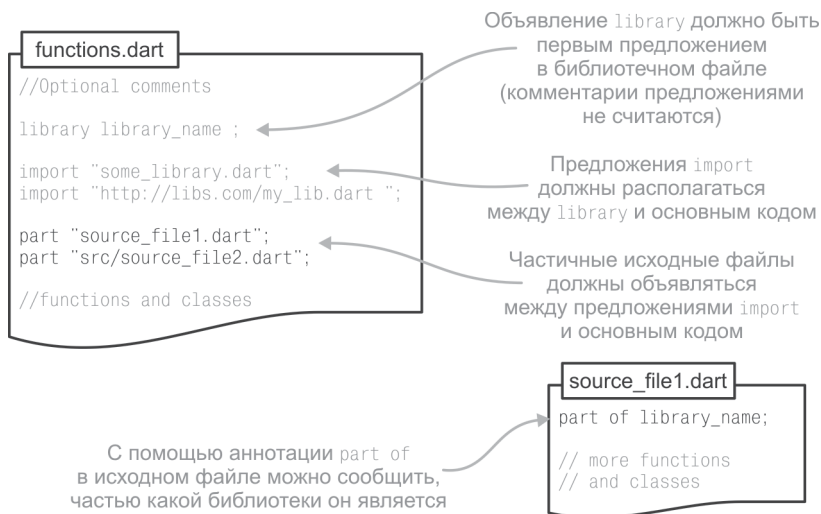


Рис. 5.18. Предложения part должны предшествовать основному коду

Ограничения на исходные файлы

При использовании ключевого слова `part` следует помнить о нескольких ограничениях.

- Файлы, включенные в библиотеку с помощью `part`, рассматриваются так, будто они являются частью оригинального библиотечного файла. Поэтому они не могут сами содержать предложений `library`, `import` и `part`. В противном случае был бы нарушен строго определенный порядок этих предложений.
- Каждый исходный файл может принадлежать только одной библиотеке. В библиотеках `loglib` и `webloglib` не должно одновременно встречаться предложение `part "classes.dart";`.
- Любой класс или функция должен находиться целиком в одном файле. Не существует способа поместить часть кода класса или функции в один файл, а часть – в другой (нет понятия частичного класса, как в C#).

Если рассматривать частичные файлы как части одного и того же логически единого библиотечного файла, то эти ограничения имеют смысл. Не может часть принадлежать двум разным целым и не может библиотечный файл включать где-то в середине еще одно предложение `library`. Кроме того, должно быть запрещено разбивать класс или функцию на куски, находящиеся в разных файлах.

Важно помнить, что распределение классов и функций по разным файлам никак не связано с ограничением доступа. Все они счита-

ются частями одной библиотеки, а ограничения доступа определены на уровне библиотеки.

Памятка

- Единый библиотечный файл можно разбить на несколько частей.
 - Внешние пользователи не знают, что библиотека физически состоит из нескольких файлов.
 - В Dart библиотечный файл, физически разбитый на несколько частейных файлов, считается единой логической библиотекой.
-

Помимо инкапсуляции функциональности и предоставления своего кода в распоряжение других разработчиков, библиотеку можно запускать и напрямую, как обычный скрипт.

5.4. Упаковка библиотек

В Dart пакетом называется автономное приложение или одна либо несколько библиотек, собранных в единое целое, обладающее номером версии. Пакеты служат двум целям: с одной стороны, позволяют приложению импортировать пакеты, созданные другими людьми, а с другой – организовывать собственные файлы так, чтобы код можно было упаковать и импортировать в чужое приложение.

Инструмент `pub`, встроенный в редактор Dart и доступный также в виде командной утилиты, позволяет импортировать пакеты, необходимые вашей программе. Сами пакеты могут находиться на веб-сервере, в репозитории GitHub (или любом другом Git-репозитории) или в репозитории `pub.dartlang.org`. Инструмент `pub` играет ту же роль, что другие менеджеры пакетов, например Maven в Java или `npm` в Node.js: он автоматически загружает зависимости вашего приложения и – рекурсивно – пакеты, от которых они зависят.

Вся информация, необходимая `pub` для поиска и загрузки зависимостей, хранится в файле `pubspec.yaml`, который должен находиться в корне дерева исходных файлов. Этот файл записан в формате YAML – ориентированном на человека языке разметки, в котором для обозначения разделов и подразделов применяются отступы. В листинге 5.9 приведен пример файла `pubspec.yaml`. Поле `name` обязательно, а поля `version` и `description` должны присутствовать, если вы собираетесь разместить пакет в репозитории `pub.dartlang.org`. Остальные поля необязательны. Особенно важен раздел `dependencies`, в котором содержится информация о зависимостях (если ваш па-

кет не зависит ни от чего, кроме базового Dart SDK, то этот раздел можно опустить).

Листинг 5.9. Пример файла `pubspec.yaml` для библиотеки `loglib`

```
name: loglib ← Имя пакета

version: 1.2.3 ← Версия пакета

description: >
  Provides a simple logging framework
  to log to the console
  Описание
  библиотеки

author: You <you@your.email.com> ← Можно использовать поле authors, в котором в формате
  YAML перечисляются несколько авторов

homepage: http://your.website.com/loglib ← Домашняя страница библиотеки

dependencies: ← Зависимости, которые должен загрузить pub

hosted_library:
  versioned_library: '1.2.3'
  Пакеты, размещенные на pub.dartlang.org,
  с номерами версий или без них

unittest:
  any
  Пакет размещается как часть Dart SDK,
  но не базовая библиотека

my_security_library:
  hosted:
    url: http://your.internal.server.com
    version: '>=1.0.0 <2.0.0'
  Закрытый пакет,
  размещенный
  на внутреннем сервере
  с указанием диапазона версий

some_opensource_library
  git:
    url: git://github.com/my/open_source_library.git
  Пакет, размещенный на GitHub
  (программа Git должна
  находиться в перечне путей
  на вашем компьютере)
```

В основу работы `pub` положен принцип примата соглашения над конфигурацией, он ожидает, что приложение организовано определенным образом. Основные файлы и папки показаны на рис. 5.19; к счастью, редактор Dart формирует требуемую структуру при создании нового проекта.

Для включения в проект различных зависимостей необходимо воспользоваться командами `pub install` и `pub update`, присутствующими в меню редактора Dart Editor. Первая команда устанавливает зависимости в структуру приложения, вторая обновляет их, и

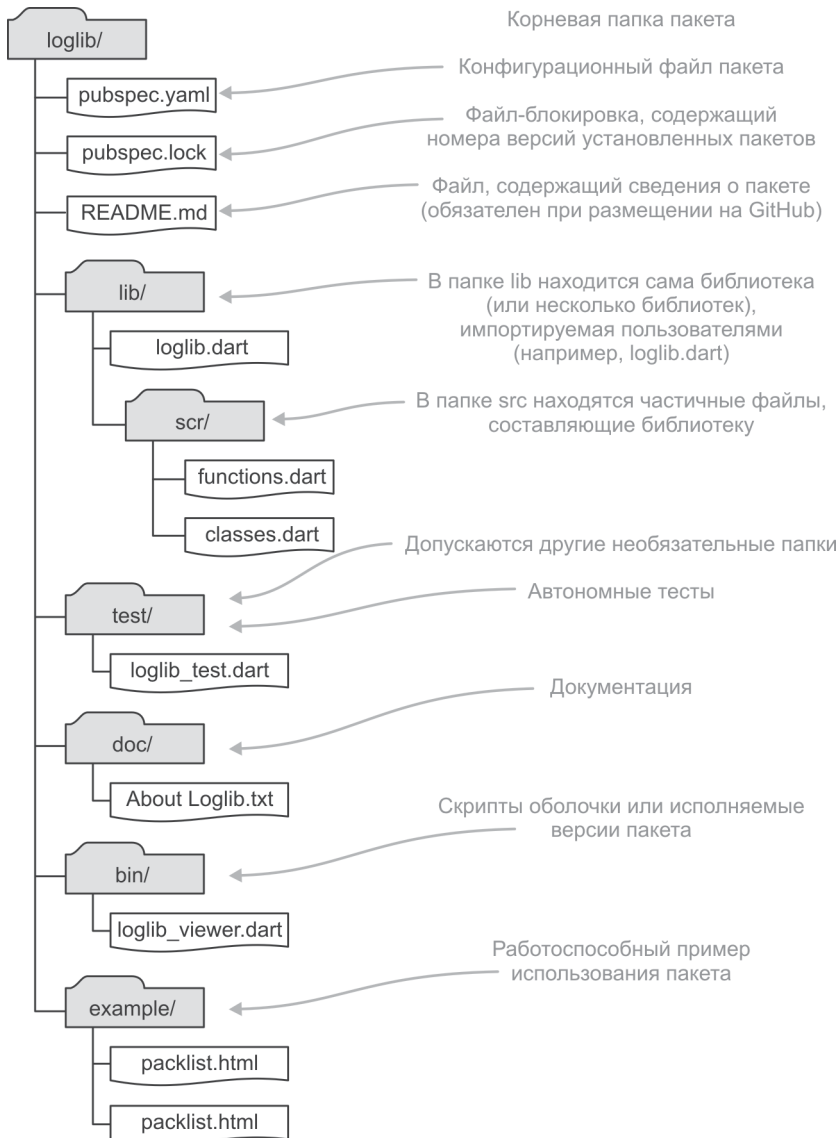


Рис. 5.19. Соглашение, определяющее структуру пакета

обе создают файл `pubspec.lock`. Пакеты загружаются в кэш, обычно в папку `.pub-cache/` в вашем домашнем каталоге. Файл `pubspec.lock` содержит актуальные версии различных зависимостей, установленных менеджером `pub` (полезно, когда задается диапазон версий). Этот файл можно сохранить в системе управления версиями, чтобы все члены команды гарантированно пользовались одними и теми же версиями зависимостей.

После того как зависимости установлены, на них можно ссылаться с помощью предложения `import`. Например, чтобы приложение `PackList` могло воспользоваться пакетом `loglib`, нужно написать:

```
import "package:loglib/loglib.dart";
```

← Сопоставляется с папкой пакета `/lib/`

5.5. Скрипты – это исполняемые библиотеки

Библиотека `loglib` предоставляет пользователям ряд классов и функций. Но ее можно запускать и непосредственно: вспомните, что Dart-скрипт – не более чем файл с расширением `.dart`, содержащий функцию верхнего уровня `main()`.

Пример использования библиотеки `loglib` в таком качестве – воспроизведение последовательности сообщений с сервера на консоли разработчика. Можно включить открытую функцию `replay(url)`, которая обращается к серверу и передает каждое из полученных от него сообщений уже существующей закрытой функции `_logMsg()`.

Чтобы воспользоваться новой функциональностью, можно было бы написать отдельное приложение, которое импортирует библиотеку `loglib` и вызывает `replay()`. Но для вызова одной-единственной функции слишком много работы. По счастью, Dart предлагает альтернативу. Библиотека также может содержать функцию `main()`, а больше ничего и не нужно, чтобы сделать библиотеку исполняемой (напомним, что функция `main()` – точка входа в любой Dart-скрипт). В листинге ниже показана функция `main()`, добавленная в библиотеку `loglib`.

Листинг 5.10. Библиотека, содержащая функцию `main()`

```
library loglib;

import "dart:html";

part "classes.dart";
```

```
part "functions.dart";

main() {  ← В библиотеке присутствует main()
  replay("http://www.someserver.com/logMessages");  ← main() вызывает функцию replay()
}

replay(url) {
  //опущено... загрузка списка msgsFromUrl
  for (msg in msgsFromUrl) {
    _logMsg(msg);  ← replay() вызывает закрытую функцию _logMsg()
  }
}
```

Теперь можно воспользоваться этой функциональностью в HTML-файле, включив в него `script`:

```
<script type="application/dart" src="loglib.dart"></script>
```

В результате сразу после загрузки скрипта вызывается функция `main()`.

Функцию `main()` лучше всего оставить в главном файле библиотеки (то есть том, который содержит предложение `library`), хотя можно вынести ее и в какой-нибудь частичный файл. Напомним, что работа функции или класса не зависит от того, находятся они в главном или в частичном файле, и функция `main()` в этом смысле — не исключение.

Отсюда вывод — любое написанное вами Dart-приложение можно превратить в библиотеку, добавив в начало файла объявление `library`. Значит, любое существующее приложение можно сделать библиотекой и погрузить в другое приложение. Добавив объявление `library` в приложение `PackList`, его можно сделать частью комплекса приложений с независимой функциональностью, объединенных единой темой (см. рис. 5.20).

Dart характеризуется исключительной модульностью и гибкостью и независимо от того, с чего вы начали — с библиотеки или с приложения, можно элементарно превратить одно в другое.



Рис. 5.20. Существующее приложение можно превратить в библиотеку, добавив в начало предложение `library`

5.6. Резюме

Dart поддерживает библиотеки для организации и повторного использования исходного кода. С помощью главного и частичных библиотечных файлов можно структурировать код таким образом, что над ним будет удобно работать коллективно и предоставлять в распоряжение третьих сторон.

Предложение `import` позволяет импортировать библиотеки в другие библиотеки и приложения. При этом с помощью префиксов легко удастся избежать конфликтов имен. Ограничение доступа, реализованное на уровне библиотеки, дает возможность отделить функции и классы, составляющие открытый интерфейс библиоте-

ки, от внутренних деталей реализации. Если добавить в библиотеку функцию `main()` – точку входа в любой Dart-скрипт, то она станет приложением.

Памятка

- Библиотеки могут импортировать другие библиотеки.
 - Библиотеку можно использовать как исполняемый скрипт.
 - Исходный код библиотеки можно распределить по нескольким частичным файлам.
 - Любой код, объявленный в библиотеке закрытым, доступен только внутри этой библиотеки.
 - Любой код, не объявленный в библиотеке закрытым, доступен также из внешней программы, импортировавшей эту библиотеку.
-

Итак, теперь вы знаете о Dart достаточно для разработки структурированного приложения, состоящего из нескольких библиотек и файлов. Вы также понимаете, что механизмы ограничения доступа в Dart работают на уровне библиотеки, а не класса. Пришло время глубже рассмотреть классы, интерфейсы и механизмы наследования в Dart и понять, как они сочетаются с динамической природой языка с факультативной типизацией.



Глава 6. Классы и интерфейсы

В этой главе

- ❑ Определение классов.
- ❑ Неявные интерфейсы в Dart.
- ❑ Конструирование классов.

Dart – объектно-ориентированный язык на основе классов с одиноким наследованием. С точки зрения механизмов классов и интерфейсов, он во многом напоминает Java и C#. В этой главе мы рассмотрим особенности Dart, которые позволяют проектировать гибкие библиотеки классов и интерфейсов. Мы поговорим также о рекомендуемых приемах, например о программировании относительно интерфейсов и включении именованных конструкторов для различных целей.

Мы будем иметь дело с различными способами конструирования классов, в том числе с *кажущейся* возможностью создать экземпляр абстрактного класса, и объясним, почему вызов конструктора может вернуть уже имеющийся объект. При обсуждении фабричных конструкторов мы затронем вопрос о статических свойствах и методах и о состоянии, общем для всех экземпляров класса. Мы увидим, что такое константные конструкторы и почему их следует использовать лишь в сочетании с финальными свойствами, допускающими только чтение, – это позволяет создавать фиксированные, неизменяемые экземпляры классов.

Мы рассмотрим также интерфейсы, которые так широко применяются в Java и C#, но отсутствуют в JavaScript. Памятуя уроки Java и C#, проектировщики Dart предложили ряд удобных способов, облегчающих программирование относительно интерфейсов, например тот факт, что всякий класс одновременно является неявным интерфейсом. Вы увидите, как воплотить в жизнь максимум «программируйте относительно интерфейсов, а не классов». Проектируя библиотеку, так чтобы в ней использовались интерфейсы, вы упрощаете пользователям задачу автономного тестирования функций и подмены реализации классов из вашей библиотеки.

Примечание. Если вы уже знакомы с Java или C#, то, возможно, захотите всего лишь просмотреть эту главу по диагонали, но обратите внимание на неявные интерфейсы, то есть идею о том, что определение класса одновременно является подразумеваемым интерфейсом. Логическая структура классов и интерфейсов в Dart имеет много общего с концепциями Java и C#. Краткий справочник по синтаксису можно найти в приложении.

Для начала посмотрим, как определяется простой класс в Dart.

6.1. Определение простого класса

Класс в Dart может содержать простые поля, методы чтения и установки этих полей (аксессуары) и методы. Сообща они называются *членами* класса. Говоря о профиле пользователя, можно определить класс `User`, в котором есть поля `forename` и `surname`, а также метод, возвращающий полное имя — конкатенацию `firstname` и `surname` (см. листинг ниже). Для создания экземпляра класса служит ключевое слово `new`, показанное в функции `main()`.

Листинг 6.1. Простой класс `User`

```
class User {                                ← Имя класса
  String _forename;                        ← Закрытое поле, о чем говорит подчеркив в начале имени

  String get forename => _forename;        | Открытые методы чтения и установки
  set forename(value) => _forename = value; | для доступа к закрытому полю

  String surname;                          ← Открытое поле типа String

  String getFullName() {                  | Метод, возвращающий
    return "$forename $surname";          | конкатенацию forename
  }                                       | и surname в виде строки
}

main() {
  User user = new User();                 ← Создается экземпляр класса User
  user.forename = "Alice";                | Устанавливаются
  user.surname = "Smith";                 | значения полей
  var fullName = user.getFullName();      ← Вызывается метод объекта user
}
```

Это очень простой класс. Важно отметить, что члены класса — метод `getFullName()` и поля `forename` и `surname` — подразумевают ин-

терфейс. В Dart этот подразумеваемый интерфейс рекомендуется использовать для воплощения в жизнь идеи о программировании относительно интерфейсов, а не конкретных классов. В любой момент можно подменить `User` другим классом, реализующим тот же интерфейс. Это существенное отличие Dart от языков Java и C#, в которых интерфейс должен быть определен явно. Рассмотрим практический пример.

6.1.1. Программирование относительно интерфейса класса

Рассмотрим воображаемую библиотеку `login`, в которой используется класс `User`. Детали реализации класса `User` и класса службы аутентификации находятся во внешней библиотеке `LogonLibrary`, разработанной другим коллективом. Наш код использует класс `AuthService` и класс `User`, не зная конкретных деталей, — как показано в листинге ниже.

Листинг 6.2. `logon.dart`: использование класса `AuthService`

<pre> import "logon_library.dart"; User doLogon(AuthService authSvc, String username, String password) { User user = authSvc.auth(username, password); print("User is authenticated:\${user==null}"); return user; } buttonClickHandler(event) { AuthService authSvc = new AuthService(); User user = doLogon(authSvc, query("#username").value, query("#password").value); } </pre>	Импортируется библиотека <code>LogonLibrary</code>, ← содержащая закрытые детали реализации В функцию передается экземпляр класса, выглядящего как <code>AuthService</code> ← Функция аутентификации ← предъявляет требования к интерфейсу, который реализует класс <code>AuthService</code> ← Печатается true или false в зависимости от того, успешно ли аутентифицирован пользователь Пример обработчика щелчка, который вызывает функцию <code>doLogon()</code>

При условии, что экземпляр `AuthService` имеет метод `auth()`, который возвращает экземпляр `User`, функция `doLogon()` будет отлично работать.

О внедрении зависимости, или Инверсии управления

Функция `doLogon()` принимает объект `authSvc` в качестве параметра. Это пример внедрения зависимости – приема, позволяющего подменять одну реализацию другой. Например, при тестировании `doLogon()` можно было бы передать в параметре `authSvc` подставной объект (mock object). При локальном тестировании можно использовать простой объект `authSvc`, а при развертывании в производственной среде – корпоративный объект `authSvc`.

Если бы у функции `doLogon()` не было параметра `authSvc`, то пришлось бы создавать внутри нее экземпляр конкретного класса. Тогда подменить реализацию `AuthService` было бы невозможно, что усложнило бы автономное тестирование и развертывание.

Допустим, мы хотим протестировать функцию `doLogon()`. С помощью внедрения зависимости можно использовать ту или иную реализацию `AuthService`. Вместо того чтобы работать с классом `AuthService`, который обращается за аутентификацией к реальному серверу, мы можем взять подставной класс, реализующий подразумеваемый интерфейс `AuthService` (рис. 6.1).

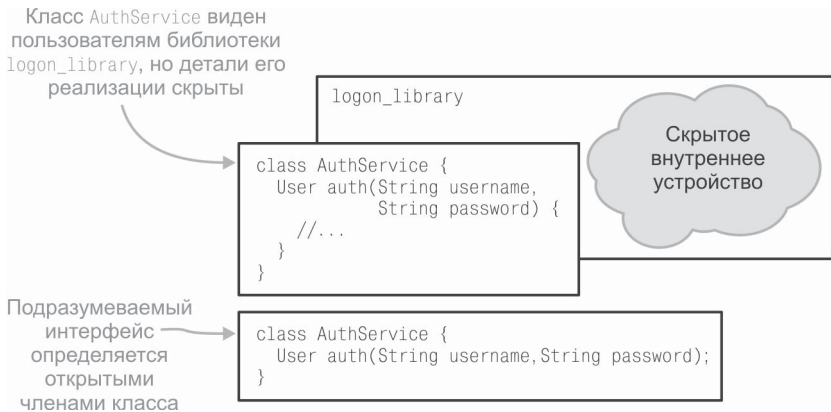


Рис. 6.1. Подразумеваемый интерфейс определяется открытыми членами класса

Открытый интерфейс `AuthService` состоит из единственного метода, который принимает два строковых параметра и возвращает экземпляр класса `User`. Используя ключевое слово `implements`, мы можем предоставить подставную версию – назовем ее `MockAuthService`, – с по-

мощью которой и протестируем `doLogon()`. В листинге ниже показан подставной класс и простой тест, помещенный в функцию `main()`.

Листинг 6.3. Использование подставной службы аутентификации

```
import "logon_library.dart";

class MockAuthService implements AuthService {
  User auth(String username, String password) {
    var user = new User();
    user.forename = "testForename";
    user.surname = "testSurname";
    return user;
  }
}

User doLogon(AuthService authSvc, String username, String password) {
  User user = authSvc.auth(username, password);
  print("User is authenticated:${user==null}");
  return user;
}

main() {
  AuthService authService = new MockAuthService();
  var user = doLogon(authService, "Alice", "password");
  print(user.forename);
  print(user.surname);
}
```

Подставной класс реализует подразумеваемый интерфейс другого класса

Реализация метода `auth()` в подставном классе

Существующая функция `doLogon()`

Пользуемся подставной реализацией

При вызове `doLogon()` используется подставная реализация

Создается экземпляр `MockAuthService`

Если объявлено, что класс явно реализует интерфейс, то инструментальные средства Dart могут проверить, что при изменении интерфейса изменились и все реализующие его классы. Если мы изменим класс `AuthService`, так что у метода `auth()` станет три параметра, то редактор Dart подскажет, что класс `EnterpriseLogonService` также необходимо изменить, тогда как для класса `MockLogonService`, в объявлении которого нет ключевого слова `implements`, ошибка выявится только на этапе выполнения.

Совет. Хотя в Dart разрешается не указывать в определении класса типы его членов (благодаря динамической типизации), рекомендуется не пренебрегать этим хотя бы для членов, составляющих открытый интерфейс класса. В этом случае вы сможете рассчитывать на помощь со стороны инструментальных средств, а заодно документируете свои намерения.

6.1.2. Формализация интерфейса путем явного его определения

Интерфейс определяет контракт, который класс обязуется выполнять. До сих пор мы работали с интерфейсами, неявно подразумеваемыми определением класса. Но Dart позволяет также определить интерфейс явно, независимо от класса, предоставляющего его реализацию. Явный интерфейс определяется, как и класс, — объявляются открытые члены класса, но без предоставления реализации (так называемый *абстрактный* класс). Давайте превратим AuthService в интерфейс и создадим его реализацию EnterpriseAuthService. Класс MockAuthService оставим без изменения.

Листинг 6.4. Определение явного интерфейса

```
abstract class AuthService {  ← Определяется абстрактный класс AuthService
    User auth(String username, String password); ← Определение метода auth()
}                               без реализации

class EnterpriseAuthService implements AuthService {  ← Реализует AuthService
    User auth(String username, String password) {    Реализация
        // некая реализация для корпоративной системы метода auth()
    }
}
```

Ключевое слово `abstract` важно, потому что благодаря ему Dart знает, что в этом классе есть методы, не имеющие реализации. В результате становится невозможно создать экземпляр AuthService непосредственно, то есть следующее предложение считается ошибкой:

```
var authService = new AuthService(); ← Создать экземпляр абстрактного класса AuthService
                                     невозможно
```

Однако ниже мы увидим, что такой синтаксис все же возможен даже для абстрактного класса, если воспользоваться фабричными конструкторами, предоставляющими реализацию по умолчанию.

Примечание. В Dart, говоря об интерфейсе, мы имеем в виду *подразумеваемый* интерфейс класса. Является ли этот класс абстрактным или конкретным — содержащим реализации всех своих методов, — не играет роли. Именно в таком смысле мы будем употреблять слово *интерфейс* на протяжении этой книги.

6.1.3. Реализация нескольких интерфейсов

Одно из преимуществ программирования относительно интерфейса заключается в том, что Dart позволяет одному классу реализовывать несколько интерфейсов. Например, можно определить абстрактный класс `RolesService`, в котором имеется метод `getUserRoles()` для получения информации о ролях пользователя. Тогда класс `EnterpriseAuthService` сможет реализовать интерфейс `RolesService` в дополнение к `AuthService`, предоставив открытый метод `getRoles()`, как показано в листинге ниже.

Листинг 6.5. Реализация нескольких интерфейсов

<pre>abstract class RolesService { List getRoles(User user); }</pre>	Новый класс <code>RolesService</code>
<pre>abstract class AuthService { User auth(String username, String password); }</pre>	
Реализует <code>AuthService</code> и <code>RolesService</code>	
<pre>class EnterpriseAuthService implements AuthService, RolesService { User auth(String username, String password) { // некая реализация для корпоративной системы } List getRoles(User user) { // некая реализация для корпоративной системы } }</pre>	
	Предоставляет реализацию <code>getRoles()</code>

Теперь мы можем использовать класс `EnterpriseAuthService` всюду, где требуется экземпляр `RolesService`. Таким образом, между классом и всеми интерфейсами, которые он реализует, существует отношение «является». Можно сказать, что `EnterpriseAuthService` «является» `RolesService` и «является» `AuthService`. Следовательно, показанный в листинге ниже код, где экземпляр `EnterpriseAuthService` передается в различные функции, ожидающие получить `AuthService` или `RolesService`, допустим.

Листинг 6.6. Класс `EnterpriseAuthService`, реализующий несколько интерфейсов

```
User doLogon(AuthService authService, String username, String password) { ←
    return authService.auth(username, password);           Ожидает экземпляр AuthService
}

showRoles(RolesService rolesService, User user) { ← Ожидает экземпляр RolesService
    List roles = rolesService.getRoles(user);
    print(roles);
}

main() {
    var entService = new EnterpriseAuthService(); ← Создается экземпляр EnterpriseAuthService
    var user = doLogon(entService, "Alice", "password"); ← EnterpriseAuthService связан
    showRoles(entService, user); ← EnterpriseAuthService отношением «является»
}                                     связан отношением с AuthService
                                     «является» с RolesService
```

В этом примере во всех интерфейсах объявлено только по одному методу. Но вообще-то никаких ограничений на количество методов или свойств в определении интерфейса, как и в определении класса, не накладывается.

Совет. Рекомендуется проектировать интерфейсы так, чтобы в одном интерфейсе была сгруппирована взаимосвязанная функциональность. Тогда пользователь сможет написать один класс, реализующий несколько интерфейсов или несколько классов – как будет удобно. Создав один большой интерфейс, вы лишите пользователя такого выбора.

6.1.4. Объявление акцессоров свойств

До сих пор мы рассматривали интерфейсы и классы, содержащие только методы. Но во многих классах определены также и поля. Например, в интерфейсе `AuthService` можно было бы определить поле `isConnected`, показывающее, подключена служба аутентификации к серверному хранилищу данных или нет.

В главе 3 мы видели, что к свойству, для которого определены методы чтения и установки (акцессоры), можно обращаться как к простому полю:

```
authService.isConnected = true;
```

Но раз для вызывающего кода нет разницы между полями и аксессуарами свойств, то объявлять интерфейс тоже можно обоими способами. Добавим аксессуары свойства `isConnected` (с помощью ключевых слов `get` и `set`) в интерфейс `AuthService` и простое поле `isConnected` — в интерфейс `RoleService`, как показано в листинге ниже.

Листинг 6.7. `logon_library.dart`: аксессуары чтения и установки свойств в интерфейсах

```
library logonlib;

abstract class AuthService {
  User auth(String username, String password);
  bool get isConnected;    ← С помощью get определяется аксессуар чтения свойства
  void set isConnected(bool value); ← С помощью set определяется аксессуар установки
                                свойства
}

abstract class RoleService {
  List getRoles(User user);
  bool isConnected;        ← Определяется поле
}

class EnterpriseAuthService implements AuthService, RoleService {
  bool _isConnected;
  bool get isConnected => _isConnected;
  void set isConnected(bool value) => _isConnected = value;

  // auth() и getRoles() опущены
}
```

Реализация аксессуаров
чтения и установки
удовлетворяет требова-
ния интерфейсов
`AuthService` и `RoleService`

Поскольку синтаксис чтения и записи одинаков что при обращении к самому полю, что при использовании аксессуаров, то и реализовывать свойство, объявленное в интерфейсе, можно любым из двух способов. Тот и другой удовлетворяют требованию интерфейса, которое, как вы помните, состоит в возможности писать такой код:

```
authService.isConnected = true;
```

Если реализующий класс это допускает, то как именно реализован интерфейс, не важно.

Памятка

- Определение класса неявно является также определением интерфейса.
 - Ключевое слово `abstract` позволяет объявить класс, не предоставляя реализации методов, его можно использовать для явного определения интерфейса.
 - Ключевое слово `implements` означает, что класс предоставляет реализацию методов интерфейса.
 - Класс и интерфейс связаны отношением «является».
-

Определяя интерфейс, вы предоставляете механизм расширения и повторного использования своего кода; именно так и устроены библиотеки Dart. Фундаментальные «классы» в Dart, такие как `String`, `int` и `List`, на самом деле являются интерфейсами. Выше в этой главе мы отмечали, что иногда код на Dart выглядит так, будто мы создаем экземпляр интерфейса, хотя на самом деле нужно создать экземпляр класса. В качестве убедительного примера такого рода приведем следующее предложение:

```
var myList = new List();
```

На первый взгляд, этот код вообще не должен работать, потому что нельзя создать экземпляр интерфейса. Это правда, но в следующем разделе мы покажем, почему подобная конструкция работает и какие вообще существуют способы создания новых экземпляров классов.

6.2. Конструирование классов и интерфейсов

Проектируя библиотеку для собственного пользования или для предоставления третьим лицам, нужно позаботиться о том, чтобы она была достаточно гибкой в использовании. Всячески рекомендуется всюду, где возможно, публиковать интерфейсы, чтобы пользователи могли подменить вашу реализацию какой-либо другой.

Следует также дать пользователям возможность создавать экземпляры классов различными способами, подходящими для конкретных случаев. В главе 3 мы поступали так при работе с классом `Element`: конструктору можно было передать фрагмент HTML-разметки или имя тега, и в любом случае получался элемент. При проектировании класса были предусмотрены различные варианты его использования.

Бывает также, что желательно повторно использовать экземпляр класса, например когда его создание обходится дорого. Скажем, если для создания объекта требуется подключаться к корпоративному серверу аутентификации, то лучше использовать уже существующее соединение, чем каждый раз создавать новое.

Наконец, иногда требуется, чтобы пользователь мог создать единственный неизменяемый экземпляр класса в определенном состоянии. Например, экземпляр класса конкретного сообщения об ошибке никогда не изменяется, и все экземпляры такого класса были бы идентичны. Dart позволяет сделать это с помощью константных конструкторов.

В этом разделе мы рассмотрим несколько способов создания экземпляра класса. Ключевое слово `new`, которое в языках JavaScript, Java и C# применяется для создания новых объектов, *почти* всегда имеет тот же эффект в Dart. Почему *почти*, а не *всегда*, мы увидим, когда перейдем к рассмотрению фабричных конструкторов, служащих для кэширования объектов. Вы узнаете также о ключевом слове `const`, которое позволяет создать неизменяемый экземпляр класса.

Начнем с изучения метода конструирования класса, с которым впервые познакомились в главе 3. Вы увидите, как обобщить функциональность конструктора, поручив ему инициализацию полей, и как создать несколько именованных конструкторов.

6.2.1. Конструирование экземпляров класса

Ключевое слово `new` создает новый экземпляр класса, например `new EnterpriseAuthService();`. При этом на самом деле вызывается метод конструирования класса, или просто *конструктор*. Если конструктор не определен явно, используется конструктор по умолчанию, генерируемый компилятором. Имя конструктора класса совпадает с именем самого класса. Например, следующий конструктор делает то же самое, что конструктор по умолчанию, и плюс к тому содержит вызов функции `print()`:

```
class EnterpriseAuthService {  
  String connection; ← Инициализируется значением null  
  
  EnterpriseAuthService() {  
    print("в конструкторе");  
  }  
}
```

Конструктор

```
}  
  
main() {  
  var entSvc = new EnterpriseAuthService(); ← Выводится строка «в конструкторе»  
}
```

Как и всем прочим методам и функциям, конструктору можно передавать параметры. Например, конструктор `EnterpriseAuthService` можно изменить так, что он будет принимать параметр `connection`, который будет использоваться для инициализации поля `connection`:

```
class EnterpriseAuthService {  
  String connection;  
  EnterpriseAuthService(String connection) { ← В конструктор добавлен параметр  
    this.connection = connection; ← С его помощью инициализируется поле connection  
  }  
}
```

Примечание. Ключевое слово `this` в Dart означает то же самое, что в Java и C#, – содержит ссылку на экземпляр текущего класса. В JavaScript дело обстоит совершенно иначе – там значение `this` изменяется во время выполнения.

Все синтаксические конструкции, относящиеся к функциям, в том числе необязательные параметры и параметры по умолчанию, применимы также и к конструкторам, но у конструкторов есть одна дополнительная возможность. Опыт работы с Java позволяет сделать вывод, что конструкторы чаще всего применяются для инициализации внутренних полей, как в примере выше. В Dart для этой цели существует сокращенная нотация: если в параметре конструктора встречается слово `this`, то соответствующий параметр неявно инициализирует одноименное поле. Предыдущий пример можно было бы переписать в таком виде:

```
class EnterpriseAuthService {  
  String connection;  
  EnterpriseAuthService(this.connection) { } ← для инициализации поля connection  
}
```

6.2.2. Проектирование и использование классов с несколькими конструкторами

В отличие от Java и C#, Dart не поддерживает перегрузку функций, методов и конструкторов (то есть возможность определять функции с одинаковыми именами, но разными списками параметров). Тогда как же удастся иметь несколько конструкторов для создания экземпляров класса разными способами?

В Java или C# мы завели бы несколько одноименных конструкторов с разными сигнатурами. Dart же предлагает иной механизм, напоминающий префиксы библиотек, рассмотренные в главе 5. Dart рассматривает имя класса как префикс конструктора и позволяет использовать именованные конструкторы с различными именами (и, если необходимо, разными параметрами). На рис. 6.2 показаны эквивалентные конструкторы в Java и в Dart.

Вспомним, что в главе 3 мы создавали элементы пользовательского интерфейса с помощью методов `Element.tag()` и `Element.html()`. Это именованные конструкторы, каждый из которых создает экземпляр класса `Element`, но по-разному. Точно так же можно включить именованные конструкторы в класс `EnterpriseAuthService`:

```
var authSvc1 = new EnterpriseAuthService();      ← Конструктор по умолчанию
var authSvc1 = new EnterpriseAuthService
↳ .withConn("connection string");              ← Именованный конструктор withConn
var authSvc2 = new EnterpriseAuthService
↳ .usingServer("localhost", 8080);              ← Именованный конструктор usingServer
```

Именованные конструкторы полезны, потому что позволяют выразить намерение проектировщика в имени конструктора. Выбирая подходящие имена, вы снабжаете код документацией, сообщая будущим пользователям, для чего предназначен данный конструктор.

6.2.3. Использование фабричных конструкторов для создания экземпляров абстрактных классов

Ранее в этой главе мы видели, что попытка создать экземпляр абстрактного класса, выступающего в роли интерфейса, завершается ошибкой:

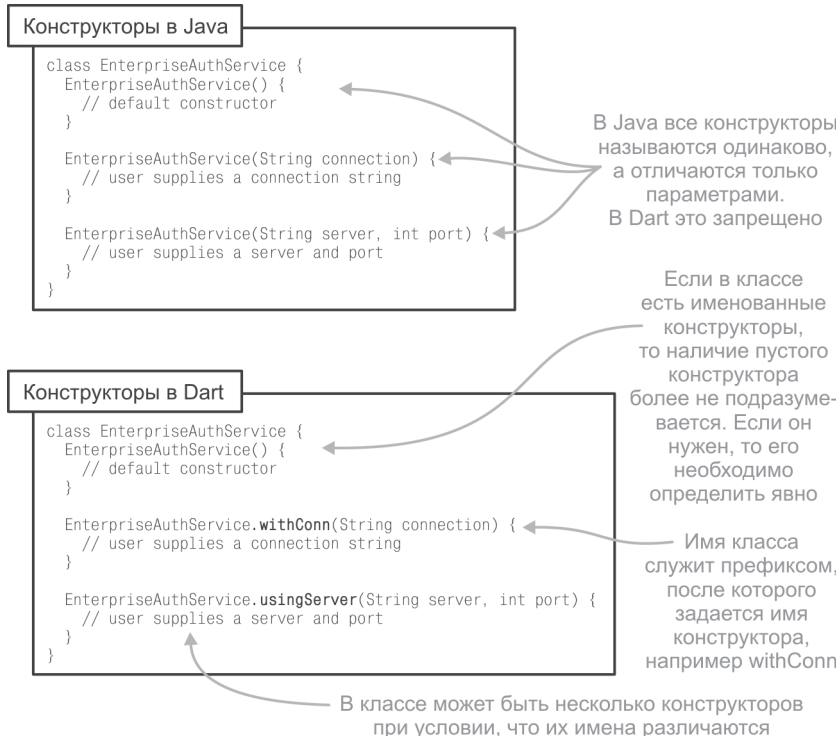


Рис. 6.2. Сравнение синтаксиса конструкторов в Java и Dart.
В Dart необходимо создавать именованные конструкторы с именем класса в качестве префикса

AuthService authService = new AuthService(); ← Выдается предупреждение о создании экземпляра абстрактного класса

При разработке приложения рекомендуется по возможности использовать имена интерфейсов, а не конкретных классов. Это позволяет подменять реализации, что очень удобно для автономного тестирования с помощью подставных объектов. Практика задания интерфейсов в определениях типов обеспечит максимальную гибкость при работе с вашей библиотекой.

Однако часто бывает, что пользователи библиотеки в большинстве случаев нуждаются в одном конкретном классе. Хорошим примером

может служить абстрактный класс `AuthService`, определяющий интерфейс. Как правило, тип интерфейса `AuthService` используется вместе с реализующим его классом `EnterpriseAuthService`, который входит в ту же библиотеку. Поэтому многие пользователи пишут такой код:

```
AuthService authSvc = new EnterpriseAuthService();
```

Подобный подход только усложняет работу с библиотекой, потому что теперь пользователь должен помнить о двух типах и о том, что один из них – интерфейс, а другой – реализующий его класс. Подобный код сплошь и рядом встречается в мире Java, где гигантские библиотеки представлены в виде цепочек интерфейсов, и понять, какой же класс содержит реализацию, не так-то просто.

К счастью, Dart позволяет проектировщику библиотеки воспользоваться фабричными конструкторами абстрактного класса, которые возвращают экземпляр конкретного класса, подразумеваемого по умолчанию. Если в определении конструктора встречается ключевое слово `factory`, значит, этот конструктор создает и возвращает корректный объект, как показано в листинге ниже.

Листинг 6.8. Использование фабричных конструкторов для создания объектов по умолчанию

```
abstract class AuthService {  
    User auth(String username, String password);  
  
    factory AuthService() {  
        return new EnterpriseAuthService();  
    }  
}
```

← Определяется фабричный конструктор
← Он обязан вернуть экземпляр конкретного класса

При таком подходе пользователь вашего интерфейса может написать такой код:

```
AuthService authSvc = new AuthService();
```

Следовательно, появляется возможность работать с абстрактным классом, будто у него есть реализация, но при этом не отказываться от всячески приветствуемой практики программирования относительно интерфейса. Употребляя типы Dart `String`, `List` и `int`, вы на самом деле используете интерфейсы, у которых есть реализация по умолчанию.

Фабричные конструкторы, как и обычные, могут иметь имена. В следующем разделе мы увидим, что способностью фабричных конструкторов возвращать экземпляры класса можно воспользоваться и в стандартных классах для реализации механизма кэширования.

6.2.4. Применение фабричных конструкторов для повторного использования объектов

В программах есть такие операции, которые желательно выполнять как можно реже. Они дорого обходятся либо с точки зрения обработки – например, влекут за собой создание большого количества вложенных объектов, либо времени – например, требуют подключения к удаленному серверу. Допустим, если Алиса входит в корпоративную систему, а спустя некоторое время то же самое делает Боб, то не следует каждый раз ждать пару секунд, пока будет установлено соединение; лучше взять уже готовое соединение из кэша. В таких случаях желательно сохранять объекты, например соединения, в пуле и возвращать объект из пула, а не создавать каждый раз новый. Это еще одно применение фабричных конструкторов, которые могут быть определены в любом классе, а не только в абстрактном.

Этим поведением можно воспользоваться в классе `EnterpriseAuthService`, преобразовав именованный конструктор `usingServer()` в фабричный (для чего достаточно добавить ключевое слово `factory`). Простейшая реализация такова:

```
class EnterpriseAuthService {  
  factory EnterpriseAuthService.usingServer(String server, int port) {  
    return new EnterpriseAuthService();  
  }  
}
```

Ключевое слово `factory` означает, что это фабричный конструктор

← Он обязан вернуть экземпляр `EnterpriseAuthService` (новый или уже существующий)

Синтаксис вызова фабричного конструктора не изменился

// где-то в другом месте программы

```
var authSvc = new EnterpriseAuthService.usingServer("localhost", 8080);
```

В этой простой реализации фабричность конструктора ничего нам не дает. Однако ситуация изменится, если кэшировать объекты, создаваемые фабричным конструктором. Один из способов сделать это – воспользоваться статическими свойствами класса, до которых мы скоро дойдем. А пока предположим, что где-то в приложении существует список объектов типа `EnterpriseAuthService`. В листинге

ниже показано, как получить существующий объект или вернуть новый. Интерфейс при этом не меняется.

Листинг 6.9. Использование фабричного конструктора для возврата существующего объекта из кэша

```
class EnterpriseAuthService {
    factory EnterpriseAuthService.usingServer(String server, int port) {
        var authService = getFromCache(server, port);
        if (authService == null) {
            authService = new EnterpriseAuthService ();
            // опущено: настроить authService и подключиться
            addToCache(authService, server, port);
        }
        return authService;
    }
    // другие методы и свойства опущены
}
```

Ключевое слово `factory` означает, что это фабричный конструктор

← Пытаемся получить существующий объект из кэша

Если в кэше нет подходящего объекта, создаем новый и помещаем его в кэш для дальнейшего использования

← Возвращаем объект. Вызывающая программа не знает, какой объект возвращен: полученный из кэша или новый

Совет. Если пользователь библиотеки пишет ключевое слово `new`, то он рассчитывает получить новый объект. Рекомендуется использовать фабричный конструктор, только если новые объекты нуждаются в предварительной обработке или если можно гарантировать, что повторное использование существующего объекта не станет сюрпризом для пользователя класса.

Фабричный конструктор становится особенно мощным средством в сочетании с механизмом сохранения экземпляров некоторого класса, но где эти экземпляры хранить? Оказывается, что подходящим местом является статическое свойство класса, об этом мы и поговорим далее.

6.2.5. Использование статических методов и свойств совместно с фабричными конструкторами

В примере фабричного конструктора выше мы использовали функции `getFromCache()` и `addToCache()`, но не объяснили, как они работают на практике. Нам необходимо какое-то центральное храни-

лишь экземпляров класса `EnterpriseAuthService`. По счастью, Dart, не отставая от Java и C#, позволяет определять в классе статические методы и свойства. Статическим называется член, который сообщается используется всеми экземплярами класса; он обозначается ключевым словом `static`. Таким образом, мы можем определить статический словарь (список пар ключ–значение), который разделяется всеми экземплярами класса `EnterpriseAuthService` в приложении. Тогда останется только посмотреть, есть ли в словаре объект с нужным ключом, определяемым именем и номером порта сервера. Если да, то мы можем вернуть его из фабричного конструктора. На рис. 6.3 показано, как работает этот алгоритм.

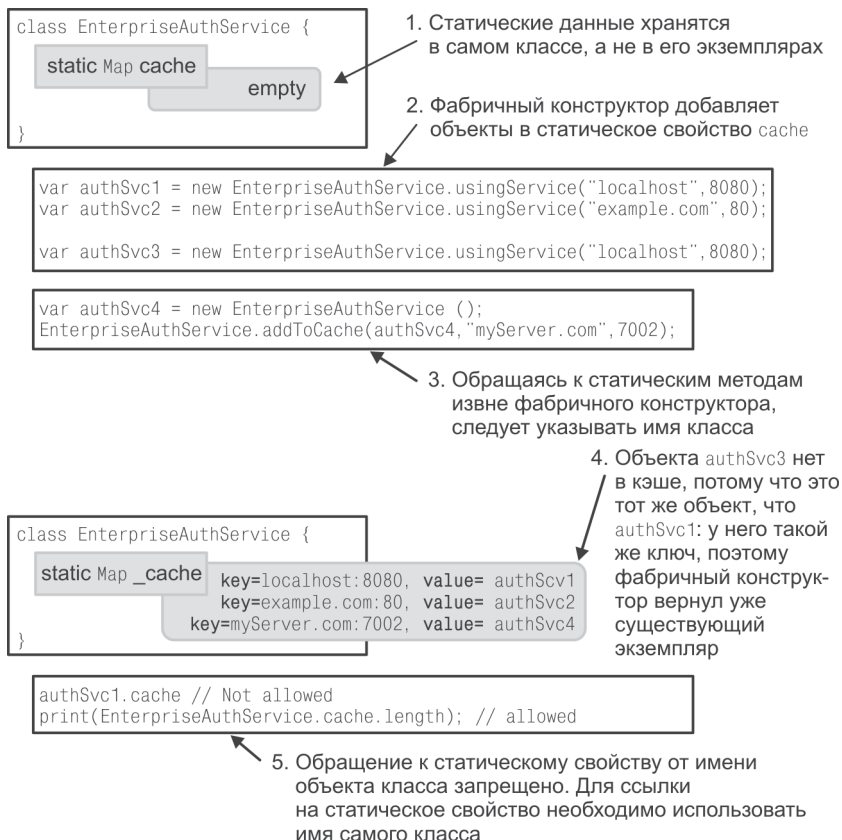


Рис. 6.3. Статические методы и свойства разделяются всеми экземплярами класса

При обращении к статическим методам и свойствам следует указывать имя класса, а не имя переменной (например, `authSvc2`): статические члены разделяются всеми экземплярами класса, то есть конкретного объекта-владельца у них нет. В листинге 6.10 показано статическое свойство `_cache` и акцессор чтения, который инициализирует его при первом обращении. Методы `getFromCache()` и `addToCache()` также статические (то есть их нельзя вызвать от имени объекта класса), первый извлекает объект из кэша, второй добавляет объект в кэш.

```
class EnterpriseAuthService {
    // другие методы и свойства опущены
    static Map _cache; ← В качестве кэша используется закрытый статический словарь

    static Map get cache { ← Открытый статический акцессор чтения при необходимости
        if (_cache == null) { инициализирует кэш и возвращает его
            _cache = new Map();
        }
        return _cache;
    }
    // getFromCache() возвращает
    // объект, если он есть в кэше
    // (и null в противном случае)

    static EnterpriseAuthService getFromCache(String server, int port) { ←
        var key = "$server:$port";
        return cache[key];
    }

    static addToCache(EnterpriseAuthService authService,
        String server, int port) { ← addToCache() добавляет объект в кэш
        var key = "$server:$port";
        cache[key] = authService;
    }

    factory EnterpriseAuthService.usingServer(String server, int port) {
        var authService = getFromCache(server, port);
        if (authService == null) {
            authService = new EnterpriseAuthService();
            // опущено: настроить authService и подключиться

            addToCache(authService, server, port);
        }
        return authService;
    }
}
```

Статические свойства и методы полезны, когда нужно сделать некое значение доступным всем экземплярам класса. В Java и C# возникают проблемы при обращении к статическим свойствам из нескольких потоков, но поскольку Dart – однопоточный язык, то об этом можно не беспокоиться. В Dart все операции потокобезопасны. О том, как обеспечивается параллелизм, мы будем говорить в главе 15 при обсуждении изоляторов.

Рассмотрев обычные и фабричные конструкторы, мы можем перейти к последнему способу создания экземпляров класса. Для этого нам потребуется новое ключевое слово `const`, которое, как и слово `factory`, употребляется в объявлении и обозначает константный конструктор. Однако у слова `const` есть и другое применение: вместо ключевого слова `new` для создания константных, неизменяемых экземпляров класса.

Памятка

- В абстрактных классах можно использовать фабричные конструкторы для создания экземпляра реализующего класса, подразумеваемого по умолчанию. Это позволяет пользователям библиотеки не отступать от качественного дизайна и работать исключительно на уровне интерфейсов.
 - В классе может быть несколько именованных конструкторов с именами, начинающимися именем класса.
 - Ключевое слово `factory` позволяет создавать классы, которые по видимости возвращают новый объект, но на самом деле берут его из кэша.
-

6.3. Создание константных классов с неизменяемыми полями

Класс `EnterpriseAuthService` может возвращать объект в ошибочном состоянии, показывающий, что пользователя не удалось аутентифицировать. Если сервер перестал отвечать, а запросов много, то пришлось бы создавать и возвращать сотни идентичных ошибочных объектов, потребляющих память на клиентском компьютере. Но, создав `const`-объект, мы получим единственный объект, представляющий ошибку, который можно использовать повторно. Для определения константного объекта нужно сделать две вещи: все поля в нем должны быть определены с ключевым словом `final`, а при определении конструктора и создании экземпляра следует употреблять ключевое слово `const`.

6.3.1. Финальные значения и свойства

Константный класс полезен, когда заведомо известно, что пользователь не должен изменять его свойств. Все значения, например коды состояний или тексты сообщений об ошибках, фиксированы на этапе компиляции.

Чтобы обеспечить такое поведение, Dart налагает на класс еще одно ограничение: все свойства класса должны быть определены с ключевым словом `final`, которое означает, что переменная не может быть изменена. Это слово употребляется вместо `var` или в сочетании со строгим типом:

<code>final myObject = new Object();</code>	← Используется ключевое слово <code>final</code> вместо <code>var</code>
<code>final Object anotherObject = new Object();</code>	← Используется ключевое слово <code>final</code>
	в сочетании с именем строгого типа
<code>myObject = new Object();</code>	Модификация содержимого финальной переменной, после того как она определена, считается ошибкой
<code>anotherObject = new Object();</code>	

Невозможно определить финальную переменную, не задав для нее начального значения, потому что компилятор должен вычислить это значение на этапе компиляции. Однако свойства класса *разрешается* объявлять финальными и без инициализации, которая может быть произведена в блоке инициализации конструктора.

6.3.2. Блок инициализации конструктора

Конструктор класса допускает инициализацию финальных свойств до начала своего выполнения. Блок инициализации конструктора располагается между списком параметров и телом конструктора; он представляет собой список записанных через запятую команд, которые конструктор должен выполнить для надлежащей инициализации объекта. Блок инициализации используется в сочетании с телом конструктора, в котором могут инициализироваться нефинальные свойства. Основное различие между блоком инициализации и телом конструктора заключается в том, что только в теле допустимо употребление ключевого слова `this`. На рис. 6.4 показан блок инициализации конструктора.

После инициализации финальных свойств их значения изменить невозможно. Кроме того, все финальные свойства должны быть инициализированы до входа в тело конструктора, иначе Dart сообщит об ошибке. Объект класса `AuthError` можно создать следующим образом:

```
AuthError error001 = new AuthError("Сервер недоступен", 1);
```



Рис. 6.4. Блок инициализации конструктора размещается после списка параметров и может использоваться для инициализации финальных свойств

6.3.3. Использование ключевого слова *const* для создания константного конструктора

Изменять свойства объекта класса `AuthError` нельзя, потому что все они помечены ключевым словом `final`. Если создать второй объект, передав конструктору точно такие же значения, то этот объект ничем не будет отличаться от предыдущего, кроме адреса в памяти. Создание второго объекта можно предотвратить, объявив конструктор как константный с помощью ключевого слова `const`.

Листинг 6.11. Создание константного конструктора

```
class AuthError {
  final prefix = "Error: ";
  final String _message;
  final int _code;

  const AuthError(String message, int code) : _message = message,
                                              _code = code;

  String get errorMessage => "$prefix [$code] $message";
}
```

Имени `AuthError` предшествует ключевое слово `const`

← У константного конструктора не может быть тела

Имея константный конструктор, мы можем создавать с его помощью объекты. Dart знает, что все такие объекты неизменяемы, и позволяет сравнивать их на равенство быстро и эффективно. При создании объекта с помощью константного конструктора вместо ключевого слова `new` следует употреблять слово `const`:

```
AuthError errorA = const AuthError("Сервер не отвечает",1);
AuthError errorB = const AuthError("Сервер не отвечает",1");
print(errorA == errorB);
```

Конструктору переданы одинаковые параметры
← Объекты расположены по одному и тому же адресу в памяти, поэтому печатается «true»

Примечание. В константных конструкторах запрещена строковая интерполяция. Значения строк должны быть константами времени компиляции, а не создаваться динамически во время выполнения. Употребление ключевого слова `new` вместо `const` разрешено, но тогда Dart будет считать экземпляры различными, так что этого следует избегать.

Отлично! Мы рассмотрели все четыре способа создания классов и интерфейсов с помощью ключевых слов `new` и `const`. Для проектирования хорошей библиотеки требуется практический опыт, но Dart хотя бы обеспечивает необходимую гибкость за счет синтаксиса конструкторов.

Памятка

- Наличие в объявлении ключевого слова `final` означает, что переменную или свойство нельзя изменять после инициализации.
 - Блок инициализации конструктора необходим для инициализации финальных свойств.
 - Ключевое слово `const` позволяет определять константный конструктор и создавать неизменяемые экземпляры класса.
-

6.4. Резюме

Мы начали эту главу с демонстрации того, как интерфейс определяет контракт, который класс может реализовать с помощью ключевого слова `implements`. Программируя относительно интерфейсов, а не классов, мы можем без труда подменять одну реализацию другой на этапе выполнения, что особенно полезно при автономном тестировании с применением подставных классов. Dart развивает эту идею, позволяя создавать экземпляры интерфейсов, которые на самом деле являются экземплярами некоторого класса, подразумеваемого по

умолчанию. Кроме того, в Dart все классы одновременно являются интерфейсами, поэтому программировать относительно интерфейса можно даже тогда, когда этот интерфейс не определен явно.

Для создания экземпляров классов и интерфейсов в Dart применяются ключевые слова `new` и `const` и конструкторы следующих типов:

- ❑ именованные конструкторы, позволяющие по-разному конструировать объекты;
- ❑ фабричные конструкторы, которые можно использовать, в частности, для поиска существующих экземпляров класса в кэше, статическом словаре или еще где-то;
- ❑ константные конструкторы для создания единственного неизменяемого экземпляра константного класса.

В следующей главе мы рассмотрим наследование классов и интерфейсов. Dart – объектно-ориентированный язык, в котором имеется механизм наследования, аналогичный тому, что есть в Java и C#. Прототипического наследования, как в JavaScript, в Dart нет. Благодаря наследованию мы можем повторно использовать код, написанный и протестированный другими людьми, добавляя собственную функциональность.



Глава 7. Расширение классов и интерфейсов

В этой главе

- ❑ Построение иерархий наследования классов и интерфейсов.
- ❑ Знакомство с базовым классом `Object`.
- ❑ Смысл аннотации типа `dynamic`.

Объектно-ориентированное программирование не сводится только к инкапсуляции данных и поведения в классе. Есть также возможность расширять поведение, определенное в другом месте, что способствует повторному использованию кода. А это, в свою очередь, позволяет писать меньше собственного кода и, кроме того, повышает уверенность в правильности программы, так как существующий код, по идее, должен быть протестирован.

В этой главе мы изучим наследование классов и интерфейсов в Dart и в том числе покажем, как добавлять новую функциональность путем переопределения существующих методов и свойств и как писать абстрактные классы, выступающие только в роли родителей в иерархии наследования. Если вам доводилось работать на Java или C#, то все это покажется знакомым. Те же, кто переходит на Dart с JavaScript, должны обратить внимание на отсутствие прототипического наследования.

Мы рассмотрим базовый класс `Object`, которому наследуют все остальные классы и типы, включая строки, числа, булевские величины, `null` и ваши собственные классы. Класс `Object` предоставляет ряд стандартных методов и свойств, которые можно переопределять в своих классах, в том числе метод `toString()`. Мы поговорим также о типе `dynamic`, используемом в Dart для факультативной типизации.

В главе 6 мы создали несколько классов вокруг гипотетического примера службы аутентификации и авторизации `AuthService`. Когда Алиса передает приложению свои имя и пароль, функция `auth()` аутентифицирует ее и возвращает объект типа `User`, с которым приложение может работать. В настоящей главе мы разовьем этот при-

мер, построив на базе типа `User` иерархию классов и интерфейсов для простой службы `AuthService`. Напомним, что в библиотеке `logon_lib` были определены абстрактный класс `AuthService` и его реализация `EnterpriseAuthService` (рис. 7.1).

Вызов метода `auth()`
возвращает объект `User`

```
logon_lib

abstract AuthService {
  User auth(String username, String password msg);

  factory AuthService() {
    return new EnterpriseAuthService();
  }
}

class EnterpriseAuthService implements AuthService {
  User auth(String username, String password msg) {
    //...
  }
}
```

Рис. 7.1. Определение класса `AuthService` из библиотеки `logon_lib`

Мы воспользуемся наследованием для расширения типа `User`, так чтобы корпоративная служба аутентификации могла вернуть объект, представляющий корпоративного пользователя и обладающий определенной функциональностью.

Примечание. Если вы уже знакомы с Java или C#, то можете лишь бегло просмотреть эту главу, поскольку механизм наследования классов в Dart точно такой же. Dart допускает одиночное наследование класса и множественное наследование интерфейсов. Прочитайте «Памятки» в конце каждого раздела, где резюмируются наиболее важные идеи, и обратите особое внимание на метод `noSuchMethod`.

7.1. Расширение классов с помощью наследования

Начнем с определения базового класса для типа `User`, следуя той же модели, что при определении класса `AuthService` в главе 6. Определение типа `User` приведено в следующем листинге.

Листинг 7.1. Новый класс User

```
class User {  
    User (String this._username) { } ← Конструктор согласуется с определенным  
                                     в интерфейсе и инициализирует свойство _username  
  
    String _username;                | Закрытые  
    String _existingPasswordHash;    | свойства  
  
    String get username => _username; ← Акцессор чтения возвращает  
                                     закрытое имя пользователя  
  
    String emailAddress; ← Свойство для чтения и записи адреса электронной почты  
  
    bool isPasswordValid(String newPassword) { ← Метод для сравнения переданного  
        //... код проверки ...                пароля с существующим  
    }  
}
```

AuthService использует этот класс, так как возвращает объект User. Например, мы можем написать предложение `User aUser = new User("Alice");`, которое вернет экземпляр класса User, инициализированный именем "Alice". Теперь приложение может использовать эту переменную для проверки пароля и изменения адреса электронной почты.

Однако корпоративной службе нужно от класса User больше – например, возможность помечать, что учетная запись просрочена, или надежная схема смены паролей, гарантирующая, что последние пять паролей не повторяются. Этого можно добиться с помощью наследования. Механизм наследования позволяет добавить новую функциональность в существующий класс путем порождения *подкласса*, для чего предназначено ключевое слово `extends`. Унаследовав класс, вы можете также переопределить поведение существующих методов и свойств.

7.1.1. Наследование класса

Наследующий подкласс получает все методы и свойства родительского класса, называемого также *суперклассом*. Кроме того, подкласс связан отношением «является» со всеми классами, расположенными выше в иерархии наследования.

Dart допускает только одиночное наследование, то есть у подкласса не может быть больше одного родительского класса. Однако родительский класс может иметь много дочерних классов, а у каждо-

го дочернего могут быть свои потомки, так что образуется глубокая иерархия наследования.

В нашем примере пользователь Алиса, зашедшая в систему, будет представлена экземпляром класса `EnterpriseUser`, который возвращает метод `auth()` корпоративной службы аутентификации. Благодаря слову `extends` в своем объявлении этот класс наследует `User` и, стало быть, получает от него все члены. В данный момент Алису можно с одинаковым успехом представить любым из классов `User` или `EnterpriseUser`. Класс `EnterpriseUser` связан с классом `User` отношением «является». Проверить наличие отношения «является» позволяет ключевое слово Dart `is`, которое возвращает `true`, если объект является экземпляром указанного типа, и `false` в противном случае (рис. 7.2).

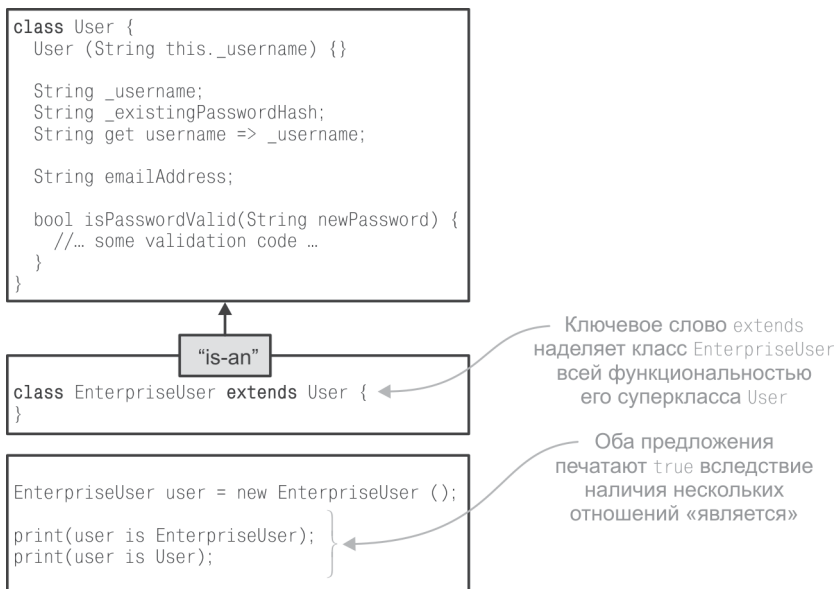


Рис. 7.2. Класс `EnterpriseUser` связан отношением «является» со всеми классами, расположенными выше него в иерархии наследования

Этот механизм обеспечивает значительную гибкость. Если известно, что некоторому методу требуются только возможности базового типа `User`, то можно объявить, что параметр имеет тип `User`,

передавая тем не менее экземпляры класса `EnterpriseUser`. Это позволяет другим разработчикам воспользоваться вашим кодом, передавая объекты созданных ими подклассов `User`, в твердой уверенности, что метод будет работать правильно.

Во фрагменте ниже показано, что объект `EnterpriseUser` можно использовать точно так же, как объект `User`:

<code>var user = new User("Alice");</code>		Используется
<code>print(user.username);</code>		экземпляр <code>User</code>
<code>var enterpriseUser = new EnterpriseUser("Alice");</code>		Экземпляр <code>EnterpriseUser</code>
<code>print(enterpriseUser.username);</code>		используется точно так же, как <code>User</code>

Примечание. Неявный интерфейс `EnterpriseUser` также наследуется от суперкласса. Открытый интерфейс любого класса является объединением открытых интерфейсов всех его предков.

В следующем листинге показана функциональность, присущая только классу `EnterpriseUser`. Таким образом, `EnterpriseUser` может пользоваться всей существующей функциональностью `User`, а добавлять только новую.

Листинг 7.2. Класс `EnterpriseUser` с добавленной функциональностью

```
class EnterpriseUser extends User {  
    EnterpriseUser() { } ← Пустой конструктор по умолчанию  
  
    void markExpired() {  
        // новая реализация  
    }  
}
```

7.1.2. Наследование конструкторов

Дочерний класс получает все методы и свойства его суперкласса, но конструкторы суперкласса не наследуются. На рис. 7.3 показано, какие члены класс `EnterpriseUser` наследует от `User`.

Чтобы воспользоваться конструктором суперкласса, мы должны написать свой конструктор в подклассе `EnterpriseUser` и с помощью ключевого слова `super` сослаться на конструктор суперкласса. Сделать это можно в *блоке инициализации конструктора*, который размещается между списком параметров и телом конструктора:

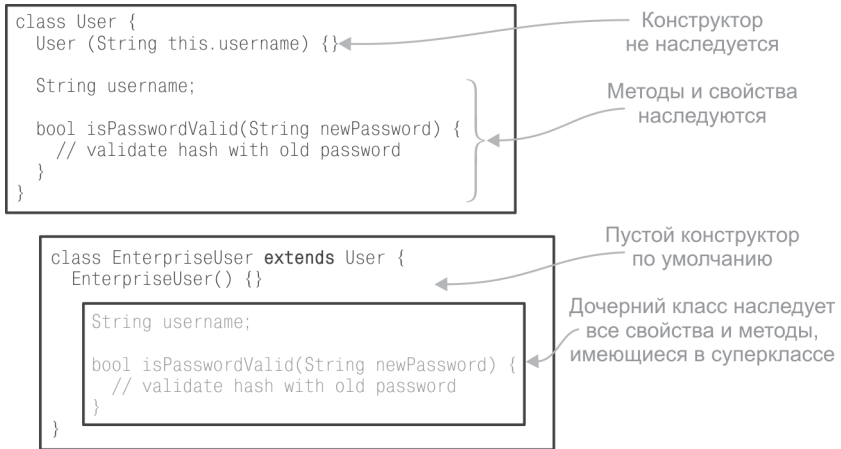


Рис. 7.3. Дочерний класс наследует всю функциональность суперкласса, за исключением конструкторов

```

class EnterpriseUser extends User {
  EnterpriseUser(String uname) : super(uname) {
    // пустое тело конструктора
  }
}

```

Конструктор родительского класса
← можно вызвать с помощью ключевого слова `super`, помещенного в блок инициализации конструктора

Dart исполняет блок инициализации еще до создания экземпляра класса. Это единственное место, в котором можно вызвать конструктор суперкласса, вызывать его в теле конструктора недопустимо. Разрешается вызывать любой конструктор суперкласса, в том числе именованный; список параметров конструктора подкласса не обязан совпадать со списком параметров какого-то конструктора суперкласса. Если в базовом классе имеется именованный конструктор, то его можно вызвать, указав имя суперкласса в качестве префикса имени конструктора. Точно так же именованный конструктор подкласса может вызывать произвольный конструктор суперкласса:

```

class EnterpriseUser extends User {
  EnterpriseUser (String uname, String email) : super.byEmail(email) {
    // пустое тело конструктора
  }

  EnterpriseUser.byUsername(String username) : super(uname) {
  }
}

```

Вызывается именованный конструктор родительского класса

Вызывается конструктор базового класса

Блок инициализации конструктора состоит из перечисленных через запятую команд, которые служат для инициализации финальных свойств (о чем рассказывалось в главе 6), однако вызов конструктора суперкласса всегда должен стоять первым.

7.1.3. Переопределение методов и свойств

Наследуя родительский класс, вы получаете всю функциональность его членов. Но бывает так, что требуется подменить унаследованную функциональность своей собственной. Рассмотрим, например, операцию смены пароля в классе `EnterpriseUser`. В его суперклассе `User` уже есть метод `isPasswordValid()` проверки пароля, и мы получаем его в результате наследования. Но если Алиса хочет сменить свой пароль в корпоративной системе, то недостаточно просто сравнить новый пароль со старым. В этом случае последние пять паролей должны различаться, а это означает, что класс `EnterpriseUser` должен *переопределить* функциональность, предоставленную родительским классом `User`, например сохранять пять последних сверток паролей. К счастью, это нетрудно сделать, написав новую реализацию метода с таким же именем и параметрами, что в суперклассе (см. листинг 7.3). Кстати, остается возможность все-таки воспользоваться унаследованным от класса `User` методом `isPasswordValid()`, если добавить при его вызове все то же ключевое слово `super`.

Листинг 7.3. Класс `EnterpriseUser` переопределяет функциональность, унаследованную от класса `User`

```
class EnterpriseUser extends User {
    bool isPasswordValid(String newPassword) {
        // опущено... сравнение с последними 5 паролями

        return super.isPasswordValid(newPassword);
    }
}
```

Функциональность родительского класса
← переопределяется в новой версии метода
← Вызов метода родительского класса с помощью ключевого слова `super`

Свойства можно переопределять аналогично. Свойство – это всего лишь сокращенная запись метода чтения или установки, поэтому принцип остается неизменным. В классе `User` имеется свойство `username`, но в корпоративной системе имя пользователя не должно быть короче четырех символов. Поэтому мы переопределим методы

чтения и установки, реализовав контроль по-другому, как показано в следующем листинге.

Листинг 7.4. `login.dart`: переопределение свойств, унаследованных от родительского класса

```
class EnterpriseUser extends User {           Унаследованная функциональность
  String get username => super.username;      ← повторяется без изменений

  void set username(String value) {
    if (value.length < 4) {                  ← Производится дополнительная проверка
      throw new ArgumentError("Ошибка: имя пользователя короче 4 знаков");
    }
    super.username = value;                 ← Значение передается в свойство родительского класса
  }
}
```

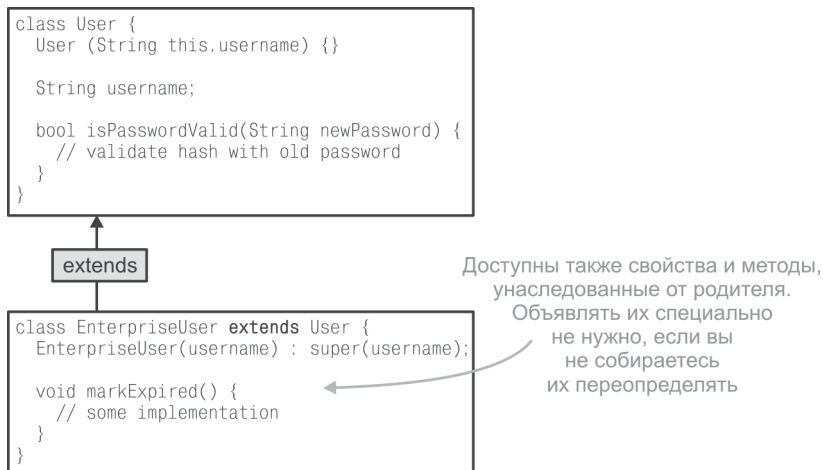
До сих пор мы видели, как в Dart производится наследование родительского класса, возможно, с переопределением части функциональности. Но иногда иерархия классов и интерфейсов намеренно проектируется так, чтобы пользователь *обязан* был предоставить отсутствующую реализацию. Для этого служат абстрактные классы.

7.1.4. Включение абстрактных классов в иерархию наследования

В главе 6 мы уже встречались с ключевым словом `abstract`, когда определяли явный интерфейс, то есть класс, содержащий лишь сигнатуры методов без реализации. Абстрактные классы можно использовать также для того, чтобы вынудить пользователя предоставить реализации некоторых методов, а другие реализовать самостоятельно.

Сейчас в нашем примере иерархия наследования состоит из двух классов. В системе для разработки Алиса представлена классом `User`, а в производственной системе, развернутой на корпоративных серверах, – классом `EnterpriseUser`, который связан со своим родителем отношением «является». Эта иерархия показана на рис. 7.4.

Система может создавать экземпляры классов `User` и `EnterpriseUser`, потому что тот и другой полны, то есть реализуют всю функциональность, требуемую интерфейсом. Класс `User` можно использовать независимо от его потомков. Однако часто при проектировании библиотеки требуется, чтобы программист, использующий ваши клас-

**Рис. 7.4.** Текущая иерархия наследования классов и интерфейсов

сы, кое-что реализовал сам, потому что на этапе проектирования просто неизвестно, какая функциональность может понадобиться.

Если говорить об интерфейсе `User`, то можно было бы добавить функцию `checkPasswordHistory()`, которая позволяет реализующим классам проверять историю паролей в функции `isPasswordValid()`. Увы, при проектировании класса `User` мы не знаем, как именно проверять историю. Поэтому мы можем заставить пользователей класса `User` дописать эту функциональность, вынудив их унаследовать `User` и переопределить метод `checkPasswordHistory()`.

Для этого при определении класса `User` мы добавим ключевое слово `abstract`, как показано в листинге ниже. Это слово означает, что создать экземпляр такого класса невозможно; мы должны разработать его подкласс, предоставляющий недостающую функциональность.

Листинг 7.5. Преобразование `User` в абстрактный класс

```

abstract class User {
    User (String this._username) { }

    String _username;
    String _existingPasswordHash;

    String get username => _username;
}

```

← Класс объявлен абстрактным, потому что отсутствует реализация метода `checkPasswordHistory()`

```
String emailAddress;

void checkPasswordHistory(String newPassword); ← Объявление метода без реализации

bool isValid(String newPassword) {
    // проверить, что newPassword не совпадает с
    // текущим паролем, сравним свертки
    checkPasswordHistory(newPassword); ← checkPasswordHistory() нет, мы можем его
}                                       вызвать, потому что он обязательно будет
}                                       реализован в дочернем классе
```

Одно из преимуществ абстрактного класса заключается в том, что мы можем вызывать отсутствующие методы, передавая ответственность за их реализацию подклассам. Коль скоро класс `User` сделан абстрактным, мы ожидаем реализацию метода `checkPasswordHistory()` от класса `EnterpriseUser`. Этот класс, связанный с классом `User` отношением «является», должен либо объявить себя тоже абстрактным, либо удовлетворить требования, предъявляемые классом `User`. На рис. 7.5 показано, как предоставленная реализация метода `checkPasswordHistory()` используется при вызове метода `isValid()` экземпляра класса `EnterpriseUser`.

Абстрактные классы хороши, когда нужно оставить решения по реализации на усмотрение пользователя библиотеки, но при этом сохранить за собой решение о порядке выполнения некоторых операций. Например, в методе `isValid()` мы сначала проверяем свертку пароля, а потом вызываем метод `checkPasswordHistory()`, потому что это может потребовать обращения к внешней системе. Таким образом, не выполняя этого шага в случае, когда новый пароль не прошел даже простую проверку, мы экономим ресурсы.

В этом – весьма насыщенном – разделе мы разобрались, как в Dart организовано наследование классов и интерфейсов. Объектно-ориентированное программирование вообще и наследование в частности – это сложная тема, и изучать ее лучше всего, экспериментируя и пытаясь воспользоваться теми или иными идеями в своих библиотеках. Встроенные в Dart SDK-библиотеки изобилуют интерфейсами и наследованием; их исходный код открыт, и его легко посмотреть в редакторе Dart. Обзор некоторых библиотек приведен в приложении.

Теперь, разобравшись с наследованием, перейдем к классу `Object`, который является предком всех классов в Dart вне зависимости от того, задано ли наследование от него явно.

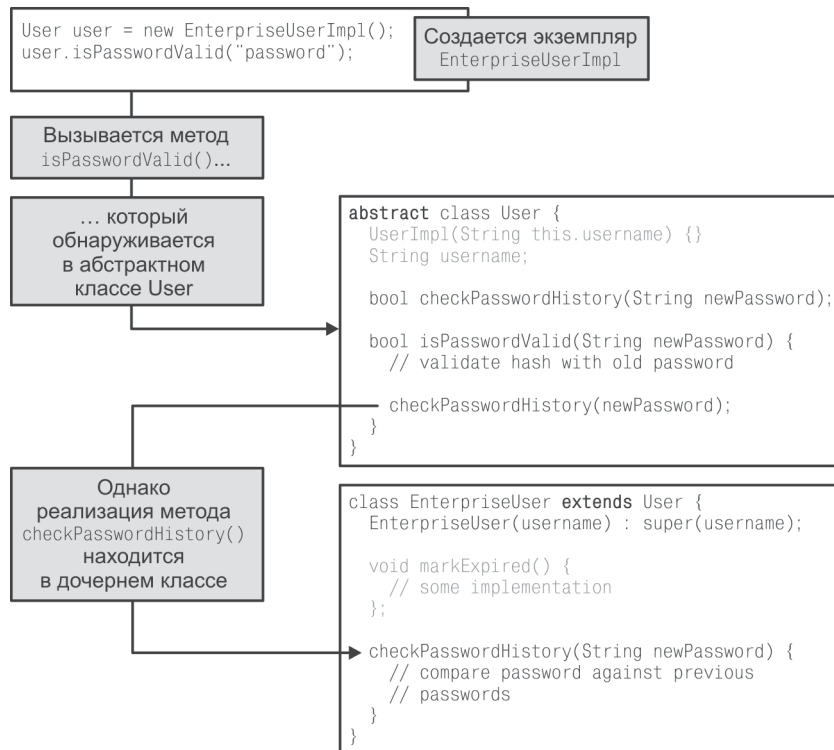


Рис. 7.5. Методы, объявленные в интерфейсах, могут быть реализованы дочерними классами, если родительский класс объявлен абстрактным

Памятка

- Ключевое слово `extends` означает, что класс наследует (является под-классом) другому классу.
- Ключевое слово `abstract` означает, что класс не предоставляет собственной реализации какого-то метода. Отсутствующую реализацию должны предоставить подклассы абстрактного класса.
- Подклассы не наследуют конструкторов суперкласса. Для вызова конструктора родительского класса нужно сослаться на него в блоке инициализации конструктора с помощью ключевого слова `super`.
- Ключевое слово `super` позволяет также обращаться к методам и свойствам суперкласса из тела его подкласса.

7.2. Все является объектом

В Dart все является объектом, и этим он отличается от Java и C#, где имеются примитивные типы, например `int` и `boolean` (у которых есть объектные эквиваленты, например `java.lang.Integer`). Класс `Object` встроен в сам язык Dart и является предком любого класса, кроме самого себя.

Создавая переменную в своем приложении – типа `String`, `int` или какого-нибудь класса, например `EnterpriseAuthService`, – вы создаете экземпляр класса `Object`. На это можно посмотреть с двух точек зрения. Во-первых, в объектно-ориентированном программировании создаются объекты, а с точки зрения информатики, объект – это область памяти компьютера, в которой хранятся данные в работающем приложении. Это не то же самое, что класс, каковой является выраженным на языке программирования чертежом, по которому конструируются объекты.

Во-вторых, в Dart имеется класс `Object`, которому автоматически наследует любой другой класс. Это происходит вне зависимости от того, имеется в определении вашего класса ключевое слово `extends` или нет. Все встроенные классы и типы, в частности `String`, `int`, `null` и функции, связаны отношением «является» с `Object`.

7.2.1. Проверка отношения «является» `Object`

Наследование классов формирует отношение «является» на восходящем пути по иерархии, и наличие такого отношения можно проверить с помощью ключевого слова `is`, которое возвращает `true` или `false`. Все показанные ниже предложения печатают `true`:

```
print(Object is Object);
print(1 is Object);
print(false is Object);
print(null is Object);
print("Hello" is Object);
var someFunction = () => "";
print(someFunction is Object);
```

Во всех случаях печатается «true»

Отношение «является» применимо и к написанным вами классам. Любой созданный вами класс, например `EnterpriseAuthService`, автоматически наследует `Object`. Обычно, чтобы унаследовать какому-то

классу, мы указываем его имя после ключевого слова `extends`. Но если родителем является `Object`, то это необязательно; ему ваш класс наследует и так, даже без `extends` (рис. 7.6).

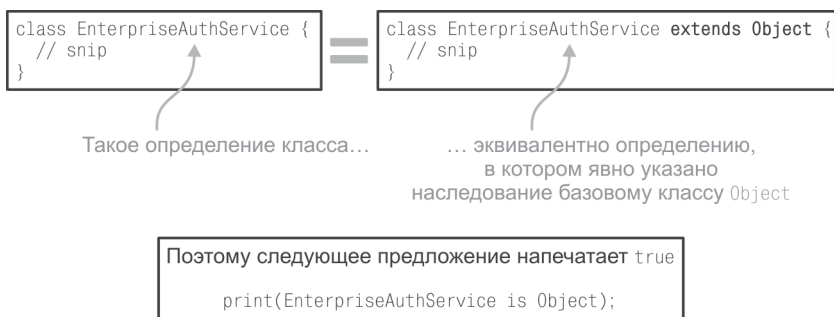


Рис. 7.6. Любой класс автоматически расширяет базовый класс `Object`

Наследование `Object` распространяется вниз по иерархии. Если класс `EnterpriseUser` наследует `User`, то он наследует также и `Object` — только опосредованно, через `User`. Так происходит потому, что отношение «является» транзитивно: если А «является» В, а В «является» С, то А «является» С. Это проиллюстрировано на рис. 7.7.

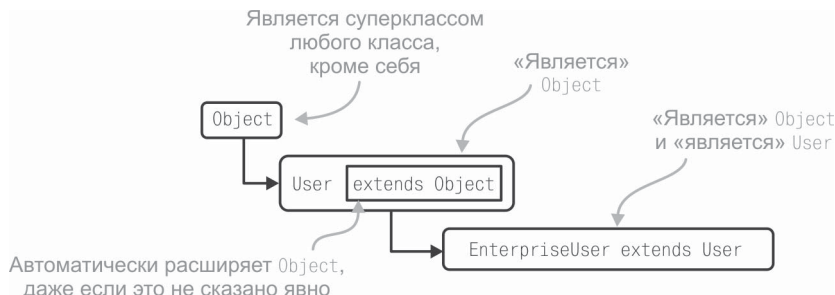


Рис. 7.7. Любой класс связан отношением «является» с классом `Object`

С помощью ключевого слова `is` мы можем проверить наличие отношения «является» и между собственными классами. Все показанные ниже предложения печатают `true`:

```
print(User is Object);
print(EnterpriseUser is Object);
print(EnterpriseUser is User);
```

Во всех случаях печатается «true»

Примечание. Отношение «является» одностороннее (распространяется вверх по иерархии, от потомка к родителю), потому `print(Object is User);` напечатает `false`, так как `Object` не является `User`.

Отметим два следствия из наличия общего базового класса:

- ❑ так как любой класс связан отношением «является» с `Object`, то в объявлениях переменных и параметров всегда можно указывать `Object` вместо конкретного типа;
- ❑ любой класс наследует всю функциональность базового класса `Object`.

Мы рассмотрим эти положения по очереди.

7.2.2. Использование отношения «является» применительно к `Object`

Поскольку любой тип автоматически наследует `Object`, мы можем передать любой тип туда, где ожидается объект; это относится и к значению `null`, и к функциям, и к булевым величинам. Например, в функции `doLogon(AuthService authService, String username, String password)` из главы 6 можно было бы явно проверять тип входного параметра:

```
doLogon(Object authService, String username, String password) {
  if (authService is AuthService) { ← Явно проверяется тип входного параметра
    return authService.auth(username, password); ← Использовать параметр authService,
  }                                              если переданный объект «является»
  else {                                       AuthService
    // возбудить исключение
  }
}
```

При таком подходе к написанию кода мы не зависим от модуля проверки типов, который гарантирует правильность переданных функции параметров. Ключевое слово `as` сообщает модулю проверки типов, что мы ожидаем переменную определенного типа. Например, предложение `return` в примере выше можно было бы записать и так:

```
if (authService is AuthService) {  
    return (authService as AuthService).auth(username, password);  
}
```

Используется

← ключевое слово **as**

Но иногда полезно иметь функцию, которая готова принять в качестве параметра значения разных типов, не реализующих общий интерфейс и не имеющих общего родительского класса. В примере ниже мы объявляем параметр типа `Object` и проверяем конкретный тип в самой функции:

```
processValue(Object someObject) {  
    if (someObject is String) {  
        // опущено  
    }  
    else if (someObject is int) {  
        // опущено  
    }  
    // и т. д...  
}
```

Предупреждение. Таким приемом лучше не пользоваться в открытом интерфейсе библиотеки, потому что он не дает никакой полезной документации ни пользователям, ни инструментам. Если вы все же вынуждены будете к нему прибегнуть, включите комментарий и объясните, почему вы так поступили.

Ниже в этой главе мы рассмотрим аннотацию типа `dynamic` — типа переменной или параметра в отсутствие явной информации о типе, — который имеет тонкое отличие от `Object`. При использовании `Object` мы явно заявляем, что программа, вызывающая функцию, *вправе* передать параметр любого типа. Если же информация о типе вообще не задана (то есть используется динамическая типизация), то мы не сообщаем явно, какой тип можно использовать. Употребляя тип `Object` в списке параметров, мы документируем тот факт, что пользователь действительно может передать значение *любого* типа.

7.2.3. Использование метода `toString()`, унаследованного от класса `Object`

В классе `Object` определен ряд методов и свойств, которыми могут пользоваться все классы. Наиболее употребительным из них является метод `toString()`, возвращающий строковое представление объекта.

При попытке использовать экземпляр класса как строку – например, передав его функции верхнего уровня `print()`, – вызывается метод `toString()` класса `Object`. В примере ниже мы вызываем `toString()` явно, но если бы мы этого не сделали, то Dart вызвал бы его автоматически.

Листинг 7.6. Метод `toString()` выводит текстовое представление объекта

```
print(1.toString());           ← 1
print("dart".toString());     ← dart
print(true.toString());       ← true
print(null.toString());       ← null
print(new Object().toString()); ← Instance of 'Object'
print(1);                     ← Вызов toString() подразумевается. Явно вызывать его необязательно
```

Особый интерес представляет вызов `toString()` для объекта типа `Object`. При этом возвращается строка `Instance of 'Object'`, содержащая все, что известно на данный момент. Если бы мы вызвали `toString()` для объекта типа `User`, то получили бы в ответ строку `Instance of 'User'`, потому что именно такая функциональность встроена в класс `Object`. Но при вызове `toString()` для списка чисел мы получили бы сам список чисел, а не строку `Instance of 'List'` потому что в классе `List` метод `toString()` переопределен.

Переопределить его можно и в собственном классе. Например, можно сделать так, что в классе `User` метод `toString()` будет выводить имя пользователя. Такое решение будет особенно полезно в сочетании с протоколированием, потому что теперь мы можем передать экземпляр `User` функции протоколирования или функции верхнего уровня `print()`, и та выведет содержательное сообщение. В примере ниже используется также функциональность, встроенная в класс `Object`, – за счет явного вызова `super.toString()`.

Листинг 7.7. Метод `toString()` выводит текстовое представление объекта

```
class User {
  String username;

  String toString() { ← Новая реализация toString()
    var myType = super.toString(); ← Вызывается toString() из родительского класса
    return "$myType: $username";   ← Возвращается "Instance of 'User': Alice"
  }
}
```

Подклассы `User`, в том числе `EnterpriseUser`, наследуют эту функциональность. Вызов `toString()` для экземпляра `EnterpriseUser` возвращает строку `Instance of 'EnterpriseUser': Alice`, гораздо более информативную, чем описание по умолчанию. Разумеется, никто не мешает переопределить метод `toString()` и в классе `EnterpriseUser` тоже.

7.2.4. Перехват обращений к методу `noSuchMethod()`

В классе `Object` реализован еще один полезный метод: `noSuchMethod()`. В отличие от `Java` или `C#` (но по аналогии со многими динамическими языками, например `Ruby` и `Python`), мы не ограничены использованием только тех свойств и методов, которые явно объявлены в классе.

Обнаружив обращение к методу или свойству, отсутствующему в определении класса, `Dart` сначала проверяет, есть ли такое в каком-либо родительском классе (на любом уровне иерархии наследования). Если нет, то `Dart` ищет в иерархии метод `noSuchMethod()`. Если в родительских классах он не объявлен явно, то вызывается метод `Object.noSuchMethod()`, который возбуждает исключение `NoSuchMethodError`.

Этим средством можно воспользоваться в иерархии класса `User`. Когда Алиса входит в систему, мы не всегда можем доверять данным, полученным от `EnterpriseService`, из-за возможных рассогласований в исходной системе. Разработчики `EnterpriseService` возвращают столько данных, сколько могут, а мы должны проверить, отвечают ли они нашим требованиям. По этой причине ни в абстрактном классе `User`, ни в конкретном классе `EnterpriseUser` нет метода `validate()`, но иногда он все же может понадобиться. На рис. 7.8 показано, как `Dart` обрабатывает вызов этого метода.

Когда мы вызываем `validate()`, `Dart` – будучи исполнен оптимизма и ожидая, что разработчик знает, что делает, – пытается этот метод выполнить. В конечном итоге `Dart` найдет в базовом классе `Object` реализацию метода `noSuchMethod()` по умолчанию, которая возбуждает исключение. Это исключение можно перехватить в блоке `try/catch`, как показано ниже:

```
try {
  user.validate("Alice");
  on NoSuchMethodError catch (err) {
    // обработать или игнорировать ошибку
  }
}
```

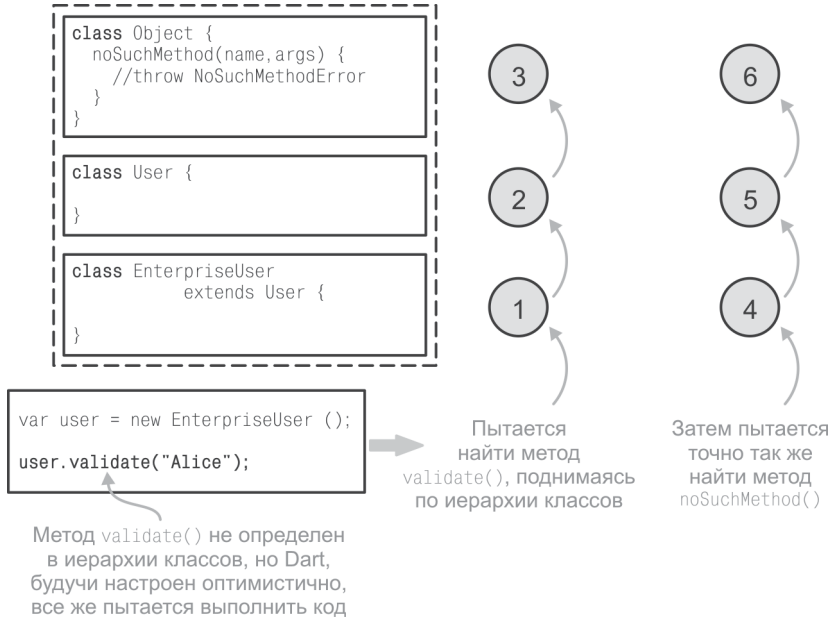


Рис. 7.8. Dart проверяет наличие вызванного метода во всех родительских классах и не находит его.

Тогда он ищет в иерархии наследования метод `noSuchMethod()`, который обнаруживается в базовом классе `Object`

Мы можем предотвратить возбуждение исключения `NoSuchMethodError`, реализовав собственный вариант `noSuchMethod()`. Это позволяет перехватить обращения к несуществующему методу и выполнить произвольный код, например проверить правильность данных.

Метод `noSuchMethod()` принимает два параметра: строку `String`, представляющую имя вызванного метода, и список `List` переданных ему параметров. Списки мы будем детально обсуждать в следующей главе, а пока достаточно знать, что нумерация элементов в них начинается с нуля и что для доступа к элементу нужно указать его индекс в квадратных скобках – как во многих других языках. В листинге ниже приведена реализация метода `noSuchMethod(String name, List args)`. Здесь мы печатаем имя метода и количество переданных ему аргументов.

Листинг 7.8. Реализация метода noSuchMethod()

```
class User {
  noSuchMethod(String name, List args) { ← Объявляется noSuchMethod()
    print("$name, ${args.length}");      ← Печатаются имя вызванного метода
  }                                       и количество переданных ему аргументов
}
```

При вызове `user.validate("Alice");` на консоли печатается строка `"validate, 1"`. Поток управления показан на рис. 7.9.

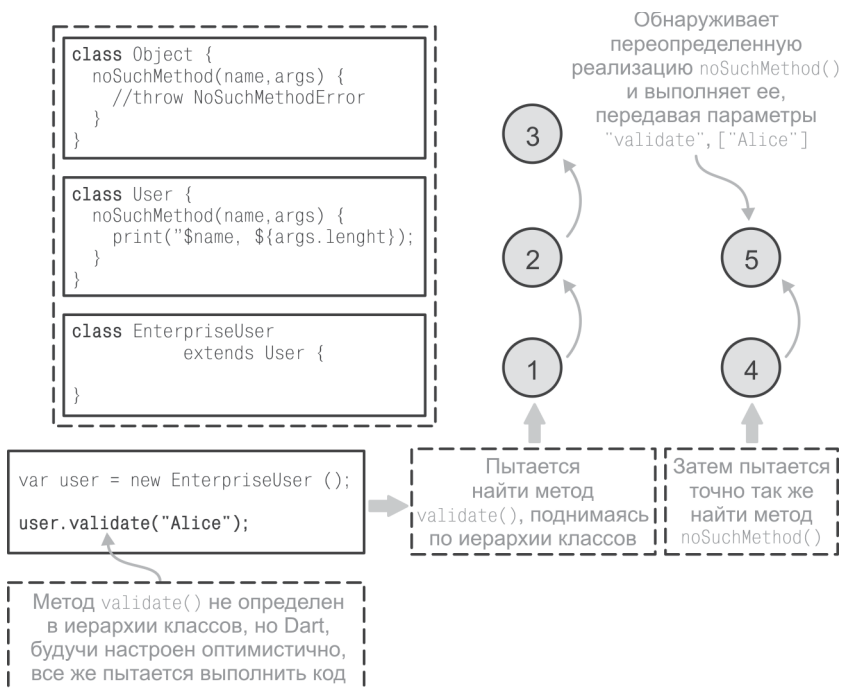


Рис. 7.9. Теперь метод `noSuchMethod()` найден и вызывается

Можно явно проверить имя метода и при необходимости все-таки возбудить исключение `noSuchMethodError`. Для этого нужно проанализировать значение параметра `name` и вызвать метод `super.noSuchMethod(name, args)`. При таком подходе перехватываются только интересные нас методы, а остальные игнорируются.

Метод `noSuchMethod()` может перехватывать также обращения к свойствам. Допустим, мы обратились к отсутствующему в классе `User` свойству `password`. Тогда можно было проигнорировать попытку его установки и вернуть строку звездочек при чтении. Если `noSuchMethod()` вызывается в результате обращения к свойству, то к имени поля добавляется префикс `get:` или `set:` в зависимости от того, какая операция – чтение или установка – производилась. Следовательно, обращение `print(user.password)` можно обработать в такой реализации метода `noSuchMethod()`:

```
noSuchMethod(name, args) {  
  if (name == "get:password") {  
    return "*****";  
  }  
  else if (name != "set:password") {  
    super.noSuchMethod(name, args);  
  }  
}
```

7.2.5. Прочая функциональность класса *Object*

В классе `Object` определена и другая функциональность, которую все остальные классы получают по умолчанию. Особенно интересен оператор сравнения на равенство `==`. В следующей главе мы обсудим, как можно переопределить этот и другие операторы в своих классах, но и реализация по умолчанию не вызывает удивления.

Оператор равенства в классе `Object` возвращает `true` или `false` в зависимости от того, совпадает данный объект с другим или нет. Иначе говоря, `true` возвращается, когда слева и справа от знака равенства находится один и тот же объект (рис. 7.10).

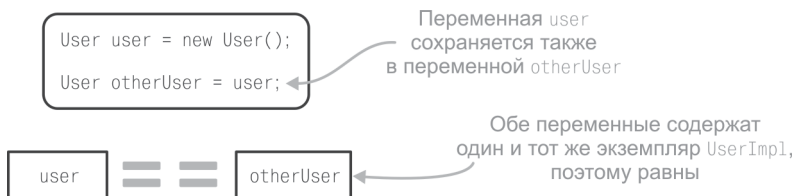


Рис. 7.10. Оператор равенства из класса `Object` позволяет сравнить два объекта

В следующей главе мы узнаем, как переопределить оператор равенства и другие операторы, но сначала следует поближе познакомиться с типом `dynamic`. Этот тип используется в Dart, когда явно не задано никакой информации о типе. У каждой переменной есть тип, который можно получить из свойства `dynamic`, определенного в классе `Object`.

Памятка

- Все «является» объектом.
 - В классе `Object` определены методы `toString()` и `noSuchMethod(name, args)`, которые можно переопределить в собственных классах.
 - Метод `noSuchMethod()` перехватывает обращения к неизвестным методам и свойствам.
-

7.3. Знакомство с типом `dynamic`

Последний тип, который мы рассмотрим, — `dynamic`. Разрабатывая библиотеку, например библиотеку `login_lib` для аутентификации, всегда полезно указывать информацию о типе в интересах разработчиков и инструментальных средств. Но в библиотеке, создаваемой для себя или в качестве прототипа, информацию о типе вполне можно опускать. Если информация о типе отсутствует, например когда переменная объявлена с помощью ключевого слова `var`, Dart применяет специальную аннотацию типа `dynamic`.

Тип `dynamic` концептуально похож на тип `Object` в том смысле, что все «является» `dynamic`. Поэтому разрешается использовать этот тип в объявлениях переменных, параметров и возвращаемых функциями значений. С точки зрения языка, указать тип `dynamic` — все равно, что не указать никакого типа; однако, явно задавая тип `dynamic`, вы сообщаете другим разработчикам о том, что специально хотели использовать этот тип, а не опускать задание типа вовсе. Ниже в этой главе мы рассмотрим данный момент подробнее. Но сначала взгляните на рис. 7.11, где показано, что тип `dynamic` подразумевается автоматически, когда никакой другой информации о типе не задано: слева мы видим, что используется тип `dynamic`, а справа — явный тип, например `String`.

Как работает отношение «является» с типом `dynamic`?

Все в Dart является объектом, однако любой класс, в том числе `Object`, также связан отношением «является» с типом `dynamic`. Это объясняется тем, что `dynamic` — базовый интерфейс, реализуемый любым классом, в частности `Object`. У интерфейса `dynamic` нет ни свойств, ни методов, он жестко зашит в код виртуальной машины и, в отличие от `Object`, ему невозможно унаследовать.

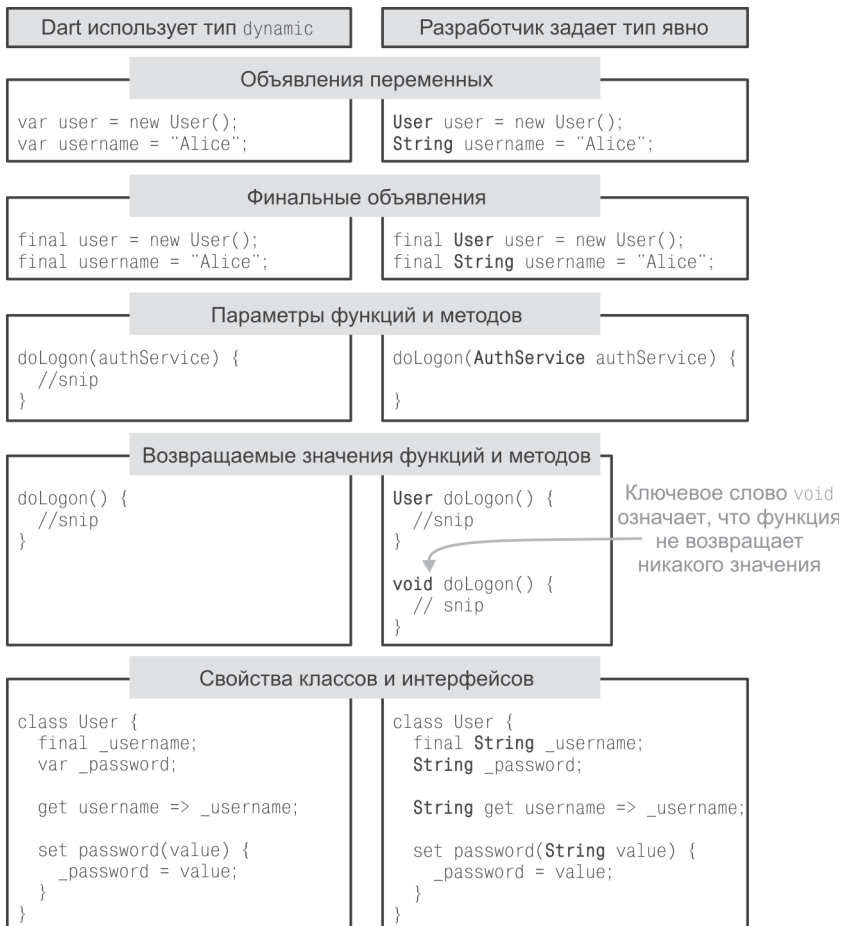


Рис. 7.11. Сравнение употребления динамического и строгого типа в Dart

7.3.1. Использование аннотации типа *dynamic*

Ключевое слово `dynamic`, обозначающее тип `dynamic`, можно указывать явно вместо иной информации о типе; обычно его употребляют, когда хотят недвусмысленно заявить о нежелании указывать тип. Здесь имеется тонкое отличие от типа `Object`, который используется,

чтобы заявить о приемлемости любого объекта. Читая чужой код, интерпретируйте намерения разработчика в части употребления `Object` или `dynamic` в соответствии с табл. 7.1.

Таблица 7.1. Как интерпретировать употребление типов `Object` и `dynamic`

Пример объявления функции	Какой вывод можно сделать
<code>doLogon(Object user);</code>	Разработчик сознательно разрешил передавать в функцию любой объект
<code>doLogon(dynamic user);</code>	Разработчик сознательно решил, что еще не знает, объект какого типа следует передавать в функцию
<code>doLogon(username);</code>	Разработчик еще не объявил, объект какого типа следует передавать в функцию (неявный <code>dynamic</code>)
<code>doLogon(User user);</code>	Разработчик объявил, что в эту функцию следует передавать объект типа <code>User</code>

На практике, однако, следует избегать явного задания типа `dynamic` или, по крайней мере, оставлять комментарий, поясняющий мотивы такого решения. Но у любого правила есть исключения, и мы поговорим о них, когда перейдем к изучению обобщенных типов в главе 8.

7.4. Резюме

В этой главе мы познакомились с механизмом наследования в Dart, который позволяет строить иерархии классов. `Object` является предком всех остальных классов и типов, в том числе и встроенных, например `String`, `int` и `null`. С другой стороны, тип `dynamic` используется в Dart на этапе выполнения, чтобы разрешить выполнение программы без аннотаций типов.

Памятка

- Ключевое слово `extends` означает, что класс наследует другому классу (своему родителю).
- Ключевое слово `super` позволяет вызывать члены (свойства и методы) родительского класса.
- Разрешается переопределять члены родительского класса, предоставляя собственную реализацию.
- Любой объект обладает методом `toString()`, с помощью которого можно предоставить дополнительную информацию, например при выводе сообщений в журнал.
- Метод `noSuchMethod()` базового класса `Object` можно использовать для перехвата обращений к несуществующим методам и свойствам.



- Аннотация типа `dynamic` представляет нетипизированные переменные и параметры в Dart.
-

Мы еще не закончили с классами! В следующей главе мы познакомимся с обобщенными типами и обсудим классы коллекций в Dart: `Collection`, `List` и `Map`. Вы также научитесь сами создавать подобные гибкие универсальные классы. Кроме того, мы рассмотрим механизм перегрузки операторов, с помощью которого можно писать по-настоящему самодокументированный код, изменяя семантику встроенных стандартных операторов.



Глава 8. Классы коллекций

В этой главе

- ❑ Работа с коллекциями.
- ❑ Построение типобезопасных обобщенных классов.
- ❑ Перегрузка операторов.

В двух предыдущих главах мы говорили об использовании классов и интерфейсов в приложениях и библиотеках, написанных на Dart. В этой, последней главе, посвященной классам, мы увидим, как сделать классы еще более гибкими и наглядными.

Мы начнем со встроенных классов коллекций, таких как `List` и `Map`, которые позволяют работать со списками. Благодаря обобщенным типам мы сможем сделать доступ к коллекциям типобезопасным и воспользоваться услугами модуля проверки типов. Для прямого доступа к элементам списков и словарей мы будем применять синтаксис индексатора. Мы также покажем, как создавать литеральные списки и словари из уже известных значений, и остановимся на методах JSON (JavaScript Object Notation), которые позволяют преобразовывать словари в строки и наоборот.

Далее мы рассмотрим, как сделать обобщенными собственные классы, то есть создавать не два почти идентичных класса, а один, строго типизированный, но тем не менее допускающий двойное использование.

Наконец, мы рассмотрим перегрузку операторов. Под этим понимается создание новой реализации стандартных операторов, например `>` (больше) и `<` (меньше), специализированной для конкретного класса. В результате работать с классом становится более удобно. Мы также вернемся к типу `Map` и изучим перегруженные в нем операторы доступа по индексу `[]` и `[]=`, позволяющие вам писать свои классы, которые реализуют интерфейс `Map` и могут быть преобразованы в формат JSON средствами встроенной библиотеки.

8.1. Работа с коллекциями данных

При разработке программного обеспечения часто приходится сталкиваться с задачами перемещения и обработки коллекции данных типобезопасным способом. Так, в примере службы аутентификации в главе 7 учетные данные Алисы передавались экземпляру класса `AuthService`, который возвращал объект `User`. В этом разделе мы разовьем данный пример, введя в рассмотрение две роли пользователя – Читатель и Администратор, – наделенные разными полномочиями. Если Алиса входит в блог, намереваясь написать статью, то система должна убедиться, что она имеет полномочия Администратора, необходимые для создания и редактирования статей. Все остальные пользователи являются Читателями, им разрешено только читать новости, добавлять свои комментарии и делиться ссылками. Для решения этой задачи нам понадобится иерархия классов, показанная на рис. 8.1.

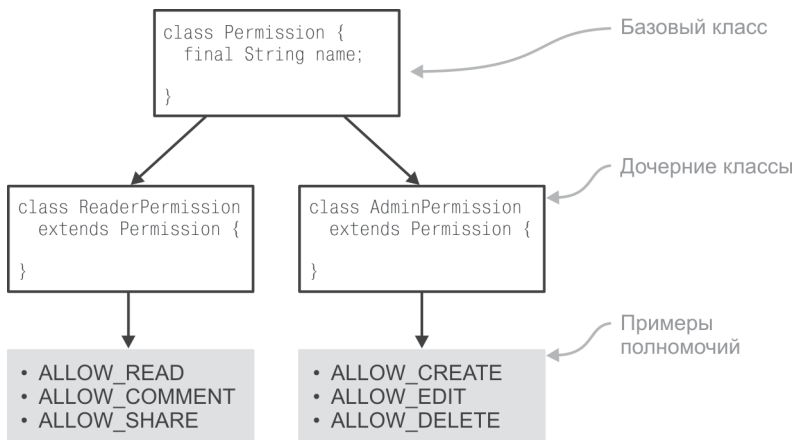


Рис. 8.1. Иерархия классов для ролей и полномочий

Когда Алиса входит в систему, та извлекает список объектов `Permission`, содержащий экземпляры классов `ReaderPermission` и `AdminPermission`, который работающее приложение записало в объект `User`, представляющий Алису. Приложение для ведения блога может справиться с этими объектами (полномочиями) и разрешить доступ к тем или иным частям системы, как показано на рис. 8.2.

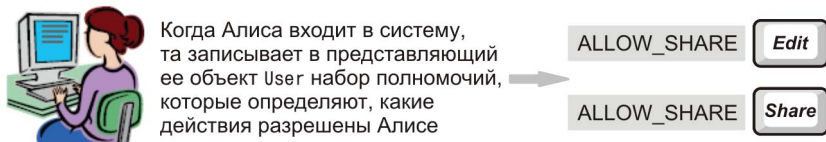


Рис. 8.2. В объект User, представляющий Алису, система записала набор полномочий

Полномочия — это константные экземпляры классов `ReaderPermission` и `AdminPermission`. Мы рассматривали константные классы в главе 5; напомним, что у такого класса должен быть константный конструктор (с ключевым словом `const` в объявлении), который инициализирует все поля на этапе компиляции. Создавая экземпляр константного класса с помощью ключевого слова `const` вместо `new`, мы можем быть уверены, что два экземпляра, все поля которых одинаковы, на самом деле идентичны. Этим можно воспользоваться, чтобы узнать, имеет ли пользователь определенное полномочие.

Мы должны создать объекты, представляющие отдельные полномочия, и сохранить их в статических переменных, на которые можно будет ссылаться из любого места приложения (статические переменные также рассматривались в главе 5). В листинге ниже приведен стереотипный код работы с полномочиями.

Листинг 8.1. Стереотипный код работы с полномочиями

```
class Permission { ← Базовый класс Permission
    final String name; ← Поле для хранения имени полномочия
    const Permission(this.name); ← Константный конструктор
}
                                Дочерний класс
                                ReaderPermission

class ReaderPermission extends Permission { ←
    const ReaderPermission(String name) : super(name);

    static final ReaderPermission ALLOW_READ =
    ↪          const ReaderPermission("ALLOW_READ");
    static final ReaderPermission ALLOW_COMMENT =
    ↪          const ReaderPermission("ALLOW_COMMENT");
    static final ReaderPermission ALLOW_SHARE =
    ↪          const ReaderPermission("ALLOW_SHARE");
}

class AdminPermission extends Permission { ← Дочерний класс AdminPermission
```

Создаются и сохраняются в статических переменных полномочия Читателя

```

const AdminPermission(String name) : super(name);

static final AdminPermission ALLOW_EDIT =
    const AdminPermission("ALLOW_EDIT");
static final AdminPermission ALLOW_DELETE =
    const AdminPermission("ALLOW_DELETE");
static final AdminPermission ALLOW_CREATE =
    const AdminPermission("ALLOW_CREATE");
}

```

Создаются и сохраняются
в статических переменных
полномочия Администратора

Теперь у нас есть стереотипный код, содержащий два типа полномочий (Администратор и Читатель), и мы можем приступить к его использованию.

8.1.1. Коллекции объектов

Алиса – экземпляр класса `User`, и в общем случае у объекта `User` может быть много полномочий. Иными словами, с Алисой должен быть ассоциирован список полномочий. В некоторых языках программирования для хранения списков применяются массивы, но в Dart нет специального типа `Array`, а есть интерфейсы `Collection` и `List`, причем `List` расширяет `Collection`. У интерфейса `List` имеется реализация по умолчанию, поэтому мы можем создавать его экземпляры с помощью ключевого слова `new`:

```

Collection permissionsCollection = new List();
List permissionsList = new List();

```

Ключевое слово `as`

В некоторых случаях желательно рассматривать конкретную переменную как экземпляр другого типа. Классический случай: имеется переменная типа `Collection`, но записывается в нее список. Это допустимо, однако в интерфейсе `Collection` нет метода `add()`.

Чтобы использовать переменную типа `Collection` как список, нам необходимо ключевое слово `as`. Оно сообщает модулю проверки типов, что мы намереваемся хранить в переменной типа `Collection` экземпляр `List`, то есть работать с коллекцией как со списком:

```

(permissionsCollection as List).add(... какое-то полномочие ...);

```

Различия между интерфейсами `Collection` и `List` мы обсудим чуть ниже. А пока добавим в класс `User` из главы 7 новое поле, которое будет содержать коллекцию полномочий, как показано на рис. 8.3. Когда Алиса входит в систему, в эту коллекцию помещаются ее полномочия.

1. Определение класса User

```
class User {  
    //snip... other properties  
  
    Collection permissions;  
  
    User() {  
        permissions = new List();  
    }  
}
```

Интерфейс Collection
можно использовать
для хранения списка
полномочий пользователя

Для создания экземпляра
списка полномочий используется
встроенная реализация
интерфейса List по умолчанию

2. Использование класса User в AuthService

```
class AuthService {  
    //snip... other methods  
  
    User login(username, password) {  
        User user = new User();  
        //snip ... logon code  
  
        (user.permissions as List).add(ReaderPermission.ALLOW_READ);  
        (user.permissions as List).add(AdminPermission.ALLOW_EDIT);  
        return user;  
    }  
}
```

После входа Алисы в систему
представляющий ее объект user
будет содержать полномочия,
определяемые бизнес-правилами

В свойстве permissions объекта user
хранятся экземпляры классов
ReaderPermission и AdminPermission

Рис. 8.3. Добавляем коллекцию полномочий в класс User.
Когда Алиса входит в систему, в эту коллекцию
помещаются ее полномочия

Методы интерфейса Collection

В интерфейсе Collection объявлен ряд обобщенных методов и свойств, применимых к любой коллекции объектов, например: получение длины коллекции, выяснение того, пуста она или нет, возврат подмножества элементов, удовлетворяющих некоторому условию. Эта возможность полезна, например, для возврата списка административных полномочий.

В листинге 8.2 приведен код функции, которая возвращает список административных полномочий из коллекции, применяя к каждому ее элементу ключевое слово `is`, которое возвращает `true`, если этот элемент является объектом класса `AdminPermission`. Функция `filter()` из интерфейса `Collection` принимает в качестве параметра анонимную функцию, вызываемую для каждого элемента (полномочия).

Если эта функция возвращает `true`, то полномочие добавляется в возвращаемый результат.

Совет. Анонимная функция, которая вызывается для каждого элемента коллекции и возвращает `true` или `false`, называется предикатом.

Листинг 8.2. Возврат списка объектов класса `AdminPermission`

```
Collection extractAdminPermissions(User user) {
  return user.permissions.filter( (currentPermission) {
    return currentPermission is AdminPermission;
  });
}
```

Возвращает коллекцию, которая является списком полномочий

Предикат возвращает `true`, если `currentPermission` – экземпляр класса `AdminPermission`. В ответ `currentPermission` добавляется в возвращаемый результат

Передав функции `extractAdminPermissions()` объект `User`, представляющий Алису, мы получаем новый объект `Collection`, который содержит все административные полномочия пользователя.

Еще один полезный метод интерфейса `Collection` – функция `some()`, которая позволяет узнать, содержится ли в коллекции элемент (полномочие), удовлетворяющий некоторому условию. Эта функция принимает предикат и возвращает `true`, если предикат вернул `true` хотя бы для одного элемента коллекции. Например, чтобы узнать, есть ли у Алисы полномочие `ALLOW_CREATE`, мы можем вызвать `some()`, как показано ниже, и запомнить результат в переменной `containsAllowCreate`:

```
bool containsAllowCreate = user.permissions.some(
  (currentPerm) => currentPerm == AdminPermission.ALLOW_CREATE;
);
```

Предикат, записанный в короткой форме, вызывается для каждого элемента коллекции. Функция `some()` возвращает `true`, если предикат вернул `true` хотя бы для одного элемента

Обход коллекции

Интерфейс `Collection` наследует интерфейсу `Iterable`, который позволяет использовать конструкцию `for( in )`. Синтаксис этой конструкции такой же, как в JavaScript, а семантически она аналогична предложениям `foreach( in )` в C# и `for( : )` в Java. Эта конструкция позволяет обойти коллекцию, поочередно присваивая каждый ее элемент переменной, как показано на рис. 8.4.

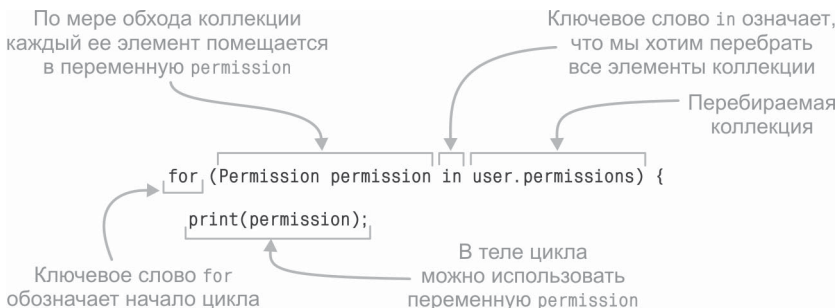


Рис. 8.4. Конструкция `for (... in ...)` позволяет перебрать все элементы коллекции

Конструкция `for ( in )` дает удобный способ обойти все элементы списка. Чтобы досрочно выйти из цикла, например после обнаружения первого подходящего полномочия, применяется ключевое слово `break`:

```
for (Permission permission in user.permissions) {
    if (permission is AdminPermission) { ← Найден первый объект типа AdminPermission
        break; ← Сразу выходим из цикла
    }
}
```

Обход коллекции вручную

Интерфейс `Iterable` возвращает интерфейс `Iterator`, который используется автоматически в конструкции `for (...in...)`, но его свойство `hasNext` и метод `next()` можно вызывать и явно для обхода коллекции вручную. Свойство `hasNext` возвращает `true`, если имеется следующий элемент; метод `next()` возвращает следующий элемент и перемещает итератор на одну позицию вперед. Если итератор уже находился за последним элементом, то возбуждается исключение `StateError`. В примере ниже из коллекции извлекаются первые два элемента. Метод коллекции `iterator()` возвращает новый итератор, который позиционирован перед первым элементом.

Листинг 8.3. Извлечение первых двух элементов коллекции с помощью итератора

```
Iterator iterator = user.permissions.iterator(); ← Получаем новый итератор,
var perm1 = null; | Создаем переменные | позиционированный перед первым
var perm2 = null; | для хранения полномочий | элементом
```

```

if (iterator.hasNext) {
    perm1 = iterator.next();
}
if (iterator.hasNext) {
    perm2 = iterator.next();
}

```

← Проверяем, есть ли следующий элемент
 ← Возвращаем следующий элемент и сдвигаем итератор вперед
 | Проверяем наличие
 | и возвращаем второй элемент

8.1.2. Использование конкретных реализаций интерфейса *Collection*

Интерфейс *Collection* в сочетании с интерфейсами *Iterable* и *Iterator* – мощное средство, но создать экземпляр *Collection* непосредственно невозможно. Напомним, что интерфейс – это способ описания методов и свойств, которые должны присутствовать в классе, реализующем интерфейс. В базовых библиотеках Dart есть несколько реализаций интерфейса *Collection*: *List*, *Queue* и *Set*. Каждая из них обладает своими особенностями. На рис. 8.5 показаны взаимоотношения интерфейса *Collection* с его потомками (*List*, *Queue* и *Set*) и родителем (*Iterable*).

В Dart нет типа *Array*, но, как уже было сказано, есть тип *List*. Списки могут быть как фиксированного, так и переменного размера (расширяемые). Для списков фиксированного размера разрешено помещать элементы в предопределенные позиции с помощью оператора доступа по индексу `[]`. С другой стороны, для списков переменного размера разрешены вставка и удаление элементов.

Создание списков

На рис. 8.4 показаны четыре способа создания списков. Проще всего создать литеральный список, применяя нотацию с квадратными скобками; при этом порождается инициализированный список переменного размера. Второй способ – воспользоваться конструктором `List()`. Конструктор без параметров создает пустой расширяемый список. Если же передать параметр, то получится список фиксированного размера с указанным в параметре количеством элементов, причем все они равны `null`. Доступ к элементам производится по числовому индексу, нумерация индексов начинается с нуля.

Наконец, имеется конструктор `List.from(...)`, который позволяет создать расширяемый список из объекта любого класса, реализующего интерфейс *Iterable*. Эта возможность полезна для преобразования существующего списка фиксированного размера в список переменного размера.



Рис. 8.5. `Collection` – базовый интерфейс Dart для работы с коллекциями объектов, но создать можно только экземпляр реализующего его класса: `List`, `Queue` или `Set`

Листинг 8.4. Различные способы создания списка

```

// Создается литеральный расширяемый
// список с известным списком значений
List literal = [ReaderPermission.ALLOW_READ, ReaderPermission.ALLOW_SHARE];

literal.add(ReaderPermission.ALLOW_COMMENT); // В список добавляются новые элементы

print(literal.length); // Печатается: 3

List growable = new List(); // Создается пустой расширяемый список

print(growable.length); // Печатается: 0
  
```

```
growable.add(ReaderPermission.ALLOW_READ);
growable.add(ReaderPermission.ALLOW_SHARE);
```

В список добавляются
новые элементы

```
List fixedSize = new List(2);
print(fixedSize.length);
```

← Создается список фиксированного размера
← Печатается: 2

```
fixedSize[0] = ReaderPermission.ALLOW_READ;
fixedSize[1] = ReaderPermission.ALLOW_SHARE;
```

Для помещения элементов в пустые
позиции используется доступ по индексу

Если раскомментировать эту
строку, то будет возбуждено
исключение `UnsupportedError`,
потому что список нерасширяемый

```
// fixedSize.add(ReaderPermission.ALLOW_COMMENT);
```

←

Создается расширяемый список из другого списка

```
List fromOther = new List.from(fixedSize);
fromOther.add(ReaderPermission.ALLOW_COMMENT);
```

← Теперь в список можно добавлять элементы

Любой элемент списка можно читать и записывать с помощью операторов доступа по индексу `[]` и `[]=`. Оператор `[]` позволяет прочитать значение элемента в позиции с указанным индексом (нумерация начинается с нуля), например `var permission = growable[2];`, а оператор `[]=` — модифицировать значение по индексу, например `growable[2] = permission`. Эти операторы важны, мы снова встретимся с ними, когда будем говорить о словарях, и еще раз — когда дойдем до перегрузки операторов в собственных классах. Ко всему прочему, они дают еще один способ обхода списка в дополнение к определенному в интерфейсе `Collection` (не забывайте, что `List` «является» `Collection`). Можно перебирать элементы по индексу в цикле `for`, синтаксис которого такой же, как у всех языков, ведущих происхождение от `C`, в частности `Java`, `JavaScript` и `C#`:

```
for (int i = 0; i < fixedSize.length; i++) {
  Permission permission = fixedSize[i];
}
```

Остальные два встроенных типа коллекций, `Queue` и `Set`, не позволяют напрямую обращаться к своим элементам, но их можно преобразовать в списки с помощью конструктора `List.from()`.

Создание множеств

Множество — это частный случай коллекции, в которой не может быть дубликатов. У класса `Set` есть специфические методы, например `isSubsetOf(collection)` и `contains(value)`, а также конструктор `Set.from()`, позволяющий создавать экземпляр `Set` из любого объекта ти-

па `Iterable`. В следующем примере создается множество, содержащее только два элемента:

```
Set set = new Set();
set.add(ReaderPermission.ALLOW_READ);
set.add(ReaderPermission.ALLOW_READ);
set.add(ReaderPermission.ALLOW_SHARE);
print(set.length);
```

Добавление двух одинаковых элементов дает тот же результат, что и добавление одного элемента

← Множество содержит всего два различных элемента

Создание очередей

Очереди полезны, когда нужно создать коллекцию, обслуживаемую по принципу «первым пришел, первым ушел», например когда одна часть программы добавляет элементы в список, а другая должна забирать их из списка в том же порядке, в каком они были добавлены. Метод `addLast()` добавляет элемент в конец очереди, а метод `removeFirst()` извлекает из ее начала. В примере ниже мы добавляем два элемента в конец очереди и извлекаем один из начала:

```
Queue queue = new Queue();
queue.addLast(ReaderPermission.ALLOW_READ);
queue.addLast(ReaderPermission.ALLOW_SHARE);
var permission = queue.removeFirst();
print(queue.length);
```

Добавляются два элемента в конец очереди

← Первый элемент, `ALLOW_READ`, извлекается из очереди

← В очереди остался один элемент

Доступ по индексу к элементам множества и очереди – как для списка – невозможен. Но можно перебирать элементы с помощью методов интерфейса `Collection`, а в случае очереди также пользоваться дополнительными методами `addLast()` и `removeFirst()`.

8.1.3. Создание специализированных коллекций с помощью обобщенных типов

У интерфейса `Collection` есть одна проблема: в коллекцию можно добавлять объекты любого типа. То есть в результате можно получить коллекцию, содержащую вперемежку экземпляры классов `ReaderPermission`, `String` и `int`, например:

```
List permissions = new List();
permissions.add(ReaderPermission.ALLOW_READ);
permissions.add("ALLOW_WRITE");
permissions.add(123456789);
```

Правильно: в список добавляется полномочие

← Неправильно: добавляется строка

← Неправильно: добавляется целое

Такого рода код, скорее всего, неправилен; список должен содержать только объекты `Permission`, а не все, что угодно. Исправить это можно, например, создав класс `PermissionList`, наследующий `List` и включив в него методы вида `add(Permission item)`, которые принимают только объекты типа `Permission`. Но в типичном приложении обычно бывает много списков, содержащих значения разных типов, у которых нет никакого общего базового класса, кроме `Object`. Было бы непрактично (и расточительно) писать различные реализации списка для каждого возможного типа.

По счастью, в системе типов Dart предусмотрены обобщенные типы, которые можно конкретизировать типами, не связанными отношением наследования, как, например, в случае списка объектов типа `Permission`, `String` или `int`.

Все типы в иерархии интерфейса `Collection` обобщенные, что позволяет использовать их для хранения только объектов конкретного типа, например `Permission`. Иными словами, мы *можем* создать список `Permission`, и модуль проверки типов в Dart выдаст ошибку при попытке случайно добавить в такой список строку `String`.

Обобщенные классы записываются в виде `Name<T>`, где `Name` — имя класса или интерфейса, а `T` — список параметрических типов через запятую. Классы коллекций, которые мы видели выше, имеют обобщенную сигнатуру, показанную на рис. 8.6, где `E` — параметрический тип элемента.

Конкретизация обобщенного типа

Чтобы воспользоваться обобщенным типом, нужно указать в объявлении типа и в конструкторе, какой тип подставить вместо параметрического. Например, мы знаем, что процедура входа в систему формирует список полномочий, и объявляем его следующим образом:

```
List<Permission> permissions = new List<Permission>(); ← Параметрический тип конкретизируется типом Permission
permissions.add(ReaderPermission.ALLOW_READ);
```

Теперь все методы списка работают так, будто мы написали специальный класс `PermissionList`, который может содержать только объекты типа `Permission`. Все методы коллекции либо ожидают объект или список объектов `Permission` на входе, либо возвращают объект типа `Permission` на выходе. В табл. 8.1 приведены сигнатуры некоторых методов интерфейса `List` и показано, что модуль провер-

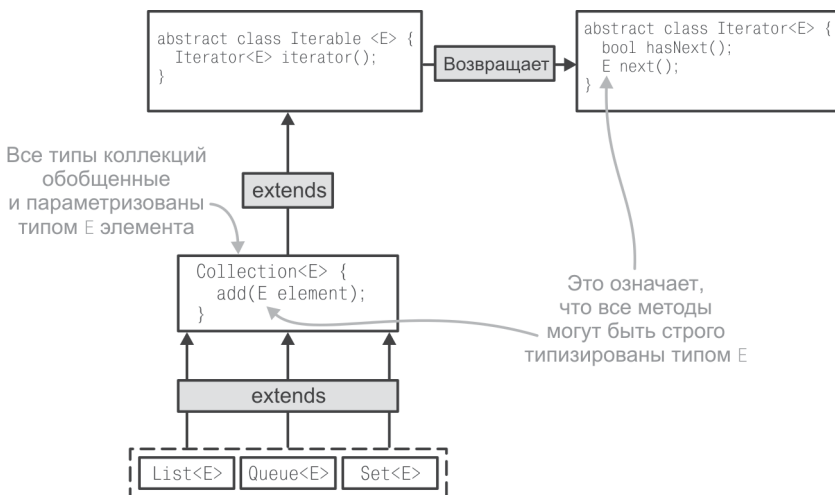


Рис. 8.6. Классы коллекций – обобщенные типы, параметризованные типом E, вместо которого можно подставить конкретный тип

ки типов Dart ожидает увидеть, когда в определении конструктора указан конкретный тип `Permission`.

Таблица 8.1. Фактические сигнатуры методов и интерпретация их модулем проверки типов

Фактическая сигнатура метода	Что ожидает увидеть Dart
<code>void add(E value);</code>	<code>void add(Permission value);</code>
<code>E next();</code>	<code>Permission next();</code>
<code>Collection<E> filter(bool f(E element));</code>	<code>Collection<Permission> filter (bool f(Permission element));</code>

Все сказанное справедливо и для любых других классов. Например, можно создать список `String` или очередь `int`, конкретизировав параметрический тип в конструкторе:

```

List<String> stringList = new List<String>();
stringList.add("Я строка");
Queue<int> intQueue = new Queue<int>();
intQueue.addLast(12345);
  
```

Если вы попытаетесь добавить `int` в список `String`, то модуль проверки типов выдаст такое же предупреждение, как при попытке передать `int` в функцию, ожидающую `String`, потому что метод `add()` класса `List` теперь ожидает получить строку.

У обобщенных типов есть еще одно достоинство: они позволяют проверять типы в сигнатурах методов, написанных нами. Допустим, у нас есть функция `extractAdminPermissions()`, которая принимает список `Permission`, а возвращает список `AdminPermission`. Рассмотрим следующие два варианта ее объявления:

```
Collection extractAdminPermissions(Collection permissions) {
    // тело функции опущено
}
```

← В этом случае
используются
нетипизированные
коллекции

```
Collection<AdminPermission> extractAdminPermissions(
    Collection<Permission> permissions) {
    // тело функции опущено
}
```

← А в этом дополнительно
производится проверка типа

В первом варианте мы можем передать функции `extractAdminPermissions()` любую коллекцию, в том числе коллекцию `String` или `int`, и модуль проверки не предупредит о возможной ошибке. Во втором варианте функция `extractAdminPermissions()` ожидает получить коллекцию объектов `Permission`, а вызывающая программа знает, что будет возвращена коллекция `AdminPermission`, и может быть уверена, что в него не попадут объекты `ReaderPermission`.

Так чем же «является» список?

В двух предыдущих главах мы с помощью ключевого слова `is` определяли, «является» ли переменная экземпляром определенного типа. Например, класс `AdminPermission`, который расширяет `Permission`, «является» `Permission`, но не «является» `ReaderPermission`. В следующем примере возвращается ожидаемый результат:

```
var p = AdminPermission.ALLOW_EDIT;
print(p is Permission);
print(p is AdminPermission);
print(p is ReaderPermission);
```

Верно

← Неверно: AdminPermission не наследует ReaderPermission

Этот принцип применим и к обобщенным типам. Список `AdminPermission` «является» списком `Permission`, потому что каждый эле-

мент списка «является» `Permission`. Следуя той же логике, он не «является» списком `ReaderPermission`:

```
var adminList = new List<AdminPermission>();
print(adminList is List);
print(adminList is List<Permission>);
print(adminList is List<AdminPermission>);
print(adminList is List<ReaderPermission>);
```

Верно

← Неверно: `AdminPermission` не наследует `ReaderPermission`

Этот код работает, как и ожидается, но возникает вопрос: почему модуль проверки типов считает, что `adminList is List`? Это логично, потому что список чего-то конкретного все равно остается списком, но откуда Dart это знает, если информация о типе отсутствует? За кулисами непараметризованный обобщенный тип – не что иное, как тип `dynamic`. Как мы выяснили в главе 7, тип `dynamic` представляет любой класс, то есть любой класс «является» `dynamic`.

Таким образом, списки объектов любого типа, в том числе неизвестного, всегда являются списками объектов типа `dynamic`:

```
var dynamicList = new List<dynamic>();
print(dynamicList is List);
```

`List<dynamic>` – это просто `List`


```
var notSpecifiedList = new List();
print(notSpecifiedList is List<dynamic>);
print(listOfAdmin is List<dynamic>);
```

Список всего, чего угодно, – это также `List<dynamic>`

И последний вопрос касается литеральных списков, потому что у них имеется специальный синтаксис, позволяющий Dart создавать списки известных значений (мы это уже видели). Как определить строго типизированный список известных значений? Это можно сделать с применением обобщенного синтаксиса `<имя типа>[элемент, элемент, ...]`, например:

```
var permissions =
↳ <Permission>[AdminPermission.ALLOW_CREATE, AdminPermission.ALLOW_EDIT];
```

8.1.4. Хранение списков пар ключ–значение в обобщенных словарях

Последним из встроенных обобщенных типов мы рассмотрим словарь `Map<K, V>`. Он используется для хранения списков пар ключ–

значение; при этом ключ служит индексом для доступа к значению. Мы можем провести аналогию с доступом к элементам списка по числовому индексу, например `myList[1]`, только теперь индекс является не числом, а чем-то иным (обычно строкой `String`). Параметрические типы `K` и `V` представляют, соответственно, типы ключа и значения. Они не обязаны совпадать и, как и для рассмотренных выше обобщенных типов, могут быть вообще проигнорированы (в таком случае мы получаем тип `Map<dynamic, dynamic>`).

Когда Алиса входит в систему, для нее создается объект `User`. То же самое происходит при входе Боба или Чарли. Но не исключено, что получение объекта `User` от корпоративной системы – накладная операция. Тогда можно реализовать простой кэш в виде словаря, ключом которого является имя пользователя, а значением – ассоциированный с ним объект `User`. Поскольку для интерфейса `Map` имеется класс реализации по умолчанию, то словари можно создавать и использовать, как показано в листинге ниже.

Листинг 8.5. Создание и использование словаря объектов `User`

	Создается новый словарь
<code>Map<String, User> userMap = new Map<String, User>();</code>	← с ключами типа <code>String</code>
	и значениями типа <code>User</code>
<code>userMap["aliceKey"] = new User("Alice");</code>	В словарь вставляются элементы с помощью индексатора
<code>userMap["bobKey"] = new User("Bob");</code>	
<code>User aliceUser = userMap["aliceKey"];</code>	Элементы читаются из словаря с помощью индексатора
<code>User bobUser = userMap["bobKey"];</code>	
	Чарли отсутствует в словаре, поэтому
<code>User charlieUser = userMap["charlieKey"];</code>	← переменная <code>charlieUser</code> равна <code>null</code>

Попытка доступа к несуществующему ключу, например `charlieKey` в примере выше, не приводит к исключению, как в других языках. Просто возвращается `null`.

Создание литеральных словарей

Словари, как и списки, можно создавать, имея заранее известный список ключей и значений. Для задания литерального словаря применяются фигурные скобки `{}`, внутри которых перечисляются пары ключ–значение через запятую, как показано на рис. 8.7.

Именно в таком виде списки пар ключ–значение записываются в формате JSON (JavaScript Object Notation). В библиотеке `dart:json` имеются методы для преобразования строки в словарь и наоборот.

```
Map myMap = {"key1": "value1", "key2": "value2"};
```

Словарь определяется
как список разделенных запятыми
пар ключ:значение

Рис. 8.7. В Dart словарь можно определить как список пар ключ–значение

Код, приведенный в следующем листинге, преобразует строку, содержащую список пар «имя пользователя – дата последнего входа в систему», в словарь, а словарь – обратно в строку.

Листинг 8.6. Использование библиотеки `dart:json` для преобразования строки в словарь и наоборот

```
import "dart:json";

main() {
    var jsonString = '{"charlieKey': '2012-07-23', 'aliceKey': '2012-08-16'}';
    Map lastLogonMap = JSON.parse(jsonString);
    print(lastLogonMap["charlieKey"]);
    jsonString = JSON.stringify(lastLogonMap);
}
```

Создается строка в форме JSON
← JSON разбирается и преобразуется в объект Map
← Доступ к значению в словаре по его ключу
← Обратное преобразование словаря в строку

Совет. Преобразование JSON-строки в словарь и обратно очень часто встречается в веб-приложениях при обмене данными между клиентом и веб-службой. Мы будем подробно рассматривать эту тему в главе 11.

Доступ к коллекциям ключей и значений

В связи с вышеизложенным возникает еще один вопрос. Допустим, что мы сохранили ключ `charlieKey`, значением которого является `null`. Как понять, есть такой ключ в словаре или нет?

К счастью, в интерфейсе `Map` предусмотрены методы `containsKey(K key)` и `containsValue(V value)`, позволяющие узнать, есть ли в словаре указанный ключ или значение. Можно также обойти всю коллекцию ключей и получить соответствующие им значения:

```
for (String key in userMap.keys) {
    User value = userMap[key];
    print(value);
}
```

← Обходим коллекцию ключей
← Получаем значение текущего ключа
← Печатаем строковое представление значения `value.toString()`

Вставка элементов в словарь

Оператор `[ ]` = позволяет вставлять в словарь новые элементы (как в предыдущем примере), но при этом ранее существовавший ключ (если он был) перезаписывается. Иногда бывает нужно вставить в словарь новый элемент, а не заменить старый. Например, можно для каждого пользователя хранить список дат, когда он заходил в систему в виде `Map<String, List<Date>>`. Эта запись означает, что строке, содержащей имя пользователя, ставится в соответствие список объектов `Date`.

Когда Боб входит в систему (причем не в первый раз), мы хотим добавить текущую дату в уже имеющийся список. А когда в систему впервые входит Чарли, мы создаем новый список дат. Это показано на рис. 8.8.

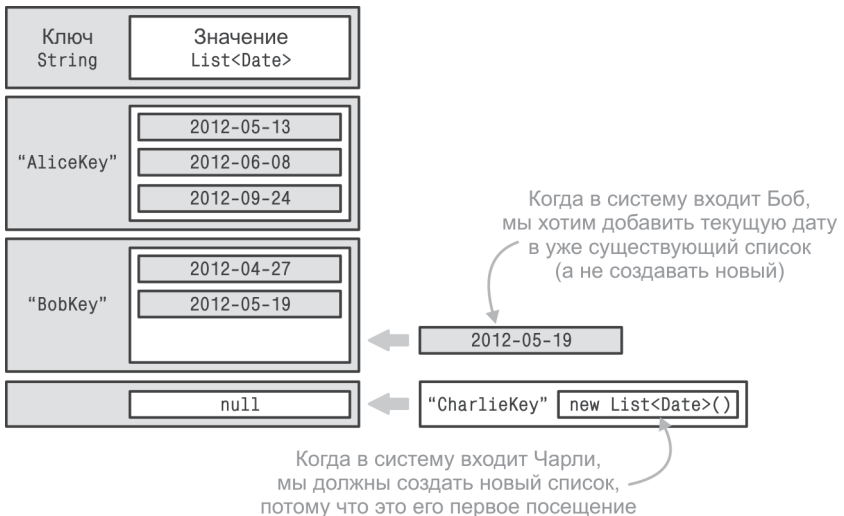


Рис. 8.8. Мы хотим создавать новую запись, только если такого ключа еще не существует

Проблема использования индексатора в виде `userLogons["bobKey"] = new DateList();` заключается в том, что всегда создается новый пустой список, в том числе и для Боба, и, значит, вся история его посещений стирается. С другой стороны, предложение `userLogons["charlieKey"].add(new Date.now());` приведет к исключению доступа по нулевому указателю, так как еще не существует списка `List<Date>`, в который можно было бы вставить дату методом `add()`.

Можно было бы выполнить проверку `containsKey("bob")`, но в интерфейсе `Map` имеется альтернатива в виде метода `putIfAbsent()`. Этот метод добавляет в список новый элемент, только если указанного ключа еще не существует. Он принимает в качестве параметров ключ и функцию, и, если ключ отсутствует в списке ключей, то он вставляется в словарь, а в качестве ассоциированного с ним значения берется значение, возвращенное функцией. В следующем листинге показано, как это работает.

Листинг 8.7. Использование метода `putIfAbsent()`

```

var userLogons = new Map<String, List<Date>> >();
userLogons.putIfAbsent("charlie", () => new List<Date>());
userLogons["charlie"].add(new Date.now());
```

Создается словарь со строковыми ключами. Значением ключа является список дат. Когда Чарли входит в систему, создается новый список дат, но только в том случае, если он еще не существует.

Получаем список дат по ключу "charlie" и добавляем текущую дату

Этот способ позволяет обращаться к свойствам объекта-значения в словаре, не заменяя его.

Раздел получился длинным, но обобщенные типы и коллекции – мощные средства Dart, и, как это присуще супергероям, их мощь возрастает, когда они действуют совместно. Кстати, обобщенными могут быть не только типы коллекций. В следующем разделе мы увидим, как создать собственный обобщенный класс, параметризованный типами, вместо которых пользователь сможет подставить что-то конкретное.

Памятка

- Коллекции создаются с помощью конкретных классов `List`, `Queue` и `Set`.
- В Dart нет типа `Array`, но вместо него можно использовать тип `List`. Списки бывают фиксированного или переменного размера.
- Все типы коллекций обобщенные, конкретизировать тип можно с помощью нотации `<T>` в типе конструктора. Например, предложение `new List<String>()` создает список строк (объектов типа `String`).
- К элементам списка можно обращаться по индексу. Нумерация начинается с нуля, поэтому `myList[2]` – третий элемент списка.
- Словарь содержит список пар ключ–значение. Для доступа к элементам словаря также используются индексомеры, но индексом является ключ. Например, чтобы поместить объект `Date` в словарь `Map<String, Date>`, можно написать

```
myMap["aliceKey"] = new Date.now();
```

8.2. Создание обобщенных классов

Мы видели, что обобщенные типы полезны, когда нужно обеспечить строгую типизацию классов, не имеющих общего базового класса. И это хорошо; действительно, допустим, что начальник увидел написанный вами класс `User` и пожелал перенести его в другую систему для управления табелями учета рабочего времени, где вместо `Permission` уже существует класс `Role`, определяющий роль пользователя. Хотелось бы, чтобы когда Алиса входит в блог, ее объект `User` содержал список полномочий `Permission`, а после входа в систему управления табелями – список ролей `Role`.

Как выясняется, роли и полномочия – по существу одно и то же, только представляются разными классами. Мы переименовываем список `User.permissions` в `User.credentials` и начинаем размышлять, как с помощью строгой типизации отличить список ролей от списка полномочий.

Первое, что приходит в голову, – создать подклассы `RolesUser` и `PermissionsUser` класса `User`, реализовав в каждом из них свой метод `add()` и др. (рис. 8.9). Но, по счастью, перед тем как встать на этот путь, мы вспоминаем об обобщенных типах и решаем, что ситуация идеальна для создания обобщенного класса пользователя.

8.2.1. Определение обобщенного класса

Для определения обобщенного класса необходимо включить параметрический тип в его объявление, например: `class User<C> { ... }`. Рекомендуется обозначать этот тип одной буквой, потому что такое соглашение позволяет легко опознать параметрические типы. Впрочем, следовать ему необязательно – можно было бы написать `class User<Credential> { ... }`, но такое имя больше напоминает реальный тип, а не параметрический. В табл. 8.2 приведены буквы, которыми часто обозначаются параметрические типы. Никто не заставляет вас использовать именно эти буквы, да и интерпретацию им вы можете придумать совершенно иную, но для этого хорошо бы иметь веские основания. Таковым может, например, служить хождение специальной терминологии в предметной области. Тогда буква `E`, которая обычно означает `Element`, могла бы относиться к ребру (`Edge`) графа.

Выбрав имя класса и имя параметрического класса `<C>`, мы можем употреблять последнее в списках параметров, возвращаемых типах и телах методов и свойств.

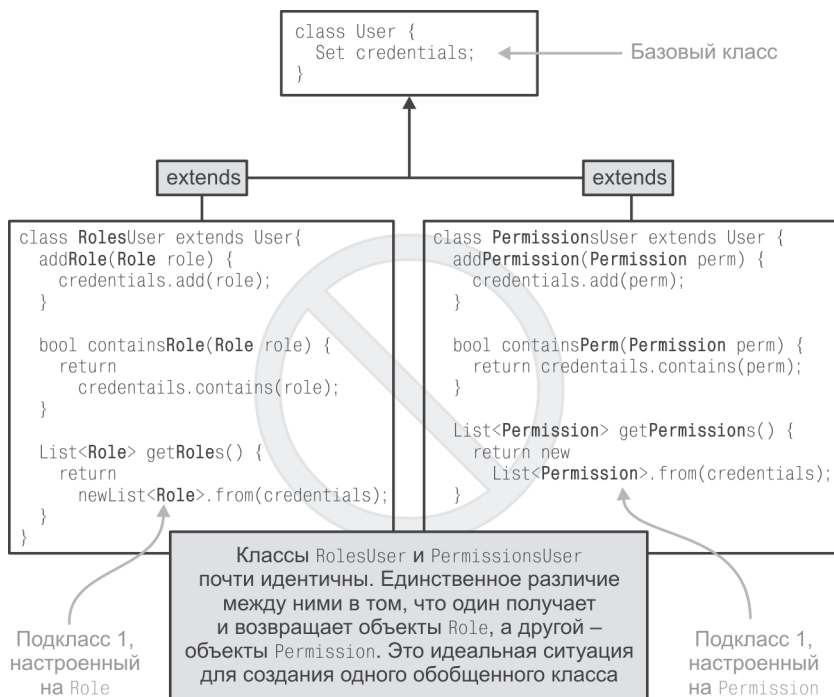


Рис. 8.9. Поймав себя на создании похожих классов, в которых используются – для одной и той же цели – незначительно отличающиеся объекты, вы можете применить обобщенные типы

Таблица 8.2. Некоторых часто употребляемые буквы для обозначения параметрических типов

Обозначение параметрического типа	Типичная интерпретация
<T>	Тип
<E>	Элемент
<K>	Ключ
<V>	Значение

В листинге ниже показано возможное определение обобщенного класса User, заменяющего классы RolesUser и PermissionsUser.

Листинг 8.8. Класс `User`, параметризованный классом прав

`class User<C> {` ← Определяется имя обобщенного класса с параметрическим типом `<C>`

List<C> credentials; ← Объявляется список, параметризованный типом C

```
User() {
    credentials = new List<C>();
}
```

← Конструктор создает новый экземпляр
списка объектов типа C

```
addCredential(C credential) {
    this.credentials.add(credential);
}
```

```
bool containsCredential(C credential) {
    return this.credentials.some((item) => item == credential);
}
```

```
List<C> getCredentialsList() {
    return new List<C>.from(credentials);
}
```

Возвращается
список объектов
типа C

Гарантируется, что
методам можно
передавать только
объекты типа C

8.2.2. Использование своего обобщенного класса

Свой собственный обобщенный класс можно использовать точно так же, как любой другой. При этом есть уверенность, что модуль проверки типов обнаружит ошибки типизации (например, если в процессе переноса вы что-то упустили и пытаетесь получить объект `Permission`, когда имели в виду `Role`).

Появляется также возможность повторно использовать класс `User` в других случаях, о которых вы поначалу и не думали, например если права представлены строками или целыми числами. В следующем листинге показано несколько способов типобезопасного использования нового класса.

Листинг 8.9. Типобезопасное использование обобщенного класса User

Создается объект
User<Permission> permissionUser = new User<Permission>(); ← класса User, параметри-
зованного типом Permission


```
User<Role> roleUser = new User<Role>();  
roleUser.addCredential(const Role("ADMIN"));  
print( roleUser.contains(const Role("ADMIN")) );
```

Создается объект класса `User`, параметризованного типом `Role`, и его методы вызываются типобезопасным способом

```
User<String> stringUser = new User<String>();  
stringUser.addCredential("ACCESS_ALL_AREAS");
```

Можно параметризовать типом `String`...

```
User<int> intUser = new User<int>();  
intUser.addCredential(999);
```

...int
или любым другим

8.2.3. Ограничения на параметрические типы

К несчастью, другие программисты сочли, что ваш обобщенный класс `User` превосходен, и начали использовать его не так, как задумывалось, например для хранения безалкогольных напитков, которые пользователь покупает в автомате: `User<SoftDrink>`. Начальник заметил это и пришел к выводу, что ваш класс используют как средство избежать написания, более подходящего для конкретной задачи кода. Шеф решил положить этому конец и разрешил вам добавить в классы `Role` и `Permission` метод `validate()` и потребовать, чтобы оба реализовывали интерфейс `CredentialsValidator`.

Новое правило гласит: обобщенный класс `User` можно параметризовать любым типом, реализующим интерфейс `CredentialsValidator`. Механизм обобщенной типизации позволяет поддержать это правило, включив ключевое слово `extends` в определение параметрического типа. Изменим определение класса следующим образом:

```
class User<C extends CredentialsValidator> { ... }
```

Теперь при использовании класса `User` разрешено параметризовать его только типами, реализующими интерфейс `CredentialsValidator` (что исключает `String`, `int` и `SoftDrink`). Это также означает, что метод `addCredential()` может вызывать метод `validate()` параметрического класса, как показано в примере ниже, не проверяя предварительно, существует ли такой метод (как пришлось бы делать, если бы мы разрешили использовать в качестве параметрических типы `String` и `SoftDrink`):

```
class User<C extends CredentialsValidator> {  
    // остальные методы опущены
```

```
addCredentials(C credential) {  
  credential.validate();  
  this.credentials.add(credential);  
}  
}
```

Отличная работа! Вот вы и дошли до конца раздела, посвященного обобщенным типам – одному из продвинутых языковых средств. Обобщенные типы – мощное средство, имеющееся во многих современных языках на основе классов; в Dart оно базируется на тех же принципах, что в Java и C#.

Памятка

- Поймав себя на создании почти идентичных классов, подумайте о том, чтобы воспользоваться обобщенными типами.
 - Указанный в объявлении обобщенного типа параметрический тип можно использовать внутри определения типа, где он представляет конкретный тип, заданный пользователем класса.
 - На параметрические типы можно налагать ограничения с помощью ключевого слова `extends`.
-

В следующем разделе мы рассмотрим перегрузку операторов и в качестве примеров напишем свою реализацию оператора равенства `==` и добавим индексатор, чтобы пользователи нашего класса могли обращаться к элементам с помощью операторов `[]` и `[]=`.

8.3. Перегрузка операторов

Когда Алиса входит в систему управления табелями, упомянутую в предыдущем разделе, та строит объекты `Role`, представляющие права Алисы в системе. Например, Алиса может быть простым пользователем, то есть ей разрешено вводить данные о своем рабочем времени. Или ей может быть разрешено запрашивать отчеты по табелям других сотрудников. Наконец, она может быть администратором, то есть обладать правом редактирования любого табеля.

Каждая роль включает в себя все права ролей более низкого уровня, поэтому для правильной работы приложения нужно знать лишь роль с наивысшим уровнем доступа. Если Алисе назначена роль `TIMESHEET_ADMIN`, то она может также делать все, что разрешено ролям `TIMESHEET_REPORTER` и `TIMESHEET_USER`. Роли можно упорядочить по уровню доступа, как показано на рис. 8.10.

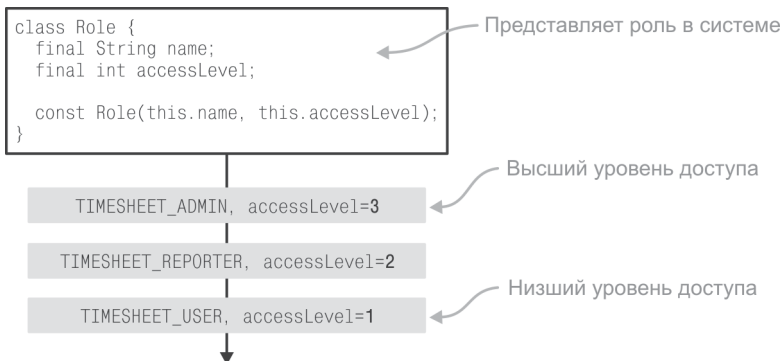


Рис. 8.10. Пример возможных уровней доступа Алисы в приложении для управления табелями

8.3.1. Перегрузка операторов сравнения

На множестве ролей определен естественный порядок: любые две роли можно сравнить. Чтобы узнать, какая роль больше, а какая меньше, можно написать код для сравнения значений `accessLevel`, и все будет работать. Но программу было бы проще читать, если бы экземпляры ролей можно было сравнивать непосредственно, пользуясь операторами `>` и `<`, как показано в следующем листинге.

Листинг 8.10. Различные способы сравнения ролей

<pre> var adminRole = new Role("TIMESHEET_ADMIN",3); var reporterRole = new Role("TIMESHEET_REPORTER", 2); var userRole = new Role("TIMESHEET_USER", 1); </pre>	Создаем три экземпляра ролей
<pre> if (adminRole.accessLevel > reporterRole.accessLevel) { print("Роль Admin больше, чем роль Reporter"); } if (userRole.accessLevel < adminRole.accessLevel) { print("Роль User меньше, чем роль Admin"); } </pre>	Для сравнения ролей можно использовать значение свойства <code>accessLevel...</code>
<pre> if (adminRole > reporterRole) { print("Роль Admin больше, чем роль Reporter"); } if (userRole < adminRole) { print("Роль User меньше, чем роль Admin"); } </pre>	но лучше сравнивать роли непосредственно

К счастью, Dart позволяет это сделать с помощью перегрузки операторов, то есть изменения семантики стандартных операторов. В данном случае мы хотим изменить семантику операторов «больше» и «меньше» для объектов класса `Role`. В Dart это делается так же, как при переопределении метода `toString()` (см. главу 7) – путем предоставления собственной реализации оператора. Ключевое слово `operator` сообщает Dart, что определяется новая реализация оператора.

Листинг 8.11. Новая реализация операторов < и > с помощью ключевого слова `operator`

```
class Role {  
  final String name;  
  final int _accessLevel; ← Теперь поле _accessLevel можно сделать закрытым  
  
  const Role(this.name, this._accessLevel);  
  
  bool operator >(Role other) {  
    return this._accessLevel > other._accessLevel;  
  }  
  
  bool operator <(Role other) {  
    return this._accessLevel < other._accessLevel;  
  }  
}
```

Слово `operator` в сочетании с обозначением оператора говорит, какой оператор мы собираемся переопределить

Перегруженная версия оператора реализуется в виде метода. Обычно такой метод принимает единственный параметр, содержащий другой экземпляр того же класса. В табл. 8.3 приведены некоторые стандартные операторы сравнения, которые можно перегрузить.

Таблица 8.3. Некоторые стандартные операторы сравнения

Метод оператора	Описание
<code>bool operator >(var other) {...}</code>	Данный экземпляр больше <code>other</code>
<code>bool operator <(var other) {...}</code>	Данный экземпляр меньше <code>other</code>
<code>bool operator >=(var other) {...}</code>	Данный экземпляр больше или равен <code>other</code>
<code>bool operator <=(var other) {...}</code>	Данный экземпляр меньше или равен <code>other</code>
<code>bool operator equals(var other) {...}</code> <code>bool operator ==(var other) {...}</code>	Данный экземпляр равен <code>other</code> . Отметим, что имеются две разные версии этого метода. На момент написания книги в спецификации языка оператору сравнения на равенство соответствовало слово <code>equals</code> , но в текущих реализациях для его представления применяется знак <code>==</code>

8.3.2. Неожиданное применение перегрузки операторов

В перегруженном операторе значение `other`, как правило, является объектом того же класса, в котором определен оператор, но это необязательно. В результате открывается возможность интересного, хотя и несколько необычного синтаксиса. Например, чтобы добавить пользователю роль, можно было бы перегрузить оператор `+` в классе `User`, что позволило бы писать следующий код.

Листинг 8.12. Перегрузка оператора сложения, позволяющая складывать объекты `User` и `Role`

```
class User {
    List roles;

    User() {
        roles = new List();
    }

    operator +(Role newRole) {
        this.roles.add(newRole);
    }
}

main() {
    User alice = new User();
    Role adminUser = new Role("TIMESHEET_ADMIN", 3);

    alice + adminUser;
    print(alice.roles.length);
}
```

Переопределяется
оператор +

← Используется оператор +
← Пользователю добавлена роль

Предупреждение. Перегружать операторы рекомендуется только в случае, когда это не станет для пользователя неожиданностью. Показанный выше код, пожалуй, был бы понятнее, если бы мы просто предоставили метод `add(Role)`. Разработчики не любят сюрпризов.

8.3.3. Перегрузка операторов доступа по индексу

При работе со списками и словарями мы пользовались операторами `[]=` и `[]` для доступа к элементам коллекции, например:

```
myList[1] = "Какое-то значение";  
var myValue = myList[1];
```

Оператор `[]` позволяет прочитать значение с указанным индексом, а оператор `[]=` – изменить его. И эти операторы можно перегружать. Оператор `[]` принимает один параметр – индекс – и возвращает значение элемента, а оператор `[]=` принимает индекс и новое значение элемента. Допустим, имеется класс `User` для представления пользователя, который может иметь ровно две роли. Тогда индексатор можно было бы использовать для чтения и записи этих ролей. В листинге ниже мы с помощью индексаторов обращаемся к свойствам `_role1` и `_role2`.

Листинг 8.13. Перегрузка индексаторов

```
class User {  
  Role _role1;  
  Role _role2  
  
  User() {  
    roles = new List();  
  }  
  
  operator []=(int index, Role role) { ← Переопределяется индексатор записи []=  
    if (index == 1) {  
      _role1 = role;  
    }  
    else if (index == 2) {  
      _role2 = role;  
    }  
    else throw new RangeError();  
  }  
  
  Role operator [](int index) { ← Переопределяется индексатор чтения  
    if (index == 1) {  
      return _role1;  
    }  
    else if (index == 2) {  
      return _role2;  
    }  
    else throw new RangeError();  
  }  
}
```

```

main() {
    User alice = new User();
    alice[1] = new Role("TIMESHEET_ADMIN", 3);
    alice[2] = new Role("TIMESHEET_USER", 1);
    var roleIndex1 = alice[1];
}

```

Индексатор записи используется для установки ролей по индексу
 ← Индексатор чтения используется для чтения ролей по индексу

Часто индексаторы применяются, когда нужно написать класс, реализующий интерфейс `Map`, чтобы его свойства можно было читать так, будто они являются элементами словаря, хотя в действительности это самые настоящие свойства. Такой способ позволяет, например, использовать анализатор JSON, который понимает словари и списки, для преобразования вашего класса в формат JSON. Данные же, представленные в формате JSON, можно передавать по сети. В листинге 8.14 показано, как реализовать в классе `Role` интерфейс `Map` и преобразовать объект этого класса в формат JSON. В этом фрагменте опущены некоторые стереотипные методы, требуемые интерфейсом `Map`, но в реальной программе все они должны быть реализованы. Кроме того, в листинге 8.14 встречаются кое-какие приемы, продемонстрированные выше в этой главе, в том числе возврат литеральных списков, а также типизированных и нетипизированных обобщенных коллекций.

Листинг 8.14. Реализация в классе интерфейса `Map`, чтобы можно было выполнять преобразование в формат JSON

```

class Role implements Map {
    String name;
    int _accessLevel;

    Role(this.name, this._accessLevel) {}

    // Методы Map
    bool containsKey(String key) {
        return key == "name" || key == "accessLevel";
    }

    operator[](String key) {
        if (key == "name") return this.name;
        if (key == "accessLevel") return this._accessLevel;
        return null;
    }

    void operator[]=(String key, var value) {

```

← Реализуется интерфейс `Map`

Возвращается true, если ключ равен «name» или «accessLevel»

Перегружается оператор [], чтобы можно было читать свойства

```

    if (key == "name") this.name = key;
    if (key == "accessLevel") this._accessLevel = value;
  }

```

ПЕРЕГРУЖАЕТСЯ ОПЕРАТОР
[], ЧТОБЫ МОЖНО БЫЛО
ИЗМЕНЯТЬ СВОЙСТВА

```

Collection<String> get keys {
  return ["name", "accessLevel"];
}

```

← Возвращается типизированная коллекция ключей типа
String, созданная в виде литерального списка

```

Collection get values {
  return [this.name, this._accessLevel];
}
//...остальные методы Map опущены...
}

```

← Возвращается нетипизированная коллекция
значений, созданная в виде литерального
списка

Реализовав интерфейс Map в классе Role, мы можем воспользоваться методом JSON.stringify(), определенным в библиотеке dart:json, для преобразования роли в строку:

```

Role adminRole = new Role("TIMESHEET_ADMIN", 3);
var roleString = JSON.stringify(adminRole);

```

Это сериализованное представление роли можно передать по сети (о чем мы будем говорить в части 3).

Памятка

- Ключевое слово `operator` в сочетании с обозначением оператора применяется для реализации метода, перегружающего оператор.
- Применяйте перегрузку операторов только тогда, когда это делает программу понятнее.
- Операторы индексации можно перегружать, чтобы обеспечить доступ к свойствам класса, как к элементам словаря.
- С помощью библиотеки `dart:json` можно преобразовывать классы, реализующие интерфейс Map, в JSON-строки.

8.4. Резюме

В этой главе мы долго и подробно рассматривали работу с коллекциями данных и взаимосвязь интерфейса Collection с некоторыми конкретными реализациями, в частности List, Queue и Set. Были продемонстрированы методы коллекций, такие как `forEach()` и `filter()`.

Мы также познакомились с интерфейсом словаря Map, который применяется для хранения пар ключ–значение, и видели, как с по-

мощью встроенной библиотеки JSON можно преобразовывать строки в словари и наоборот.

Подставляя конкретный тип вместо параметрического, мы можем работать с обобщенными классами коллекций типобезопасным образом, то есть по существу получать, к примеру, список экземпляров интересующего нас класса – скажем, `String` или `User`. Мы видели, как создать собственный обобщенный класс; это следует делать, когда в программе возникает несколько почти идентичных классов, отличающихся только параметрами методов и их возвращаемыми типами.

Наконец, мы рассмотрели перегрузку операторов – механизм, позволяющий сделать программу понятнее благодаря включению в собственный класс реализаций стандартных операторов, например `>` и `<`. Кульминацией в этом отношении является перегрузка операторов `[]` и `[]=` для реализации интерфейса `Map`, что позволяет преобразовать класс в формат JSON, воспользовавшись встроенной библиотекой `dart:json`.

В следующей главе мы будем детально изучать функции. Мы увидим, как использовать функции обратного вызова и будущие значения, чтобы получить понятный и высокопроизводительный асинхронный код.



Глава 9. Асинхронное программирование с применением обратных вызовов и будущих значений

В этой главе

- ❑ Модель неблокирующего асинхронного программирования.
- ❑ Обратные вызовы в асинхронных API.
- ❑ Как сделать асинхронные программы понятнее благодаря применению будущих значений и завершителей.
- ❑ Автономное тестирование асинхронного кода.

В веб-программировании нельзя рассчитывать, что события, не контролируемые вашим приложением, происходят в каком-то определенном порядке. Получение данных браузером от сервера может занять больше времени, чем вы ожидали, и вместо того чтобы дождаться результата, пользователь может нажать другую кнопку. Серверное Dart-приложение, скорее всего, должно будет приступить к обработке нового запроса еще до того, как предыдущий закончит читать данные из файловой системы. Такой способ программирования называется *асинхронной моделью*, а его противоположностью является *синхронная модель*. В синхронной модели все происходит строго по порядку – новое действие не начинается, пока не завершится предыдущее. В некоторых программах такое решение приемлемо, но в веб-приложении мы не можем блокировать всю работу в ожидании завершения предыдущей операции.

Эта мощная, но нетривиальная модель программирования применяется также в JavaScript. В этой главе мы потратим некоторое время на то, чтобы привыкнуть к асинхронному программированию и исследовать неблокирующую природу цикла обработки событий в Dart. В JavaScript асинхронность достигается за счет функций обратного вызова, и точно так же обстоит дело в Dart. Мы уже рассматривали функции обратного вызова в главе 4, но, поскольку это самый распространенный паттерн асинхронного программирования, в этой

главе вернемся к ним. Обратные вызовы, как вы убедитесь, несколько затрудняют чтение и сопровождение программы и делают неясным, как устроен в ней поток управления, но в Dart имеются два типа, которые отчасти решают эти проблемы: `Future` и `Completer`. Тип `Future` представляет будущее значение, то есть значение, которое станет известным только в будущем; он идеально подходит для асинхронного программирования. Ниже мы рассмотрим оба типа более подробно.

Наконец, мы воспользуемся вновь приобретенными знаниями об асинхронном программировании для написания автономных тестов, специально рассчитанных на асинхронный код. Обычно автономные тесты выполняются последовательно, и приложение, содержащее комплект тестов, завершается сразу после прогона последнего теста. Однако такой подход не работает, когда программа все еще ожидает ответа от внешнего исполнителя. К счастью, библиотека автономного тестирования в Dart позволяет дожидаться завершения автономного тестирования, в чем мы и убедимся в конце главы.

Но сначала давайте посмотрим, что происходит в синхронном блокирующем веб-приложении. Во многих странах регулярно проводятся лотереи, а розыгрыш призов производится с помощью специальной машины, из которой выскакивают шары с выигравшими числами. Игроки сравнивают выпавшие числа со своими билетами. Чтобы добавить напряжения и азарта, шары вываливаются через случайные интервалы времени. Сначала мы напишем синхронное приложение, в котором браузер не будет реагировать на действия пользователя, пока генерируются выигравшие числа, а затем исправим ситуацию, воспользовавшись асинхронными API, – это позволит сделать браузер более отзывчивым. На рис. 9.1 иллюстрируется проблема, возникающая в синхронном приложении.



Рис. 9.1. Синхронная версия приложения «Лотерея Dart» блокирует выполнение в ожидании завершения операции

9.1. Почему веб-приложение должно быть асинхронным

Мы собираемся написать веб-приложение, которое будет моделировать лотерею. После нажатия кнопки **Start** генерируются три выигравших числа со случайным промежутком времени, не превышающим двух секунд. Из-за этой задержки браузер ненадолго зависает. Для реализации задержки мы будем «крутиться» в цикле `while`, пока не истечет требуемое время. Выигравшие числа отображаются на экране. В листинге 9.1 приведен полный код приложения.

Предупреждение. Приведенный код не годится для веб-приложения. Он включен только для демонстрации блокирующего поведения. В следующем разделе мы покажем, как с помощью асинхронных API правильно ждать истечения заданного промежутка времени.

Листинг 9.1. Синхронное блокирующее приложение `lottery_app.dart`

```
import "dart:html";
import "dart:math";

int getWinningNumber() {
  int millisecsToWait = new Random().nextInt(2000);
  var currentMs = new Date.now().millisecondsSinceEpoch;
  var endMs = currentMs + millisecsToWait;
  while (currentMs < endMs) {
    currentMs = new Date.now().millisecondsSinceEpoch;
  }
  return new Random().nextInt(59) + 1; ← Возвращаем выигравшее число от 1 до 60
}

startLottery() {
  int num1 = getWinningNumber();
  query("#ball1").innerHTML = "$num1";
  int num2 = getWinningNumber();
  query("#ball2").innerHTML = "$num2";
  int num3 = getWinningNumber();
  query("#ball3").innerHTML = "$num3";
}

resetLottery() {
  query("#ball1").innerHTML = "";
}
```

Получаем текущее время
и вычисляем, в какой момент
прекратить ожидание

Крутимся в блокирующем цикле, пока
не истечет требуемое время

Следующий вызов `getWinningNumber()`
не начинается, пока не закончится предыдущий

```
query("#ball2").innerHTML = "";
query("#ball3").innerHTML = "";
}

main() {
    var startButton = new Element.html("<button>Start</button>");
    document.body.children.add(startButton);
    startButton.on.click.add((e) {
        startButton.disabled = true;
        startLottery();
    });

    var resetButton = new Element.html("<button>Replay</button>");
    document.body.children.add(resetButton);
    resetButton.on.click.add((e) {
        startButton.disabled = false;
        resetLottery();
    });
}
```

Запускаем
розыгрыш лотереи

Код прямолинейный: читая функцию `startLottery()`, мы видим, что она запрашивает очередное выигравшее число и обновляет определенный `<div>` в пользовательском интерфейсе. К сожалению, в реальности дело обстоит не так: у браузера есть цикл обработки событий, в котором обрабатываются задачи, стоящие в очереди, а поскольку наша программа блокирует поток выполнения, то до обработки событий обновления пользовательского интерфейса дело не дойдет, пока наш код не выполнится до конца. Чем больше времени мы проведем в функциях `getWinningNumber()`, тем дольше браузер будет ждать возможности возобновить обработку событий. На рис. 9.2 показан цикл обработки событий браузера и порядок выполнения нашего кода.

На практике многие операции, например взаимодействие с сервером, браузер выполняет в отдельных потоках, поэтому цикл обработки событий должен только запустить операцию и ждать уведомления о ее завершении (которое тоже ставится в очередь событий). Но сделать это невозможно, пока выполняется блокирующий код, поэтому нам предстоит изменить программу, так чтобы каждый участок завершал выполнение как можно быстрее. Решить эту задачу можно благодаря асинхронным API.

Преобразуя приложение в асинхронное, мы разбиваем его на ряд функций, имея в виду, что они будут выполняться асинхронно.

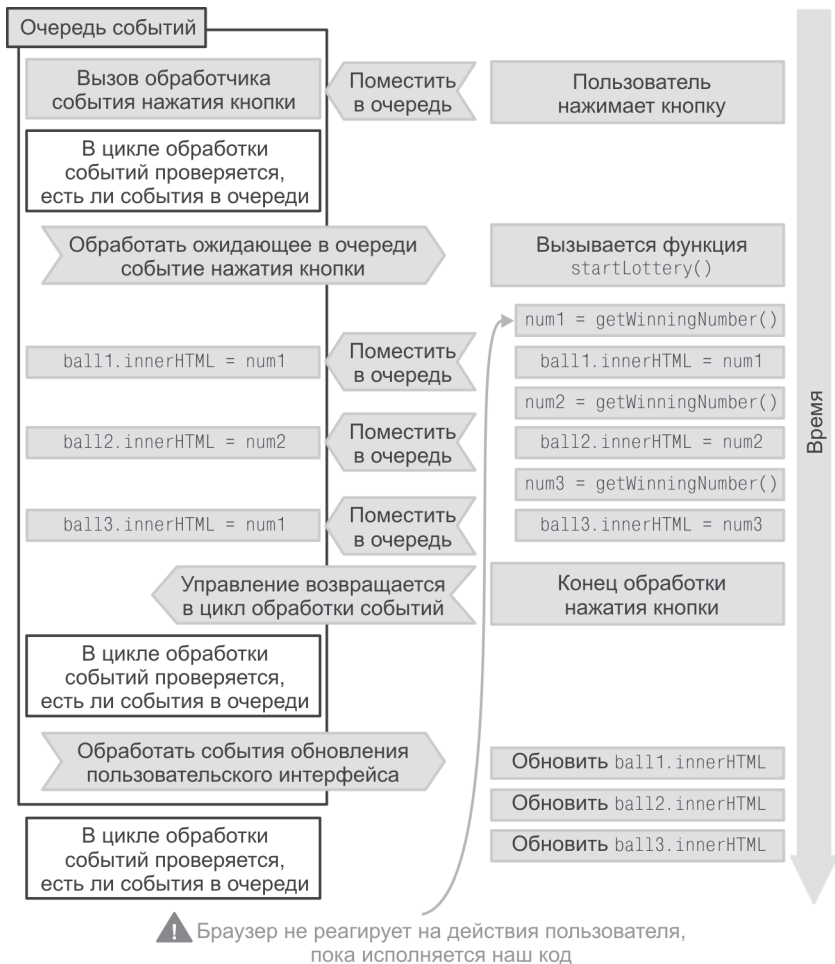


Рис. 9.2. Браузер обрабатывает события в цикле, только когда не выполняется наш код

Чтобы отделить пользовательский интерфейс от прикладных функций, нужно организовать код в виде нескольких файлов. В файл `lottery_app.dart` мы поместим код взаимодействия с интерфейсом, а в библиотеку `lottery` – прикладные функции. Эта связь показана на рис. 9.3.

При такой организации мы впоследствии сможем написать асинхронные автономные тесты для прикладных функций.

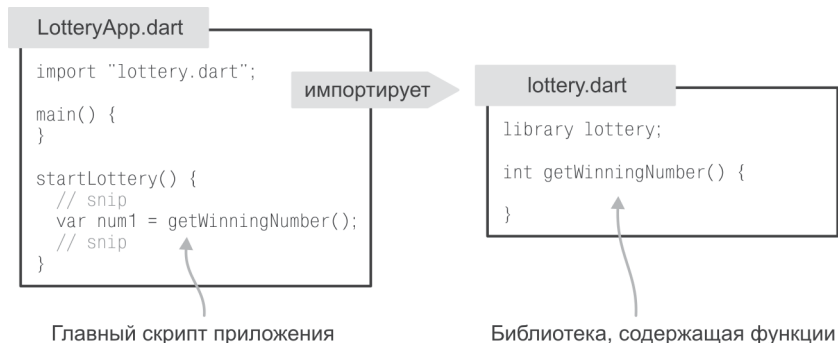


Рис. 9.3. Связь между приложением LotteryApp и библиотекой lottery

9.1.1. Преобразование приложения в асинхронное

Определившись со структурой приложения, мы можем приступить к его модификации. Чтобы сделать лотерею более захватывающей, лучше всего генерировать выигравшие числа в случайном порядке и отображать их в интерфейсе по мере готовности. Тогда сначала может появиться второе число, за ним третье и в конце первое — в зависимости от того, сколько времени займет генерация числа.

На рис. 9.4 показан порядок заполнения пользовательского интерфейса и сообщения на консоли браузера. Как видите, выигравшие числа появлялись не в том порядке, в каком запрашивались.

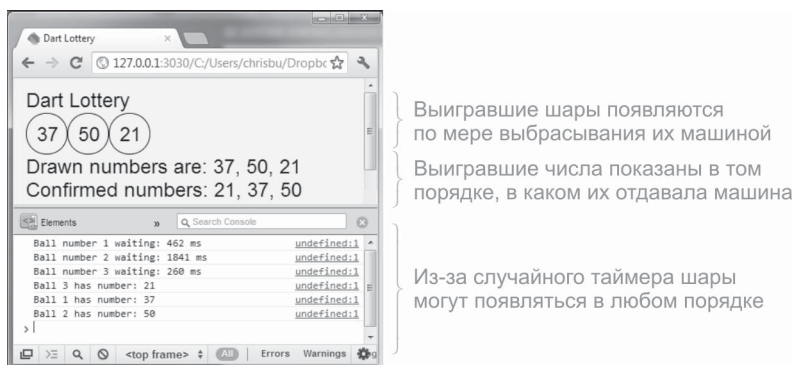


Рис. 9.4. В Лотерее Dart для генерации числа требуется случайное время

Вы, конечно, заметили, что в Лотерее Dart участвуют всего три шара, так что выиграть в ней гораздо проще, чем в реальных лотереях.

Задержки в реальных приложениях

Задержка между моментом запуска приложения и появлением пронумерованных шаров позволила нам промоделировать асинхронность, с которой приходится иметь дело клиентскому приложению. В реальной программе асинхронность может быть обусловлена тем, что данные запрашиваются у трех разных серверов или необходимостью ждать трех разных действий пользователя. В Лотерее Dart ожидаемые данные – это число на выброшенном машиной шаре, а задержка ввода-вывода моделируется таймером, срабатывающим через случайное время.

В синхронной версии Лотереи Dart мы ждали случайное время, пока машина выбросит шар, и, не получив его, не приступали к вытягиванию следующего шара. В асинхронной же версии шары могут выбрасываться в любом порядке. Может случиться, что третий шар появится раньше первого. Псевдокод асинхронной программы приведен ниже:

```
Запросить первый шар, ждать не более 2 секунд
Запросить второй шар, ждать не более 2 секунд
Запросить третий шар, ждать не более 2 секунд
```

```
Настроить обработчик события для первого шара
Настроить обработчик события для второго шара
Настроить обработчик события для третьего шара
```

```
Вернуть управление циклу обработки событий
```

```
Вызван обработчик события для первого шара
Вызван обработчик события для второго шара
Вызван обработчик события для третьего шара
```

Начало асинхронной программы – вызов `window.setTimeout()`

Основой программы лотереи является библиотека `lottery`. В ней находится код, возвращающий случайное число по прошествии случайного времени. Для этого в библиотеке используется встроенная в браузер функция `window.setTimeout(callback, duration)`, которая вы-



зывает указанную функцию спустя заданное количество миллисекунд. Тривиальное применение этой функции показано в примере ниже, где обработчик события обратного вызова выделен полужирным курсивом:

```
getWinningNumber() {  
    Random r = new Random();  
    int millisecsToWait = r.nextInt(2000);  
    window.setTimeout(() {  
        // сгенерировать выигравшее число  
        // и обновить пользовательский интерфейс  
    }, millisecsToWait);  
};
```

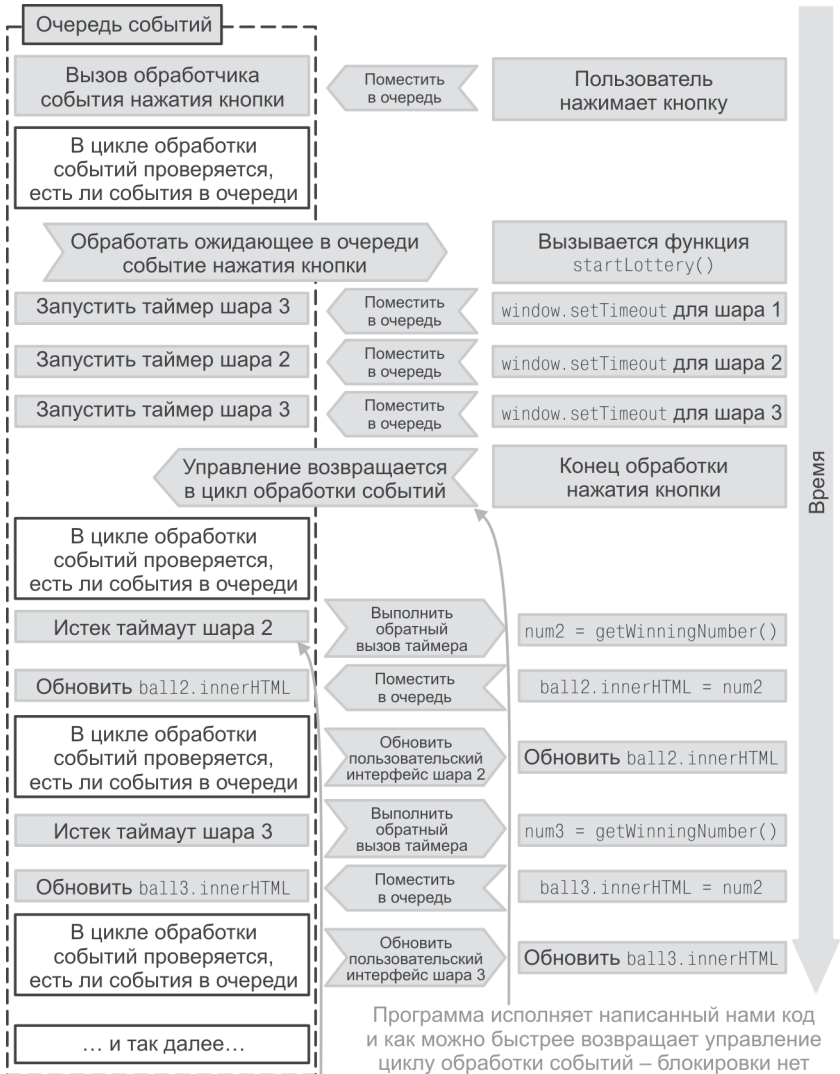
Dart-приложение вызывает функцию `window.setTimeout()` и сразу же возвращает управление циклу обработки событий. По прошествии указанного количества миллисекунд из этого цикла вызывается обработчик события, переданный функции `setTimeout()`. Это основной принцип асинхронного программирования: программа начинает операцию и возвращает управление циклу обработки событий, а когда операция завершается, программа получает от него уведомление.

На рис. 9.5 показано, как функция `window.setTimeout()` асинхронно взаимодействует с циклом обработки событий.

Такая асинхронная обработка событий в порядке их возникновения имеет место и в других API, например при запросе данных от сервера в классе `HttpRequest`. Ваше приложение запрашивает данные и возвращает управление циклу обработки событий. Когда поступят данные, цикл вызовет анонимную функцию-обработчик. Ниже приведен пример, обратный вызов анонимной функции выделен полужирным курсивом:

```
HttpRequest.get("http://example.com", (data) {  
    // обработать возвращенные данные  
});
```

В следующем разделе мы воспользуемся функцией `setTimeout()` для обратного вызова функций, взаимодействующих с асинхронными API, предоставляемыми браузером. В первый раз мы познакомились с функциями обратного вызова еще в главе 1, а теперь, поняв, что требуется от программы лотереи, вернемся к ним в контексте асинхронного программирования.



По истечении таймута осуществляется обратный вызов написанного на Dart кода, который исполняется и возвращает управление циклу обработки событий. Браузер продолжает реагировать на действия пользователя

Рис. 9.5. Асинхронность означает, что управление возвращается циклу обработки событий как можно быстрее

Памятка

- Синхронный код выполняется строго последовательно, ожидая завершения каждой блокирующей операции.
 - Асинхронный код не блокирует выполнение. Цикл обработки событий отвечает за уведомление приложения по завершении операции ввода-вывода.
-

9.2. Использование обратных вызовов в асинхронном программировании

Наше приложение состоит из двух файлов: библиотеки `lottery`, которая отвечает за асинхронную работу и содержит полезные функции для генерации чисел и последующей сортировки их по порядку, и файла `lottery_app.dart`, в котором находится функция `main()` и импортируется библиотека `lottery`. Функция `main()` первой выполняется после запуска Dart-приложения, а вынесение основных функций в отдельный файл `lottery.dart` впоследствии поможет нам произвести рефакторинг и тестирование асинхронного кода.

В первой асинхронной версии Лотереи Dart мы используем функции обратного вызова для получения шаров с числами. Эта модель программирования часто применяется при разработке на JavaScript – как на клиенте, так и на сервере – и возможна потому, что функции можно передавать в качестве параметров.

Примечание. Обратные вызовы – широко распространенная техника, и вы должны привыкнуть читать код, где они встречаются. Однако у них есть и недостатки, о которых мы скоро узнаем. В следующем разделе показано, как улучшить API, в которых применяются обратные вызовы, с помощью пары объектов `Future` и `Completer`.

Простой код обратного вызова, который по-прежнему выполняет синхронно, но без задержки, показан на рис. 9.6. Эта программа генерирует лишь одно выигравшее число и печатает сведения о нем на консоли браузера.

В коде на рис. 9.6 нет ничего асинхронного. Функция `main()` вызывает `getWinningNumber()`, передавая ей функцию обратного вызова в качестве параметра. `getWinningNumber()` генерирует случайное число и передает его функции обратного вызова, которая печатает число на консоли. Только по завершении этих шагов управление возвращается в цикл обработки событий. Пока что все хорошо, так как

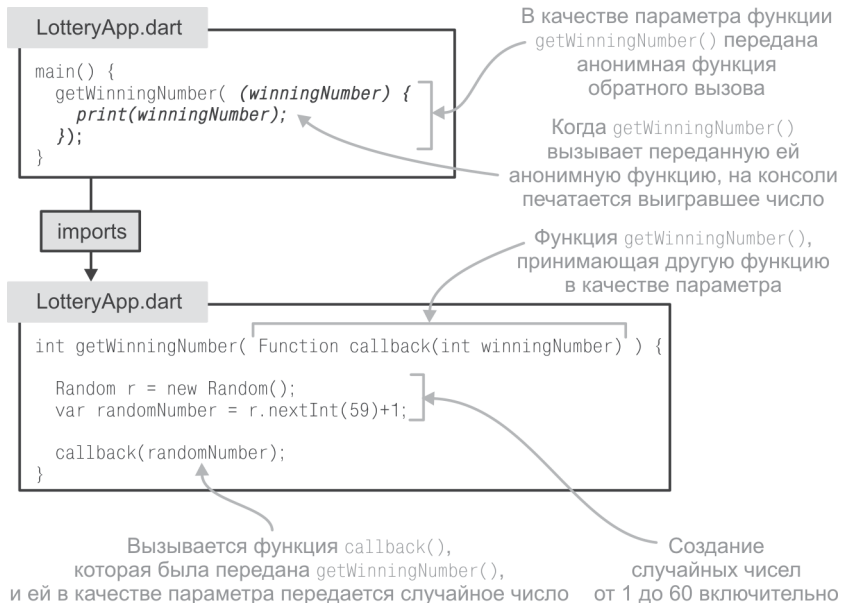


Рис. 9.6. В файле `lottery.dart` определена функция `getWinningNumber()`, которая принимает функцию обратного вызова в качестве параметра. Эта функция вызывается, когда будет получено выигравшее число

никакого блокирующего кода еще нет. Приложение может трижды вызвать `getWinningNumber()`, и на консоли сразу же появятся все три выигравших числа.

Чтобы немного улучшить пользовательский интерфейс приложения, мы добавим служебную функцию `updateResult(int ball, int winningNumber)`, которая модифицирует элемент `<div>`, представляющий шар с выигравшим числом. Теперь файл `lottery_app.dart` содержит код, показанный в следующем листинге, где продемонстрирована как короткая, так и длинная форма записи функции, вызывающей `updateResult()`.

Листинг 9.2. `lottery_app.dart`: применение функций обратного вызова

```
import "lottery.dart";
import "dart:html";

main() {
```

```

getWinningNumber( (int result1) => updateResult(1, result1) ); |
getWinningNumber( (int result2) => updateResult(2, result2) ); |
getWinningNumber( (int result3) {                               |
    updateResult(3, result3);                                   |
});                                                             |
}                                                                 |

```

getWinningNumber() передаются
анонимные функции обратного вызова
Длинная форма вызова анонимных функций
тоже разрешена

```

updateResult(int ball, int winningNumber) {
    var ballDiv = query("#ball$ball");
    ballDiv.innerHTML = "$winningNumber";
}

```

Служебная функция для обновления
элемента <div> в ГИП

В библиотеке `lottery` находится только одна функция `getWinningNumber()`, которая принимает функцию обратного вызова и генерирует случайное число (рис. 9.6). Ниже мы изменим эту функцию, так что она будет обращаться к `window.setTimeout()`, – это добавит лотерее интригу.

9.2.1. Добавление асинхронных обратных вызовов в Лотерею Dart

Теперь можно приступить к разработке асинхронной версии Лотереи Dart. Мы можем сгенерировать случайные числа и показать их на экране. Но поскольку код синхронный, то предложения будут исполняться последовательно: сначала появится `result1`, потом `result2` и, наконец, `result3`.

Но в такой лотерее нет никакой интриги (а в приложении – асинхронного кода), потому что мы получаем числа с той скоростью, с какой компьютер способен их генерировать. Однако добавить напряженности не так уж сложно, даже не изменяя файла `lottery_app.dart`: изменим функцию `getWinningNumber()` в библиотеке `lottery`, включив в нее обращение к `window.setTimeout()`, чтобы переданная нами функция вызывалась через случайное время, – тогда результаты будут генерироваться в случайном порядке. Быть может, сначала появится `result3`, почти сразу за ним – `result1` и – после небольшой задержки – `result2`. В листинге 9.3 показана модифицированная функция `getWinningNumber()`, в которой наша функция вызывается со случайной задержкой не более двух секунд.

Листинг 9.3. `lottery.dart`: включение таймера в `getWinningNumber()`

```

library lottery;

import "dart:html"; ← Включается ради window.setTimeout()

```

```
import "dart:math"; ← Включается ради класса Random

getWinningNumber(Function callback(int winningNumber)) {
  Random r = new Random();
  var randomNumber = r.nextInt(59) + 1;
  int millisecs = r.nextInt(2000); ← Случайное число, не более 2 секунд
                                   Вызывается window.setTimeout(),
  window.setTimeout( () { ← которой передается анонимная функция
    callback(randomNumber); ← По истечении тайм-аута вызывается функция обратного вызова
  }, millisecs);           ← Второй параметр – величина задержки в миллисекундах
}
```

Совет. При реализации анимации для периодического обновления экрана лучше вызывать браузерную функцию `requestAnimationFrame()`, определенную в спецификации HTML5. Связано это с тем, что она выбирает частоту обновления с учетом возможностей браузера и оборудования и работает только тогда, когда вкладка браузера видна на экране.

Все три вызова `getWinningNumber()` из `main()` происходят один за другим, без задержки. Но поскольку они обращаются к `setTimeout()`, сами результаты будут сгенерированы в неопределенный момент в будущем. После того как вызовы `getWinningNumber()` сделаны, управление возвращается в цикл обработки событий, который вызовет заданные нами функции обратного вызова, когда истекнут тайм-ауты (рис. 9.7).

Именно так должна работать реальная программа. Запрашивая данные у сервера, мы не знаем, сколько времени серверу потребуется для ответа. Тем временем пользователь может нажать какую-нибудь другую кнопку, инициировав новый запрос серверу. Приложение может получать ответы сервера и вызывать соответствующие функции обратного вызова в любом порядке.

9.2.2. Ожидание завершения всех асинхронных обратных вызовов перед продолжением

В Лотерее Dart недостаточно просто показать на экране сгенерированные машиной числа. После того как все они будут готовы, их нужно отобразить по порядку. Для этого мы введем еще одну служебную функцию `getResultsString (List<int> winningNumbers, String message)`. Она получает строку сообщения и список выигравших чисел.

1. Код выполняется последовательно с той скоростью, на которую способен компьютер

2. Таймеры запущены, и поток управления вернулся в цикл обработки событий

```
main() {  
  getWinningNumber((int result1) => updateResult(1,result1));  
  getWinningNumber((int result2) => updateResult(2,result2));  
  getWinningNumber((int result3) => updateResult(3,result3));  
}
```

Таймер 1
ждет
700 мс

Таймер 2
ждет
910 мс

Таймер 3
ждет
200 мс

3. Когда очередной таймер срабатывает, цикл обработки событий вызывает ассоциированную с этим таймером функцию обратного вызова, которая выполняет функцию, переданную `getWinningNumber()`

Выполняется: `(int result3) => updateResult(3,result3)`

Выполняется: `(int result1) => updateResult(1,result1)`

Выполняется: `(int result2) => updateResult(2,result2)`

Рис. 9.7. Программа работает

с максимально возможной скоростью до выхода из `main()`.
В этот момент цикл обработки событий еще ждет истечения тайм-аутов, после чего вызывает написанный нами на Dart код

Итак, мы добавили в приложение требование упорядоченности действий. Получать три числа и показывать их на экране можно в любом порядке, но к следующей части программы – отображению строки результатов – можно перейти только после того, как все три числа видны на экране.

Это несколько усложняет асинхронный код, потому что функции обратного вызова ничего не знают друг о друге. Поэтому необходим какой-то механизм проверки. Есть, по крайней мере, два способа решить эту задачу. Первый – завести глобальный флаг в функции `main()` и в каждой функции обратного вызова проверять, все ли числа уже получены. Второй – организовать вложенные обратные вызовы.

В листинге 9.4 показан модифицированный файл `lottery_app.dart`, в котором функции обратного вызова обращаются к функции `addAndDisplay()`, объявленной внутри `main()`, чтобы добавить свой результат в общий список `results`, определенный в `main()`. Только после того как в списке окажутся все три числа, `addAndDisplay()` вызывает `getResultsString()` и отображает содержимое списка в элементе `<div>`.

Кстати, функция `addAndDisplay()` оказывается замыканием, потому что передается каждому обратному вызову, сохраняя доступ к переменной `results`, даже когда вызывается вне области видимости `getWinningNumber()`. Если вы подзабыли, что такое замыкания, обратитесь к главе 4.

Листинг 9.4. `lottery_app.dart`: реализация упорядоченности действий в асинхронной программе

```
main() {
  List<int> results = new List<int>(); ← Список результатов

  void addAndDisplay(int result) {
    results.add(result);
    if (results.length == 3) {
      var resultString = getResultsString(results, "Winning numbers: ");
      var winningNumbersDiv = query("#winningNumbers");
      winningNumbersDiv.innerHTML = resultString;
    }
  }

  getWinningNumber( (int result1) {
    updateResult(1, result1);
    addAndDisplay(result1);
  });

  getWinningNumber( (int result2) {
    updateResult(2, result2);
    addAndDisplay(result2);
  });

  getWinningNumber( (int result3) {
    updateResult(3, result3);
    addAndDisplay(result3);
  });
}
```

`addAndDisplay()` становится замыканием в результате передачи ее функциям обратного вызова

Каждая функция обратного вызова вызывает `addAndDisplay()`, передавая ей свой результат

Эта программа работает, но ее сложность существенно возросла. Читая код сверху вниз, невозможно понять, что в каком порядке происходит. Исходя из логики `addAndDisplay()`, можно лишь сказать, что элемент `<div>` с идентификатором `winningNumbers` будет заполнен только после завершения всех трех обратных вызовов.

К счастью, другой подход к реализации упорядоченности — вложенные обратные вызовы — позволяет в какой-то мере сохранить



читабельность за счет того, что все три функции исполняются одновременно. У этого решения есть и другое достоинство – часто требуется имитировать синхронный поток управления, даже имея дело с асинхронным API. Например, бывает нужно гарантировать, что запрос к серверу 2 будет отправлен только после получения данных от сервера 1 или что подключение к базе данных произошло до того, как в ней обратились с запросом.

9.2.3. Вложенные обратные вызовы как средство управления порядком асинхронного выполнения

Техника вложенных обратных вызовов позволяет имитировать синхронное выполнение, когда имеется только асинхронный API, например `getWinningNumbers()`. Она часто применяется в JavaScript, особенно в серверном каркасе Node.js или в VM Dart для исполнения асинхронного кода в правильном порядке, например: открытие файла, чтение файла, закрытие файла. Все эти операции асинхронны, но должны выполняться именно в таком порядке. Но у этой техники имеется и крупный недостаток: когда число вложенных вызовов превышает три, максимум четыре, программу снова становится трудно читать.

Применение техники вложенных обратных вызовов в нашем приложении означает, что первый обратный вызов вызывает вторую функцию `getWinningNumber()`, второй обратный вызов – третью функцию `getWinningNumber()` и т. д. Последний обратный вызов выполнит завершающий шаг цепочки, то есть отобразит список результатов на экране.

В листинге 9.5 приведена версия `lottery_app.dart`, в которой для гарантии того, что результаты отображаются только после получения третьего выигравшего числа, применяются вложенные обратные вызовы.

Листинг 9.5. `lottery_app.dart`: использование вложенных обратных вызовов

```
main() {  
  getWinningNumber( (int result1) {  
    updateResult(1, result1);  
    getWinningNumber( (int result2) {  
      updateResult(2, result2);  
    }  
  }  
}
```

← Функции `getWinningNumber()` передается первая функция обратного вызова
Первая функция обратного вызова вызывает `getWinningNumber()`, передавая ей вторую функцию обратного вызова

```

getWinningNumber( (int result3 ) {
    updateResult(3, result3);
    List results = new List();
    results.add(result1);
    results.add(result2);
    results.add(result3);
    var resultString = getResultsString(results, "Winning numbers: ");
    var winningNumbersDiv = query("#winningNumbers");
    winningNumbersDiv.innerHTML = resultString;
});
});
});
}

```

Вторая функция обратного вызова вызывает
← getWinningNumber(), передавая ей третью
функцию обратного вызова

← Третья функция обратного
вызова отображает выигравшие
числа

Как видите, при трех уровнях вложенности читать программу уже не так просто. А требование отсортировать шары по порядку – также с использованием асинхронного API – означает еще один – четвертый – вложенный обратный вызов.

К сожалению, такое требование очень часто встречается в реальных программах, где в нашем распоряжении имеются только асинхронные API, а обеспечить определенный порядок действий или дождаться завершения нескольких асинхронных вызовов все-таки надо. Практический пример операций на стороне клиента – обработчик нажатия кнопки, который должен получить данные от сервера, что-то сделать с ними и отправить серверу ответ, уведомив пользователя о завершении всей последовательности действий. Сколько же существует JavaScript-приложений с огромным количеством уровней вложенности асинхронных вызовов! Это явление даже получило специальное название – *проклятие обратных вызовов* (callback hell). В примере ниже показано, как выглядел бы код, если бы шаров было не три, а шесть:

```

getWinningNumber( (int result1) {
    updateResult(1, result1);
    getWinningNumber( (int result2) {
        updateResult(2, result2);
        getWinningNumber( (int result3) {
            updateResult(3, result3);
            getWinningNumber( (int result4) {
                updateResult(4, result4);
                getWinningNumber( (int result5) {

```

```
        updateResult(5, result5);
        getWinningNumber( (int result6) {
            updateResult(6, result6);
            // вызов getResultsString() опущен
        });
    });
});
});
});
});
```

В мире JavaScript создано немало каркасов, ставящих целью решить проблему вложенных обратных вызовов, но Dart предлагает собственное решение в виде типов `Future` и `Completer`. Они дают элегантный механизм выполнения асинхронного кода в определенном порядке без вложенности, при этом гарантируется, что выполнение продолжится только после завершения всех асинхронных операций.

Памятка

- Обратные вызовы – стандартный прием асинхронного выполнения операций.
 - Функция обратного вызова, переданная асинхронному API, будет вызвана по завершении асинхронной операции.
-

Чтобы гарантировать, что очередной участок программы начнет выполняться только после завершения всех асинхронных операций, можно завести счетчик или еще какой-нибудь флаг и изменять его при завершении каждой операции.

Для выполнения асинхронных API в определенном порядке можно воспользоваться вложенными обратными вызовами.

9.3. Знакомство с типами `Future` и `Completer`

В предыдущем разделе мы видели, как выполнить асинхронный код с помощью функций обратного вызова. При запросе выигравшего числа от функции `getWinningNumber()` мы делаем асинхронный вызов, который возвращает результат спустя случайное время. После того как все выигравшие числа получены, мы переходим к следующему шагу приложения: отображению чисел в пользовательском интерфейсе.

Код, необходимый для того, чтобы проверить, все ли числа уже получены, стал более сложным, и ориентироваться в нем гораздо труднее. К счастью, Dart предлагает два типа, `Future` и `Completer`, которые позволяют писать асинхронный код, более пригодный для чтения и сопровождения.

Эти типы используются совместно и позволяют получить *будущее значение*, то есть объект, который примет значение в будущем. Объект `Completer` возвращает объект типа `Future`, представляющий будущее значение. Это значение заполняется при вызове метода `completer.complete(value)`, которому в параметре `value` как раз и передается значение `Future`. Мы можем обернуть обращение к функции `getWinningNumber(callback)`, так что будет возвращаться будущее значение. На рис. 9.8 показано, как это сделать; а далее мы рассмотрим, как будущее значение используется.

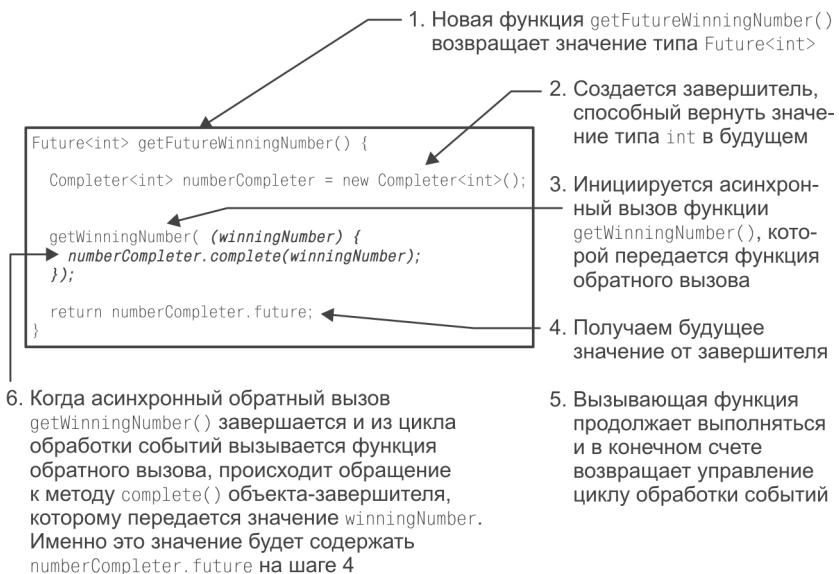


Рис. 9.8. Вызов асинхронного API можно инкапсулировать с помощью типов `Future` и `Completer`

При вызове `getFutureWinningNumber()` мы немедленно получаем результат в виде объекта `Future<int>`, который вернул завершитель. Программа продолжает выполняться, используя будущее значение вместо реального. Внутри `getFutureWinningNumber()` мы вызвали асин-

хронную функцию `getWinningNumber()`, которая запрашивает данные от асинхронного API. В какой-то момент, когда методу `complete()` завершителя будет передано реальное значение, объект `Future` наконец будет заполнен. Но как получить это будущее значение? И снова с помощью функции обратного вызова, которая передается методу `then()` объекта `Future`.

Посмотрим, как это происходит в действительности. Для этого вернемся к первому простому варианту функции обратного вызова, в котором отображаются все три числа. В листинге 9.6 приведен новый файл `lottery_app.dart`, где используется функция `getFutureWinningNumber()`, и для сравнения — первоначальная версия с функциями обратного вызова.

Листинг 9.6. `lottery_app.dart`: использование будущих значений и метода `then()`

```
main() {
  Future<int> f1 = getFutureWinningNumber();
  Future<int> f2 = getFutureWinningNumber();
  Future<int> f3 = getFutureWinningNumber();

  f1.then((int result1) => updateResult(1, result1));
  f2.then((int result2) => updateResult(2, result2));
  f3.then((int result3) => updateResult(3, result3));

  // getWinningNumber( (int result1) => updateResult(1, result1) );
  // getWinningNumber( (int result2) => updateResult(2, result2) );
  // getWinningNumber( (int result3) => updateResult(3, result3) );
}
```

Запрос выигравшего числа
возвращает
будущее значение

Функция обратного вызова,
переданная методу `then()`,
выполняется, когда объект
`Future` получает реальное значение

Первоначальный код с явными
обратными вызовами для сравнения

Этот код можно записать и короче, сцепив вызов исходной функции и метода `then()`:

```
getFutureWinningNumber().then((int result1) => updateResult(1, result1));
```

С точки зрения функциональности, мы, похоже, ничего не выиграли. Мы по-прежнему передаем функцию обратного вызова методу `then()`, и эта функция вызывается, когда будущее значение становится реальным (само будущее значение передается функции обратного вызова в качестве параметра). Так какое же «новое качество» дают нам будущие значения?

9.3.1. Передача будущих значений из одного места программы в другое

Самое главное, что можно делать с будущими значениями, – передавать их из одного места приложения в другое. В версии с обратными вызовами мы должны решить, что делать с выигравшим числом в момент вызова `getWinningNumber()`, – в данном случае мы передаем его функции `updateResult()`.

В версии с будущими значениями это решение можно отложить, оставив на усмотрение другой части программы, например функции `updateUserInterface()`, которой будущее значение передается в качестве параметра. У такого подхода два преимущества: во-первых, уменьшается количество уровней вложенности, а во-вторых, мы получаем возможность передать все три будущих значения какой-то другой функции, несмотря на то что представляемые ими асинхронные результаты еще не возвращены. Переработаем файл `lottery_app.dart` еще раз – передавая будущие значения функции `updateUserInterface()`.

Листинг 9.7. `lottery_app.dart`: передача будущих значений функции

```
main() {
  Future<int> f1 = getFutureWinningNumber();
  Future<int> f2 = getFutureWinningNumber();
  Future<int> f3 = getFutureWinningNumber();

  updateUserInterface(f1, f2, f3);  ← Будущие значения передаются другой функции
}

void updateUserInterface(Future first, Future second, Future third)
  first.then((int result1) => updateResult(1, result1));
  second.then((int result2) => updateResult(2, result2));
  third.then((int result3) => updateResult(3, result3));
}
```

Другая функция решает, что с ними делать, обращаясь к методу `then()`

Это очень важная особенность будущих значений. В реальных программах возможны, в частности, такие сценарии: передача нескольких будущих значений данных, полученных от сервера, компоненту ГИП и обработка «будущего» нажатия кнопки пользователем путем передачи его некоторой функции в приложении. Поскольку будущие значения – это переменные, которые знают, что получают значение в будущем (через метод `then()`), то их можно сохранить в объекте другого класса, например в списке. Это открывает воз-

возможность еще одного интересного применения будущих значений: упорядочения асинхронных вызовов.

9.3.2. Упорядочение асинхронных вызовов путем сцепления будущих значений

Напомним, что асинхронные обратные вызовы вкладывались, для того чтобы гарантировать определенный порядок выполнения. Когда уровень вложенности оказывается выше некоторого порога, мы получаем проклятие обратных вызовов, то есть затруднения при чтении программы. Эффекта вложенности можно добиться с помощью будущих значений – достаточно поместить следующее будущее значение в метод `then()` предыдущего:

```
getFutureWinningNumber().then( (result1) {  
    updateResult(1, result1);  
    getFutureWinningNumber().then( (result2) {  
        updateResult(2, result2);  
        getFutureWinningNumber().then( (result3) {  
            updateResult(3, result3);  
        });  
    });  
});  
});
```

Такое решение ничем не лучше вложенных обратных вызовов, так что оптимальной тактикой вложения будущих значений его не назовешь. По счастью, в Dart существует механизм сцепления будущих значений, так что следующее выполняется только по завершении предыдущего – безо всякой вложенности. Для этого служит метод `future.chain(callback)`, используемый вместо `future.then()`. Функция `chain()` возвращает еще одно будущее значение, что дает возможность сцеплять будущие значения, как показано в листинге ниже.

Листинг 9.8. `lottery_app.dart`: сцепление будущих значений для реализации упорядоченности

```
main() {  
    getFutureWinningNumber().chain( (result1) {  
        updateResult(1, result1);  
        return getFutureWinningNumber();  
    }).chain( (result2) {  
        updateResult(1, result2);  
        return getFutureWinningNumber();  
    })  
}
```

Вместо `then()` используется метод `chain()`
← возвращенного будущего значения
← В `chain()` возвращаем следующее будущее значение...
...и так далее, пока не будут сцеплены
← все будущие значения

```
}).then( (result3) {  
    updateResult(1, result3);  
});  
}
```

← На последнем шаге используется then(),
потому что он не возвращает будущего значения

Как видите, можно сцепить вместе несколько будущих значений безо всякой вложенности – и понятность программы при этом не уменьшается. Очередное обращение к `getFutureWinningNumber()` производится не раньше, чем завершится предыдущее.

9.3.3. Ожидание завершения всех запущенных операций получения будущих значений

При обсуждении обратных вызовов мы хотели, чтобы результаты отображались в виде отформатированной строки с помощью функции `getResultsString()`, но только после получения всех трех выигравших чисел. Чтобы добиться этого, мы получали каждое значение асинхронно, а потом вызывали функцию `addAndDisplay()`, которая помещала новое значение в список и отображала результат, лишь если в списке оказывалось ровно три элемента (по одному для каждого асинхронного вызова). Такое решение работает, но усложняет код, который, по идее, должен быть совершенно линейным. Нам всего-то и нужно, что три раза обратиться к функции `getWinningNumber()` и, когда все три вызова завершатся, перейти к следующему шагу.

Именно в таких ситуациях механизм будущих значений раскрывается во всем своем блеске. В Dart имеется метод `Futures.wait(futures)`, который принимает список будущих значений и возвращает одно будущее значение. Это единственное будущее значение содержит список всех значений, которые возвращают переданные на вход будущие значения. Звучит сложновато, но на практике оказывается просто, как показано на рис. 9.9. Эта функция ждет возврата трех реальных значений от трех будущих, а затем передает все три реальных значения – три выигравших числа – для отображения на экране.

Передавая все будущие значения функции `wait()`, мы можем быть уверены, что перейдем к следующему шагу не раньше, чем для каждого из них будет получено реальное значение. Поскольку `wait()` также возвращает будущее значение, ее можно включить в цепочку вызовов функции `chain()` наряду с другими будущими значениями.

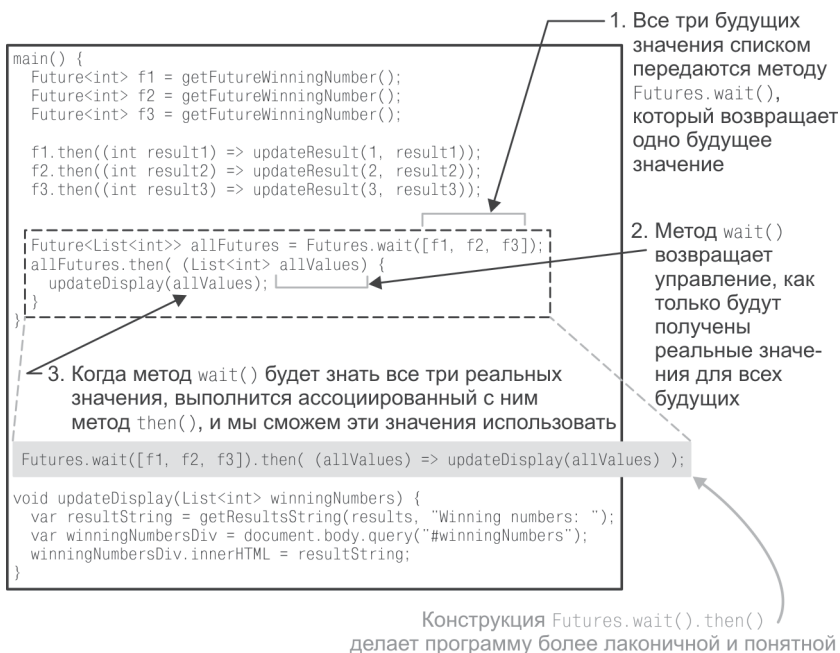


Рис. 9.9. Метод `Futures.wait()` ждет получения нескольких будущих результатов, перед тем как продолжить выполнение

Это может быть полезно, например в случае, когда мы сначала ожидаем получения трех выигравших чисел, а затем запрашиваем четвертый «бонусный шар», как показано в листинге 9.9. Заодно мы здесь помещаем полученные от `getFutureWinningNumber()` будущие значения непосредственно в список.

Листинг 9.9. `lottery_app.dart`: ожидание будущих результатов и сцепление

```
main {
  List futureValues = new List();
  futureValues.add(getFutureWinningNumber());
  futureValues.add(getFutureWinningNumber());
  futureValues.add(getFutureWinningNumber());

  Futures.wait(futureValues).chain( (List firstThreeNumbers) {
    getResultsString(firstThreeNumbers, "Winning numbers: ");
    // ... код отображения опущен ...
  });
}
```

Помещаем три будущих значения в список

С помощью `chain()` ожидаем получения будущих значений

Отображаем три выигравших числа

```

return getFutureWinningNumber(); ← Возвращаем будущее значение из chain(),
                                чтобы получить бонусный шар
}).then( (int bonusBall) {      ← В методе then() бонусного шара отображаем число
    getResultsString([bonusBall], "Bonus ball");
    // ... код отображения опущен ...

});
}

```

Используя методы `then()` и `chain()` класса `Future`, а также статический метод `wait()` этого же класса, мы смогли сделать асинхронный код понятным, избежав глубокой вложенности и в то же время сохранив порядок выполнения операций. В классе `Future` имеется еще один метод, заслуживающий внимания: `transform()`.

9.3.4. Преобразование обычных значений в будущее

Выше мы видели, как легко можно передавать будущие значения из одного места приложения в другое. Этот подход годится, когда нам нужны именно будущие значения, но иногда будущие значения — лишь средство для достижения конечного результата. В Лотерее Dart нам интересна строка результатов, порожаемая функцией `getResultsString()`. Именно эту строку мы и хотим передавать из одного места в другое. Но для получения строки результатов необходимо выполнить ряд асинхронных вызовов, поэтому просто взять значение, возвращенное `getResultsString()`, и передать его другой функции не получится. Вместо этого можно было бы вызвать `getResultsString()` в обращении к методу `then()` последнего будущего значения, но иногда требуется использовать еще одно будущее значение. В результате получаются вложенные будущие значения, то есть то, чего мы как раз хотели избежать. Описанная проблема иллюстрируется в следующем примере:

```

Future.wait([f1, f2, f3]).then((List results) {
    String resultString = getResultsString(results); ← Получаем строку результатов
                                                    Результат: вложенные будущие значения
    formatResultString(resultString).then(( formattedResultString ) { ←
        // сделать что-то с отформатированной строкой результатов ←
    });
});
});

```

Передаем resultString другой функции,
которая возвращает будущее значение

В такой ситуации на помощь приходит функция `transform(callback)`. Она похожа на функцию `chain()` в том смысле, что возвращает будущее значение. Разница же в том, что `chain()` ожидает, что будущее значение вернет переданная ей функция обратного вызова, а `transform()` преобразует любое значение в будущее, позволяя возвращать будущее значение из каждого метода в последовательности вызовов, даже если некоторые вызовы не являются асинхронными.

В листинге 9.10 показана функция `transform()` в действии: мы ждем получения всех трех выигравших чисел, затем передаем их функции `getResultsString()` и автоматически обортываем результат `getResultsString()` в новое будущее значение, сцепляемое со следующим обращением к `formatResultString()`, результат которого используется для обновления пользовательского интерфейса. Это позволяет избежать вложенного вызова `formatResultString()` из функции обратного вызова, переданной методу `wait().then()`.

Листинг 9.10. `lottery_app.dart`: преобразование обычного значения в будущее

```
main() {
  List futureValues = new List();
  futureValues.add(getFutureWinningNumber());
  futureValues.add(getFutureWinningNumber());
  futureValues.add(getFutureWinningNumber());

  Futures.wait(futureValues).transform( (List winningNums) {
    String result = getResultsString(winningNums, "Winning numbers");
    return result;
  }).chain((resultString) {
    return formatResultString(resultString);
  }).then((formattedResultString) {
    var winningNumbersDiv = query("#winningNumbers");
    winningNumbersDiv.innerHTML = resultString;
  });
}

Future<String> formatResultString(String resultString) {
  // асинхронная реализация опущена
}
```

Асинхронные вызовы
для получения
выигравших чисел

Возвращается

← будущее значение

← ...но эта строка преобразуется
в будущее значение с помощью transform()

Возвращается строка,
а не будущее значение...

Будущий результат

← передается другой функции

← Когда будущее значение resultString
получит реальное значение, оно будет
отображено на экране

Будущие значения в сочетании с завершителями – это мощный механизм написания асинхронного кода, пригодного для чтения и сопровождения. Связывая в одну цепочку асинхронные вызовы с помощью `chain()` и неасинхронные с помощью `transform()`, мы можем представить поток выполнения в единообразном виде.

Многие асинхронные API в Dart возвращают будущие значения, но в некоторых все же используются функции обратного вызова. В этом разделе мы видели, как обернуть API, основанный на обратном вызове, в данном случае функцию `getWinningNumber(callback)`, таким образом, чтобы возвращалось будущее значение, а обратный вызов был инкапсулирован внутри обертки: для этого мы возвращаем будущее значение, которое становится реальным в результате обратного вызова.

Памятка

- Асинхронные API на основе обратных вызовов можно обернуть так, что будут возвращаться будущие значения.
 - Функция обратного вызова, передаваемая методу `then(callback)` будущего значения, вызывается, когда будущее значение становится реальным.
 - Метод `transform(callback)` преобразует значение, возвращенное функцией обратного вызова `callback`, в новое будущее значение.
 - Метод `chain(callback)` ожидает, что функция обратного вызова `callback` вернет будущее значение.
 - Функции обратного вызова, переданные методам `chain()` и `transform()`, вызываются, когда будущее значение становится реальным, но позволяют возвращать новое будущее значение. Этим можно воспользоваться, чтобы построить последовательность из нескольких асинхронных и синхронных вызовов.
-

В последнем разделе этой главы мы снова обратимся к автономному тестированию, с которым расстались в главе 3, и рассмотрим, как писать автономные тесты для асинхронных библиотечных функций.

9.4. Автономное тестирование асинхронных API

Каркас автономного тестирования в Dart позволяет тестировать функции в отдельном скрипте со своей функцией `main()`. Этот скрипт импортирует прикладную библиотеку и тестирует находящиеся в ней функции и методы. Ожидается, что каждый автономный тест выполняет какой-то код, завершается и сообщает, прошел тест или

нет, сравнивая полученный результат с ожидаемым. В следующем листинге показан автономный тест для функции `getResultsString()` из библиотеки `lottery`. Код аналогичен тому, что рассматривался ранее.

Листинг 9.11. `lottery_test.dart`: автономный тест для функции `getResultsString()`

<code>import "lottery.dart";</code>	← Импортируется тестируемая библиотека
<code>import "dart:html";</code>	Стереотипный код
<code>import "lib/unittest/unittest.dart";</code>	импорта библиотек тестирования
<code>import "lib/unittest/html_config.dart";</code>	и работы с HTML

```
main() {
  useHtmlConfiguration(); ← Стереотипный код регистрации библиотеки HTML
  test('Results String', () {
    var result = getResultsString([23,44,57], "Winning numbers: ");
    expect(result, equals("Winning numbers: 23, 44, 57"));
  });
}
```

Внутри test() вызывается тестируемая функция
← Полученный результат
сравнивается
с ожидаемым

Это стандартный синхронный автономный тест. Поток управления вполне ожидаем:

1. Начинается выполнение функции `main()`.
2. Начинается выполнение функции `test()`.
3. Вызывается функция `getResultsString()` с тестовыми параметрами.
4. Возвращенное значение сравнивается с ожидаемым.
5. Функция `test()` завершается.
6. Функция `main()` завершается.

Тестирование асинхронных API типа `getWinningNumber(callback)`, в которых используются обратные вызовы, или `getFutureWinningNumber()`, в которых используются будущие значения, требует несколько иного подхода.

Из-за асинхронности функции `test()` и `main()` уже завершат выполнение к тому моменту, как обратный вызов вернет управление или будущее значение станет реальным. Поток выполнения будет таким:

1. Начинается выполнение функции `main()`.
2. Начинается выполнение функции `test()`.
3. Вызывается функция `getFutureWinningNumber()` с тестовыми параметрами.
4. Функция `test()` завершается.

5. Функция `main()` завершается.
6. Будущее значение становится реальным.

При такой последовательности возникает проблема, потому что сравнивать с ожидаемым следует именно будущее значение, но к тому моменту, когда оно становится доступным, функции `test()` и `main()` уже завершились. К счастью, в каркас автономного тестирования в Dart включено несколько функций `expectAsync` вида `expectAsync0()`, `expectAsync1()`, `expectAsync2()`, которые обертывают функцию обратного вызова, переданную асинхронным прикладным API или методу `then()` будущего значения. Число в конце имени функции `expectAsync` обозначает количество аргументов, ожидаемых функцией обратного вызова. Обратные вызовы, ассоциированные с будущими значениями, принимают единственный аргумент (число, которое станет реальным значением будущего). Для них, как и для вашей собственной асинхронной прикладной функции с обратным вызовом, принимающим единственный аргумент, подойдет функция `expectAsync1()`. Давайте напишем тест для варианта функции `getWinningNumber()`, принимающего функцию обратного вызова, а затем тест для функции `getFutureWinningNumber()`.

9.4.1. Тестирование асинхронных функций обратного вызова

Функция `expectAsync` обертывает функцию обратного вызова, переданную `getWinningNumber()`. Ниже показано, как вызывается функция `getWinningNumber()` и как ей передается функция обратного вызова:

```
getWinningNumber( (int result) {  
    // ...опущено ... сделать что-то с полученным результатом  
} );
```

Такой код вполне уместен в контексте браузера, потому что сразу после вызова функции `getWinningNumber()` мы возвращаемся в цикл обработки событий. Но в автономном тесте мы должны как-то дождаться возврата значения, перед тем как возвращать управление. Именно для этого и предназначена функция `expectAsync()`: она обертывает обратный вызов, заставляя каркас автономного тестирования ждать завершения асинхронного вызова. В результате мы получаем возможность сравнить полученное значение с ожидаемым, как показано в следующем примере, где функция `expectAsync1()` выделена полужирным курсивом:

```
getWinningNumber( expectAsync1( int result) {  
  // ...опущено ... проверить полученное значение  
});
```

В простом тесте, приведенном в следующем листинге, проверяется, что возвращенное число принадлежит диапазону 1–60, включая границы.

Листинг 9.12. `lottery_test.dart`: тестирование асинхронной функции обратного вызова с помощью `expectAsync()`

```
import "lottery.dart";  
import "dart:html";  
import "lib/unittest/unittest.dart";  
import "lib/unittest/html_config.dart";  
  
main() {  
  useHtmlConfiguration();  
  test('Winning Number Callback', () {  
    getWinningNumber( expectAsync1( int result) {  
      expect(result, greaterThanOrEqualTo(1));  
      expect(result, lessThanOrEqualTo(60));  
    });  
  });  
}
```

← Обертывание обратного вызова функцией `expectAsync()` позволяет корректно проверить результат

Поняв, как тестировать асинхронные функции обратного вызова, применим ту же технику к асинхронным функциям, возвращающим будущие значения.

9.4.2. Тестирование будущих значений

Последнее, что нам предстоит, – научиться тестировать функцию `getFutureWinningNumber()`. Метод `then()` будущего значения принимает функцию, которая будет вызвана для получения реального значения. Именно эту функцию обратного вызова мы и должны обернуть функцией `expectAsync1()`, которая позволит сравнить возвращенное будущее значение с ожидаемым. Ниже показано, как тестируется будущее значение; функция `expectAsync1()` выделена полужирным курсивом:

```
test("Future winning number", () {  
  getFutureWinningNumber().then( expectAsync1( int result) {
```

```

    expect(result, greaterThanOrEqualTo(1));
    expect(result, lessThanOrEqualTo(60));
  }));
);

```

Можно даже проверить несколько будущих значений, воспользовавшись функцией `Futures.wait()` и обернув ее метод `then()` функцией `expectAsync1()`, как показано в следующем листинге, где ожидается, что в список будут помещены три значения.

Листинг 9.13. `lottery_test.dart`: тестирование нескольких будущих значений с помощью `wait()` и `expectAsync1()`

```

import "lottery.dart";
import "dart:html";
import "lib/unittest/unittest.dart";
import "lib/unittest/html_config.dart";

main() {
  useHtmlConfiguration();

  test('Winning Number Callback', () {
    Future f1 = getFutureWinningNumber();
    Future f2 = getFutureWinningNumber();
    Future f3 = getFutureWinningNumber();

    Futures.wait([f1,f2,f3]).then( expectAsync1( (List<int> results) {
      expect(results.length, equals(3));
    }));
  });
}

```

Ждем, затем вызываем `expectAsync1()`
для проверки возвращенных результатов

← Проверяем, что в списке три элемента

При проверке будущих значений функций `expectAsync1()` следует обертывать функцию обратного вызова, которая передается методу `then()` будущего значения. Это также означает, что мы можем использовать методы `.chain()` и `.transform()` для сцепления будущих значений в определенном порядке. А функция обратного вызова, передаваемая методу `then()` последнего будущего значения, обертывается функцией `expectAsync1()` так же, как в предыдущем листинге.

Памятка

- Для тестирования асинхронных API требуется специальная техника, иначе автономный тест завершится еще до сравнения возвращенного значения с ожидаемым.

- Функцию обратного вызова, передаваемую либо асинхронной прикладной функции, либо методу `then()` будущего значения, необходимо обертывать функцией `expectAsync()`, которая заставляет каркас тестирования ждать завершения асинхронного вызова.
-

9.5. Резюме

Нет никаких сомнений – асинхронное программирование труднее синхронного. Но использование неблокирующих асинхронных API позволяет приложению реагировать на действия пользователя, даже когда одновременно выполняется несколько операций – в противном случае приложение оказалось бы заблокированным. В серверном приложении операцией мог бы быть доступ к файловой системе или запрос через сетевой сокет, в клиентском – запрос данных у сервера или ожидание ввода данных пользователем. Асинхронные API, встроенные в Dart, дают вашей программе возможность начать выполнение операции и сразу же вернуть управление циклу обработки событий. Позже, когда произойдет что-то значимое, например завершение чтения файла или получение данных от сервера, ваш код будет вызван снова.

Функции обратного вызова знакомы разработчикам на JavaScript, но когда число вложенных обратных вызовов слишком велико – а так часто бывает, когда требуется гарантировать определенный порядок выполнения, – возникает явление, получившее название «проклятие обратных вызовов». Dart не запрещает функции обратного вызова, но вместе с тем предоставляет типы `Future` и `Completer`, которые, работая совместно, дают будущее значение, допускающее передачу из одного места программы в другое. Будущее значение становится реальным только после того, как асинхронный вызов завершится.

Несколько асинхронных вызовов можно сцеплять с помощью функции `chain()`, которая позволяет избежать глубокой вложенности обратных вызовов. Кроме того, метод `transform()` класса `Future` дает возможность использовать синхронные API вперемежку с асинхронными. С помощью функции `wait()` программа может дождаться, когда все будущие значения станут реальными, и только потом продолжить выполнение.

Наконец, для автономного тестирования асинхронных API применяются функции `expectAsync`, которые заставляют каркас тестирования не завершать тест преждевременно, а дождаться возврата



из функции обратного вызова или заполнения будущего значения. Благодаря этим функциям вы можете тестировать свои асинхронные функции точно так же, как обычный синхронный код.

Теперь вы знаете почти обо всех концепциях языка Dart и можете приступать к созданию приложений. В следующей главе мы будем рассматривать взаимодействие с браузером и напишем одностраничное веб-приложение с несколькими представлениями и хранилищем данных, которое будет запрашивать данные у сервера.



ЧАСТЬ III. КЛИЕНТСКИЕ DART-ПРИЛОЖЕНИЯ

Третья часть книги посвящена клиентской стороне: разработке браузерных одностраничных Dart-приложений с помощью новейших технологий на базе HTML5.

В главе 10 мы познакомимся с циклом обработки событий в Dart и объясним, как сконструировать пользовательский интерфейс с помощью библиотек для манипуляции моделью DOM и подписаться на браузерные события.

В главе 11 мы научимся взаимодействовать с браузером, то есть реагировать на события навигации. Мы воспользуемся браузерными куками и автономным хранилищем для сохранения данных и состояния между сеансами, а также узнаем, как представлять объекты в формате JSON.

Добравшись до главы 12, мы уже сможем выйти за пределы экосистемы Dart и научить приложение взаимодействовать с внешними системами. Мы воспользуемся появившейся в HTML5 технологией AppCache, благодаря которой приложение сможет работать, когда сервер недоступен, а затем оформим приложение в виде пакета для Chrome.

В этой части мы будем иметь дело только с клиентской стороной. Но по ее завершении мы будем готовы к тому, чтобы исследовать Dart на стороне сервера и создать для нашего клиентского приложения ответную серверную часть.



Глава 10. Создание веб-приложения на Dart

В этой главе

- ❑ Структура одностраничного веб-приложения.
- ❑ Построение динамических представлений из элементов в библиотеке `dart:html`.
- ❑ Обработка событий браузера.

Основное назначение Dart – создание сложных веб-приложений. Типичное веб-приложение состоит из одной страницы, которая загружается в браузер и служит вместилищем кода приложения. Этот код отвечает за конструирование пользовательского интерфейса, навигацию и сохранение данных пользователя – на сервере или локально.

Разрабатывая приложение на Dart, имеет смысл начать с работы в локальном режиме. Локальная работа имеет ряд плюсов с точки зрения как разработчика, так и пользователя. В локальном режиме вы можете подставить тестовые данные, которые в конечном итоге будут заменены комбинацией хранения на клиенте (благодаря поддержке HTML5 API) и на сервере. При таком подходе клиентское приложение на Dart можно написать вообще без сервера или базы данных, что сократит время отладки и позволит автономно протестировать его.

Что касается конечного пользователя, то он сможет воспользоваться вашим приложением, когда нет связи, например в самолете или в районах со слабым сетевым покрытием. Это значит, что вы можете писать на Dart приложения, работающие даже тогда, когда устройство не подключено к сети. Для конечного пользователя важно и то, что приложение, спроектированное для работы в автономном режиме, будет работать гораздо быстрее приложения, которому сеть нужна постоянно: ведь данные по возможности сохраняются локально, а значит, приложение может начать работу гораздо быстрее, не дожидаясь, пока с сервера загрузится большой объем данных.

В этой главе показано, как, подготовив единый набор исходных файлов, можно с помощью HTML-элементов построить пользовательский интерфейс с несколькими представлениями. На момент написания этой книги в состав Dart SDK не входила библиотека интерфейсных виджетов, но даже с применением одних лишь классов `Element` и `Event` из библиотеки `dart:html` можно сконструировать вполне симпатичный интерфейс, написав сравнительно немного кода. Познакомившись с классами из иерархии `Element`, мы перейдем к модели браузерных событий и покажем, как браузер вызывает написанные на Dart обработчики событий, которые реагируют на действия пользователя. По ходу изучения библиотеки `dart:html` мы создадим компонент «сетка», которым вы сможете воспользоваться в собственных приложениях.

10.1. Структура одностраничного веб-приложения

По структуре одностраничное веб-приложение отличается от традиционного. В традиционном сайте, написанном на Java или C# (или PHP, Ruby, Python и т. д.), обычно имеется серверный контроллер, который строит ГИП, объединяя HTML-представление с данными (классами моделей), а затем отправляет его клиенту. Переход на другую страницу (например, для редактирования) приводит к отправке нового запроса серверу для получения другого представления и других данных.

Но в последние годы появились каркасы, которые позволяют один раз отправить клиенту представление, а затем с помощью AJAX запрашивать у сервера только данные и привязывать их к представлению. По существу, классы представления, модели и контроллера перебрались на сторону клиента, где они могут воспользоваться всей мощностью компьютера пользователя и разгрузить сервер для обработки большего числа запросов (рис. 10.1).

Именно такая модель лежит в основе системы Google GWT, ее можно реализовать и с помощью целого ряда каркасов для JavaScript. Dart-приложения также придерживаются этой модели; более того, среда и инструменты Dart дают наибольший эффект, когда приложение проектируется именно для такой работы. Объединяя представление с данными на стороне клиента, мы заодно получаем возможность подставлять данные на клиенте, что удобно для разработки и тестирования.

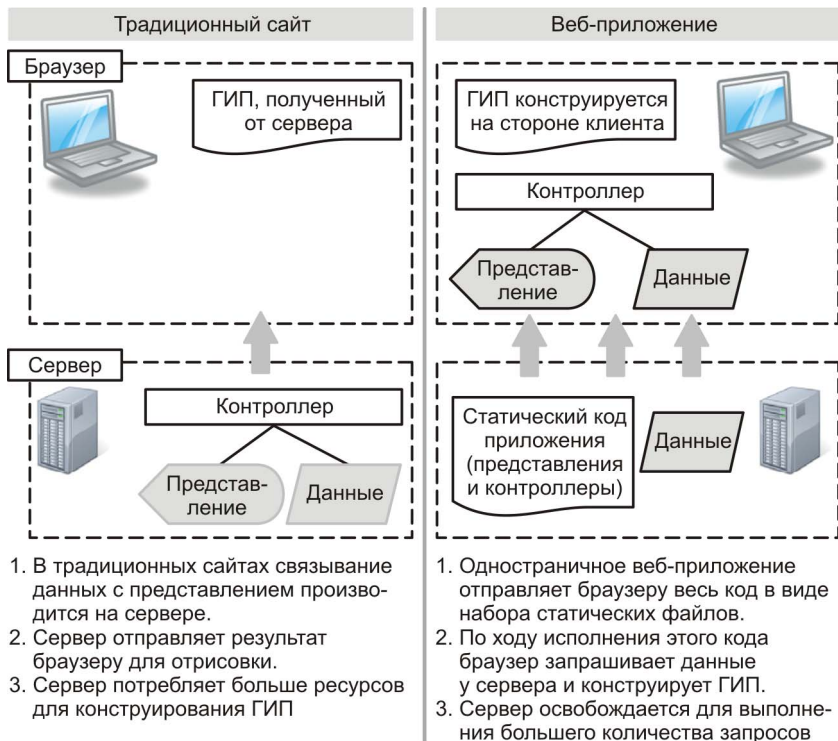


Рис. 10.1. В современных веб-приложениях обработка на сервере уступила место использованию возможностей клиента для конструирования пользовательского интерфейса из представлений и данных

10.1.1. Приложение *DartExpense* – постановка задачи

В этой главе мы напишем приложение *DartExpense*, которое будет хранить перечень производственных расходов. Бухгалтерия решила избавиться от ведомостей, которые работники отправляют по электронной почте, и заменить их веб-приложением. Текущая система на основе электронных таблиц работает вне зависимости от того, есть у пользователя связь с сетью или нет; как правило, пользователь отправляет ведомость раз в месяц. Менеджеры по продажам обычно заполняют ведомость в полетах, а бухгалтерия хочет отучить их от этого.

В существующей ведомости каждая статья расходов описывается пятью полями:

- ☐ дата расхода;
- ☐ тип расхода;
- ☐ сумма;
- ☐ пояснения;
- ☐ истребовано ли уже возмещение.

Наша задача – реализовать эту модель на Dart.

Данное приложение, хотя и упрощенное в учебных целях, устроено так же, как в реальной жизни, – с классами моделей и представлений. На момент написания этой книги в Dart еще не было специальной библиотеки, абстрагирующей пользовательский интерфейс, наподобие ExtJS или jQuery UI, однако разработчики заявляют, что хотят превратить Dart в полнофункциональное решение и библиотека пользовательского интерфейса обязательно появится. Ну а пока мы должны конструировать ГИП из фрагментов HTML и элементов, привязывая их к данным. Мы рассмотрим, как это делается с применением библиотеки `dart:html`, с которой мы уже сталкивались в главе 3. Эта библиотека аналогична базовой библиотеке jQuery для JavaScript, в которой стандартизованы манипуляции с HTML и моделью DOM.

В приложении DartExpense будет два экрана, между которыми пользователь сможет переключаться: список расходов и окно для добавления или изменения статьи расхода. Оба экрана показаны на рис. 10.2.

В этой главе мы будем работать исключительно с подставными данными, что позволит запускать приложение, не используя ничего, кроме редактора Dart и браузера Dartium. Исходный код можно найти на сайте книги по адресу www.manning.com/DartinAction. В следующей главе мы добавим клиентское хранилище HTML5, а в главе 14 – серверный компонент. Пока же для доступа к данным будем применять подставные классы, в которые данные жестко зашиты.

Преобразование Dart в JavaScript – обтрясывание деревьев

При использовании подставных данных в JavaScript необходимо предусмотреть какой-то механизм, предотвращающий их попадание в клиентский браузер (совершенно излишний трафик). Но при трансляции Dart на JavaScript остается только код, который реально выполняется. Этот механизм удаления лишнего называется *обтрясыванием деревьев* (tree-shaking). На этапе разработки вы можете создавать подставные классы, не заботясь о последствиях; после того как все ссылки на подставные функции доступа к данным будут удалены, отладочные данные не будут отправляться конечному пользователю, хотя в проекте останутся.

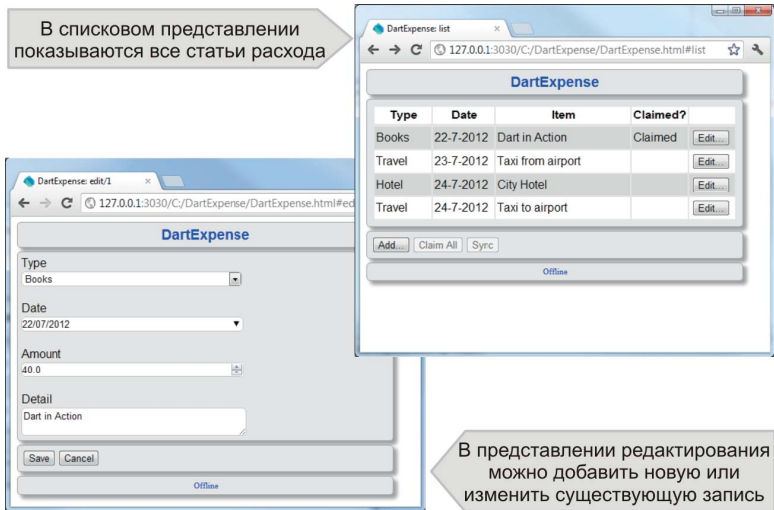


Рис. 10.2. В приложении DartExpense имеются два представления: список и редактирование

В двух модельных классах `Expense` и `ExpenseType` хранятся данные, полученные от пользователя, и справочник. Имеются также два представления: `ListView` — для показа списка расходов — и `EditView` — для редактирования. В каждом представлении есть два элемента верхнего уровня: само представление, хранящееся в свойстве `viewElement`, и контейнер действий, в котором размещены кнопки, например **Save** и **Cancel**. Единственный класс контроллера, `AppController`, связывает между собой представления и данные. Открытые свойства и методы этих классов показаны на рис. 10.3.

Кроме классов моделей, представлений и контроллера, имеются еще классы, соответствующие функциям доступа к данным. В интерфейсе `DataAccess` определены свойства и методы для получения списка расходов и добавления новой статьи расхода. В этой главе мы создаем автономное приложение, поэтому интерфейс `DataAccess` реализуется классом `MockData`:

```
abstract class DataAccess {
    Map<String, ExpenseType> expenseTypes;
    Map<int, Expense> _expenses;
    List<Expense> get expenses;
    bool addOrUpdate(Expense expense);
}
```




Рис. 10.3. Основными классами в приложении являются модели, представления и контроллер; их конструкторы выделены полужирным шрифтом

В классе `MockData` все расходы и справочник их типов жестко защищены в код, а экземпляр класса передается конструктору класса `AppController`. Метод `addOrUpdate()` просто добавляет в список новую статью расхода или обновляет существующую. В следующей главе мы заменим класс `MockData` локальным хранилищем данных, а затем – сохранением данных на сервере.

10.1.2. Структура Dart-приложения

Когда мы переходим к Dart-приложению, на реальном веб-сервере или запуская его из редактора Dart, браузер запрашивает HTML-файл, содержащий ссылку на код приложения. HTML-файл прило-

жения `DartExpense`, показанный в листинге 10.1, минимален. В нем всего два тега `script`, стандартно создаваемых редактором. Первый, с типом `application/dart`, ссылается на исходный файл, содержащий точку входа в приложение. Второй ссылается на JavaScript-файл с кодом для определения того, поддерживает браузер язык Dart или нет. Если нет, то все присутствующие на странице теги `script` с типом `application/dart` заменяются соответствующими JavaScript-версиями.

Листинг 10.1. `DartExpense.html`

```
<!DOCTYPE html>  ← DOCTYPE, соответствующий HTML5

<html>
  <head>
    <title>DartExpense</title>
    <link rel="stylesheet" type="text/css"
          href="dartexpense.css" /> | Присоединяется каскадная
                                   | таблица стилей
  </head>
  <body>

    <div id="dartexpense"></div> ← В этом <div>'е будет находиться
                                | все приложение DartExpense
    <script type="application/dart" src="DartExpense.dart"> ← Тер <script>
                                                                ← загружает весь код
                                                                приложения
  </script>
    <script src="http://dart.google...snip.../client/dart.js"> ← Скрипт
                                                                распознавания
                                                                поддержки Dart
  </script>
</body>
</html>
```

Совет. В JavaScript для определения того, поддерживается браузер Dart или нет, можно проверить наличие функции `navigator.webkitStartDart()`.

Тег `script` с типом `application/dart` загружает начальный `.dart`-скрипт, который, в свою очередь, загружает все остальные файлы, указанные в предложениях `part` и `import` (кроме встроенных библиотек). После того как все приложение загружено и разобрано ВМ Dart, вызывается функция `main()`. Полная компиляция приложения на этапе загрузки дает существенный выигрыш, по сравнению с JavaScript-приложением. Однако у всего есть цена – в отличие от JavaScript, написанный на Dart код нельзя динамически модифицировать во время выполнения – такая практика получила название *monkey-patching* (латание по живому).

Dart-приложение может быть представлено одним монолитным файлом или, что гораздо лучше, разбито на несколько частичных файлов, как было показано в главе 5. Dart рассматривает частичные файлы, как если бы они были частями одного физического библиотечного файла, то есть поведение приложения не зависит от того, находится оно в одном файле или в нескольких. Это просто удобный способ организации кода.

Совет. Начинайте разрабатывать приложение с одного файла, а затем – по мере выявления взаимосвязанной функциональности – переносите классы и функции в частичные файлы.

Приложение DartExpense состоит из одной библиотеки. Оно импортирует две базовые библиотеки – `dart:html` и `dart:json`, а его собственные классы и функции находятся в частичных файлах, как показано на рис. 10.4.



Рис. 10.4. Приложение DartExpense разбито на несколько файлов, связанных в единое целое предложением `part`

Бывает, что по мере разработки Dart-приложения выявляется иная логическая структура кода. Это может стать причиной изменения организации файлов; например, классы моделей можно поместить в одну библиотеку, а функции доступа к данным – в другую, особенно если вы планируете повторно использовать их в других приложениях.

10.1.3. Поток выполнения в Dart-приложении

Выполнение любого Dart-приложения начинается с функции `main()`, которая автоматически вызывается после загрузки всего кода, и приложение `DartExpense` – не исключение. Из функции `main()` обычно вызывается код конструирования пользовательского интерфейса и присоединения обработчиков событий, в частности событий нажатия кнопки. После того как этот код отработает, приложение не завершается, но находится в пассивном состоянии, ожидая возникновения браузерного события.

События могут быть внутренними, например завершение HTTP-запроса, или внешними, например взаимодействие с пользовательским интерфейсом или нажатие кнопки **Назад** в браузере. Приложение продолжает ждать событий до тех пор, пока не будет выгружено из браузера – то ли в результате закрытия вкладки, то ли вследствие перехода на другой URL-адрес. Описанный поток управления показан на рис. 10.5.

Dart – однопоточный язык, как и JavaScript. Эта особенность – осознанное проектное решение, поскольку одно из основных требований к Dart – возможность трансляции его на JavaScript. Следовательно, события обрабатываются по очереди, по мере того как приложение получает их от браузера.

Как видите, функция `main()` особенно важна. Если бы она построила пользовательский интерфейс, не присоединив обработчики событий, то интерфейс появился бы на экране, но взаимодействовать с ним было бы невозможно. С другой стороны, если бы приложение обратилось куда-то за данными, не построив пользовательского интерфейса, то данные были бы загружены, только пользователь их не увидел бы.

Таким образом, функция `main()` в приложении `DartExpense` производит следующие действия.



Рис. 10.5. Приложение продолжает работать и ждет событий

1. Найти в HTML-файле `<div>` с идентификатором `dartexpense`; именно внутри него будет размещен пользовательский интерфейс приложения.
2. Создать источник данных, к которому приложение будет обращаться за данными. Пока что источник будет подставным.
3. Создать объект класса `AppController`, передав его `<div>`, и источник данных.
4. Вызвать метод `buildUI()` этого объекта, который построит основные элементы ГИП, в том числе верхний и нижний колонтитулы, а также контейнеры `<div>` для обоих представлений и панели действий. Пока еще никакие обработчики событий не подключены.
5. Вызвать метод `loadFirstView()` объекта `AppController`, передав ему в качестве параметра положение окна. Этот метод отвечает за подготовку начального представления `ListView`, в котором будет показан список расходов. Он же присоединяет обработчики событий к кнопкам в этом представлении.

Объект `AppController` хранится в закрытой переменной библиотеки, к которой дает доступ метод чтения `app()`. Это позволяет обратиться к контроллеру из любого места приложения.

Разумеется, лучше один раз увидеть код, поэтому в листинге ниже показано, как выглядит функция `main()` из файла `DartExpense.dart`.

Листинг 10.2. DartExpense.dart: функция main()

```
library "dartexpense";  ← Приложение объявляется как библиотека

import "dart:html";  | Импортируются
import "dart:json";  | встроенные библиотеки

part "models.dart";  | Считываются
// ...прочие частичные файлы опущены...  | частичные
part "app.dart";      | файлы

part "mocks/mock_data.dart";

AppController _app;  |
AppController get app => _app;  | Закрытый на уровне библиотеки экземпляр AppController
                                | и возвращающий его метод доступа

void main() {  ← Выполнение начинается с функции main()
    var uiContainer = query("#dartexpense");
    var dataSource = new MockData();
    _app = new AppController(uiContainer, dataSource);  | Найти контейнер <div>
                                                         | и создать источник данных;
                                                         | они нужны для создания
                                                         | экземпляра AppController

    app.buildUI();  | Строится пользовательский интерфейс;
    app.loadFirstView();  | в него загружается первое представление
}

```

Сконструировав ГИП, содержащий начальное представление, функция `main()` завершается. Приложение переходит в фазу ожидания событий, в ходе которой пользователь может взаимодействовать с ним: нажать кнопку **Add** для создания новой записи о расходе или кнопку **Edit** для изменения существующей записи.

Памятка

- Основная цель Dart – создание сложных веб-приложений.
 - Одностраничное веб-приложение связывает код представления с данными на стороне клиента, а не сервера.
 - Удобно начинать разработку приложения с локального режима, потому что так оно доступнее и быстрее – как для самих разработчиков, так и для пользователей.
 - Dart-приложение может состоять из нескольких частичных файлов, объединенных в единое целое головным скриптом.
 - Исполнение начинается с функции `main()`, которая должна построить пользовательский интерфейс и присоединить обработчики событий.
-

Итак, мы теперь знаем, из чего состоит Dart-приложение. Следующий шаг – построить пользовательский интерфейс с помощью

библиотеки `dart:html`, которая позволяет создавать различные элементы из программного кода и HTML-разметки.

10.2. Конструирование пользовательского интерфейса с помощью `dart:html`

В этом разделе мы займемся конструированием пользовательского интерфейса из HTML-разметки и классов в библиотеке `dart:html`. Мы подробно рассмотрим вышеупомянутые методы `buildUI()` и `loadFirstView()`.

Библиотека `dart:html` предоставляет набор API и классов для работы с элементами модели DOM. В этом смысле она аналогична библиотеке jQuery для JavaScript и, в частности, включает средства для поиска конкретных элементов в DOM, для доступа к атрибутам и дочерним элементам и для манипуляций с обработчиками событий.

А где же библиотеки виджетов?

На момент написания этой книги еще не было встроенной библиотеки элементов пользовательского интерфейса, хотя в Сети есть немало сторонних библиотек с открытым исходным кодом. Вместо того чтобы работать с библиотечными абстракциями типа `LayoutPanel` или `MenuBar` (имеющимися, например, в библиотеке виджетов GWT), мы будем иметь дело с низкоуровневыми классами Dart, такими как `DivElement` или `ButtonElement`. При этом мы получаем точный контроль над HTML-разметкой, тем более что во многих случаях работаем прямо на уровне HTML. В результате вы научитесь применять Dart для манипуляций моделью DOM, и это умение пригодится и позже, когда появятся высокоуровневые библиотеки виджетов.

Заглядывая вперед, можно предположить, что в Dart будет применяться технология веб-компонентов, позволяющая создавать компоненты с помощью шаблонов теневой модели DOM (Shadow DOM). Этими компонентами можно будет манипулировать как стандартными HTML-элементами. Дополнительные сведения можно найти в системе по запросу «Dart web components».

10.2.1. Интерфейс *Element*

Основой библиотеки `dart:html` является интерфейс `Element`. В нем определены два именованных конструктора — `Element.html()` и `Element.tag()`, с которыми мы уже встречались в главе 3. Оба конструируют HTML-элемент, но по-разному.

С другой стороны, конструктор `Element.html(String html)` создает иерархию HTML-тегов с одним ограничением: все дочерние элементы должны быть потомками единственного элемента верхнего уровня. Например, воспользовавшись синтаксисом многострочных строк в Dart (строка начинается и заканчивается тремя знаками кавычки: `"""text"""`), мы можем определить такой блок HTML, состоящий из одного `<div>`'а и вложенных в него элементов:

```
var actionDiv = new Element.html("<<<
  <div id='actions'>Select:
    <button id='saveButton' class='primary, selected'>Save</button>
    <button id='cancelButton' name='cancel'>Cancel</button>
  </div>'><<<");
```

- ❑ `query(String selector);`
- ❑ `queryAll(String selector).`

Эти методы позволяют искать соответственно один или несколько подходящих дочерних элементов. Так, в следующем примере мы сначала находим кнопку `saveButton`, а затем все кнопки в предварительно объявленном элементе `actionDiv`:

```
var saveButton = actionDiv.query(".primary");    ← Первый элемент с классом primary
var cancelButton = actionDiv.query("#cancelButton"); ← Первый элемент с идентификатором cancelButton
var buttonList = actionDiv.queryAll("button");    ← Все элементы типа button
var primaryButtons = actionDiv.queryAll(".primary .selected"); ← Все элементы, имеющие оба класса primary и selected
var cancelButtonList = actionDiv.queryAll("[name='cancel']"); ← Все элементы, для которых атрибут name равен cancel
```


Эти методы принимают CSS-селектор в качестве параметра – точно так же, как функция `$()` в jQuery. CSS-селекторы позволяют находить один или несколько элементов с заданными свойствами, например идентификатором (он определяет элемент однозначно в пределах страницы) или классом (элементов с одним и тем же классом может быть несколько).

В интерфейсе `Element` определены свойства и методы для программного манипулирования элементами. В частности, этот интерфейс реализуют объекты браузера `document` и `window`, представленные свойствами верхнего уровня в библиотеке `dart:html`.

Эмуляция функции `$()` из библиотеки jQuery?

`$` – допустимое имя функции в Dart, так что если вам нравится функция `$()` из библиотеки jQuery, то можете эмулировать в Dart, включив в свою библиотеку функцию, которая делегирует работу методу `document.query()`:

```
$(String selectors) => query(selectors);
```

В результате вы получите в своей библиотеке функцию `$`, которую можно использовать точно так же, как в jQuery:

```
$("#actionDiv").innerHTML = "No Actions";
```

Все остальные HTML-элементы также реализуют интерфейс `Element`; некоторые его наиболее употребительные свойства и методы перечислены в табл. 10.1.

Таблица 10.1. Основные свойства и методы интерфейса `Element`

Свойство или сигнатура метода	Назначение
<code>Element.tag(String tagName)</code>	Конструктор для создания элемента по имени тега
<code>Element.html(String html)</code>	Конструктор для создания элемента из HTML-разметки
<code>String id</code>	Прочитать или установить идентификатор элемента
<code>String innerHTML</code>	Прочитать или установить внутреннюю HTML-разметку элемента
<code>List<Element> children</code>	Доступ к коллекции дочерних элементов
<code>List<Element> queryAll(String selectors)</code>	Выбор группы подходящих дочерних элементов. Например, <code>myDiv.queryAll("button")</code> возвращает все кнопки внутри элемента <code>myDiv</code>
<code>Element query(String selectors)</code>	Выбор одного дочернего элемента
<code>ElementEvents on</code>	Доступ к событиям элемента для присоединения обработчика

Таблица 10.1. Основные свойства и методы интерфейса *Element* (окончание)

Свойство или сигнатура метода	Назначение
AttributeMap attributes	Доступ к атрибутам элемента
Классы коллекций	Доступ к классам коллекций элементов

У конкретных HTML-элементов интерфейс *Element* дополнен рядом свойств и методов; например, в интерфейсе *InputElement* имеется свойство *value* для доступа к введенному значению. При создании конкретных элементов – `<div>`, `<input>`, `<button>` и прочих – можно использовать классы, наследующие *Element*, и сохранять результат в строго типизированной переменной. Это позволит инструментальным средствам проверить правильность употребления атрибутов элементов:

```
InputElement textbox = new Element.tag("input");
textbox.value = "Some text";
```

← В интерфейсе *InputElement*
свойство *value* существует

В текущей версии интерфейсу *Element* наследуют примерно 70 подынтерфейсов; наиболее употребительные перечислены в табл. 10.2. Дополнительные сведения можно найти в справочной документации по Dart API на сайте <http://api.dartlang.org>.

Таблица 10.2. Подынтерфейсы, наследующие интерфейсу *Element*

BodyElement	ButtonElement	CanvasElement
DivElement	EmbedElement	FormElement
HeadingElement	ImageElement	InputElement
LabelElement	LinkElement	OptionElement
ParagraphElement	ScriptElement	SelectElement
SpanElement	StyleElement	TableElement

Для большинства стандартных тегов HTML определены специальные классы, но всегда можно воспользоваться свойством базового класса *attributes* для доступа к любому атрибуту, не присутствующему явно в классе конкретного элемента, например:

```
textbox.attributes["value"] = "Текст";
```

10.2.2. Конструкторы элементов в действии

Чтобы понять, как работать с конструкторами элементов, рассмотрим метод `AppController.buildUI()`, который вызывается из функции `main()` и порождает заготовку пользовательского интерфейса, показанную на рис. 10.6. Заготовка содержит верхний и нижний колонтитулы, а также пустые элементы `<div>` с идентификаторами `content` (для будущего представления) и `actions` (для кнопок).

Примечание. В файле `dartexpense.css`, на который ссылается головной файл `DartExpense.html`, находятся стили, определяющие внешний вид пользовательского интерфейса. Отделение содержимого от дизайна – правильный подход, потому что таким образом можно без труда менять облик приложения, например использовать разные таблицы стилей для настольных и мобильных браузеров. Особенности работы с CSS выходят за рамки этой книги, но по этому вопросу существует немало хороших книг и интернет-ресурсов.

В начале работы приложения `DartExpense` метод `buildUI()` конструирует заготовку пользовательского интерфейса, которая содержит секции верхнего и нижнего колонтитулов, а также элементы `<div>` с идентификаторами `content` и `actions`, где будут находиться, соответственно, представление и кнопки. Все это помещено в корневой элемент `<div>`, переданный конструктору объекта `AppController`. Метод `buildUI()` и созданная им HTML-разметка показаны на рис. 10.7.

После того как заготовка интерфейса построена и ссылки на два важных элемента `<div>` сохранены в закрытых свойствах `_actionsContainer` и `_viewContainer` объекта `AppController`, функция `main()` вызывает метод `app.loadFirstView()`. Его задача – загрузить нужное представление в `<div>`’ы `content` и `action`, удалив, если требуется, ранее находившееся в них представление. Оба представления реализуют интерфейс `View`, в котором определены два свойства типа `DivElement`: `viewElement` и `actionsElement`. При создании представления, например `ListView`, необходимо заполнить эти элементы подходящей HTML-разметкой (какой именно, мы скоро увидим). Метод `AppController.updateView()` сначала стирает текущее представление, на которое ссылаются свойства `_viewContainer` и `_actionsContainer`, а затем добавляет элементы, составляющие новое представление. В листинге ниже показан содержащийся в классе `AppController` код заполнения `<div>`’ов `content` и `actions`.

Метод `buildUI()` класса `AppController` строит заготовку пользовательского интерфейса, создавая нужные элементы

```
buildUI() {
    var header = new Element.html("""
        <header class='section'>
            DartExpense
        </header>""");
    _uiRoot.children.add(header);

    var content = new Element.tag("div");
    content.id= "content";
    content.classes.add("section");
    _uiRoot.children.add(content);
    this._contentContainer = content;

    var actions = new Element.tag("div");
    actions.id= "actions";
    actions.classes.add("section");
    _uiRoot.children.add(actions);
    this._actionsContainer = actions;

    var footer = new Element.html("""
        <footer class='section'>
            Offline
        </footer>""");
    _uiRoot.children.add(footer);
}
```

HTML-разметка заготовки

```
<div id='dartexpense'>

    <header class='section'>
        DartExpense
    </header>

    <div id='content'></div>

    <div id='actions'></div>

    <footer class='section'>
        Offline
    </footer>

</div>
```

Ссылки на элементы `actions` и `content` хранятся в закрытых свойствах объекта `AppController`. Это позволяет обращаться к ним напрямую без привлечения метода `query()`

Новые элементы становятся потомками `<div>`'а с идентификатором `#dartexpense`, который был передан конструктору `AppController` и сохранен в закрытом свойстве `_uiRoot`

Рис. 10.7. Метод `buildUI()` вызывает конструкторы класса `Element` для построения заготовки пользовательского интерфейса

Листинг 10.3. Загрузка представления в `<div>`'ы `content` и `actions`

```
class AppController {
    ...остальные методы и свойства опущены...

    DivElement _viewContainer; | Контейнеры для представления
    DivElement _actionsContainer; | и связанных с ним действий

    List<Expenses> get expense => ...опущено... ← Свойство для хранения списка расходов,
                                                полученных от источника данных

    loadFirstView() {
        var view = new ListView(this.expenses); | Создаем новый объект ListView,
        updateView(view); | передавая ему список расходов
    }
}
```

```

updateView(View view) {
  _viewContainer.children.clear();
  _viewContainer.children.add(view.viewElement);
  _actionsContainer.children.clear();
  _actionsContainer.children.add(view.actionsElement);
}
}

```

Вставляем элементы `viewElement` и `actionsElement` из `ListView` в нужные места заготовки

Метод `elements.clear()`, с помощью которого удаляются дочерние элементы из контейнеров `_viewContainer` и `_actionsContainer`, — часть стандартного интерфейса Dart `List`; он удаляет элементы из модели DOM браузера. Метод `elements.add()` добавляет новые элементы в DOM, после чего приложение готово к обработке событий.

Примечание. Используемые в приложении `DartExpense` типы, в частности интерфейс `View` и классы `ListView` и `EditView`, созданы специально для этого приложения и не являются частью какого-то встроенного в Dart каркаса модель-вид. На момент написания этой книги такого каркаса вообще не существовало, но разработчики Dart намерены рано или поздно создать его.

10.2.3. Организация взаимодействия с представлениями и элементами

В представлении `ListView` для отображения списка расходов применяется HTML-таблица. Список расходов — первое, что видит пользователь, обратившийся к приложению `DartExpense` (рис. 10.8).

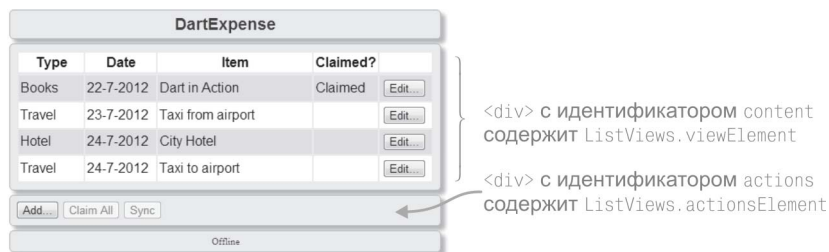


Рис. 10.8. Содержимое и действия в представлении `ListView`, показаны подставные данные

Всего в этом приложении два представления. В классе `ViewType`, показанном ниже, имеется закрытый константный конструктор для создания представлений обоих типов, `ViewType.LIST` и `ViewType.EDIT`:

```
class ViewType {
    final String name;

    const ViewType._withName(this.name); ← Закрытый конструктор
    static final ViewType LIST = const ViewType._withName("list");
    static final ViewType EDIT = const ViewType._withName("edit");
}

```

Инициализируются константы LIST и EDIT

В Dart нет перечислений enum

В отличие от C# и Java, в язык Dart не встроены перечисления, то есть тип, позволяющий определить именованный набор констант. Элегантной альтернативой является определение класса с закрытым константным конструктором и фиксированным списком статических финальных полей.

Переключение представлений

Экземпляры класса `ViewType` используются в библиотечной функции верхнего уровня `navigate()`, которая является основным инструментом навигации в приложении. Эта функция вызывается при нажатии на кнопку **Edit** и позволяет перейти от одного представления к другому. Для этого она создает новое представление, исходя из значения `ViewType`, и передает его методу `updateView()` объекта `AppController`, который, как мы уже видели, уничтожает текущее представление и заново заполняет контейнеры `contentContainer` и `actionsContainer`:

```
void navigate(ViewType view, int id) {
    if (view == ViewType.LIST) {
        app.updateView(new ListView(app.expenses));
    }
    else if (view == ViewType.EDIT) {
        app.updateView(new EditView(app.getExpenseById(id)));
    }
}

```

Новому объекту ListView
← передается список расходов

Новому объекту EditView
передается новая или
существующая статья
расхода

Реализация очень проста, чтобы не усложнять примера. Было бы эффективнее кэшировать представления в объекте `AppController`, а не создавать их каждый раз заново.

Конструирование представления *ListView*

Метод `AppController.loadFirstView()`, который вызывается из `main()`, обращается к конструктору `ListView(List<Expense> expenses)`, переда-

вая ему список расходов, который нужно показать на экране. Этот список конструктор далее передает функции `refreshUI()`, которая строит HTML-таблицу из экземпляров `Element` и сохраняет результат в свойстве представления `viewElement`, как показано в листинге 10.4. Это довольно длинный код, но по существу он просто создает элемент, соответствующий таблице, и добавляет в эту таблицу заголовок и строки. Для создания строк в цикле обхода списка расходов вызывается закрытая функция `_getRowElement(Expense ex)`, которой передается текущий элемент списка. Для каждой строки создается и добавляется в коллекцию `table.children` объект `TableRowElement`. Кроме того, именно в функции `_getRowElement(ex)` мы впервые добавляем обработчик события, а именно события `on.click` нажатия на кнопку **Edit** в правой части строки.

С обработчиками событий мы ближе познакомимся в следующем разделе, а пока достаточно знать, что этот обработчик вызывает библиотечную функцию `navigate(ViewType, int id)`. Интересно здесь то, что анонимная функция-обработчик (выделена в листинге 10.4 курсивом), которая вызывает `navigate()`, образует замыкание вокруг объекта статьи расхода, переданного `_getRowElement(ex)`. Иными словами, даже после того как метод возвращает управление, обработчик события сохраняет ссылку на `ex.id` (подробнее о замыканиях см. главу 4). Идентификатор расхода `ex.id` — число, автоматически генерируемое в классе `Expense`; оно необходимо функции `navigate()` наряду с именем представления, на которое мы собираемся переключиться.

Листинг 10.4. Класс `ListView`, который строит HTML-таблицу расходов

```
class ListView implements View {
  DivElement viewElement;    ← Свойства для реализации интерфейса View
  DivElement actionsElement;

  ListView(List<Expense> expenses) { ← Конструктор обновляет контейнеры content и actions
    refreshUI(expenses);
    _buildActions();
  }

  refreshUI(List<Expense> expenses) {
    viewElement = new Element.tag("table"); ← Создается новый пустой элемент table
    var head = new Element.html("<thead>" ← Создается заголовок — элемент <thead>
    <thead>
      <td class="type">Type</td>
      <td class="date">Date</td>
```

```

        <td class="detail">Item</td>
        <td class="amount">Amount</td>
        <td class="claimed">Claimed?</td>
        <td class="edit">&nbsp;</td>
    </thead>""");

viewElement.children.add(head); ← Заголовок добавляет в таблицу

for (Expense ex in expenses) {
    var rowElement = _getRowElement(ex); | Цикл по всем расходам, на каждой итерации
    viewElement.children.add(rowElement); | добавляется строка TableRowElement
}

TableRowElement _getRowElement(Expense ex) {

    TableRowElement row = new Element.tag("tr"); ← Создается элемент
                                                    TableRowElement

    row.children.add(new Element.html('<td>${ex.type.name}</td>')); |
    ... остальные столбцы опущены ...                               |
    row.children.add(new Element.html('<td>${ex.detail}</td>'));      |
                                                                    | В строку добавляются
                                                                    | столбцы с данными

    var editCol = new Element.html('""' |
        <td>
            <button>Edit...</button>
        </td>"""); | Добавляется столбец с кнопкой Edit
    row.children.add(editCol);

    editCol.query("button").on.click.add((event) { | Обработчик события нажатия
        navigate(ViewType.EDIT, ex.id);           | образует замыкание
    });                                           | вокруг ex.id
    return row;
}

_buildActions() {
    ...опущено...
}
}

```

Код в методе `ListView.refreshUI()` перебирает все элементы списка `expenses` и создает строки в HTML-таблице. В реальном приложении подобная функциональность могла бы понадобиться в нескольких местах, поэтому было бы лучше написать обобщенную функцию, которая умела бы динамически создавать таблицу на основе переданных ей данных и не была бы жестко связана с классом `Expense`.

10.2.4. Построение простой обобщенной сетки

Для построения обобщенной сетки мы должны уметь динамически создавать таблицу, исходя из переданного списка объектов и конфигурационных данных, определяющих, какие должны быть показаны столбцы. Например, в случае объектов `Expense` в каждой строке таблицы нужно показать следующие значения:

```
expense.type.name, expense.date, expense.detail, expense.amount
```

В отличие от JavaScript, мы не можем динамически обращаться к этим свойствам как к атрибутам объектов. В JavaScript объект одновременно является словарем, содержащим пары ключей и значений, поэтому можно передать список объектов и список имен столбцов и обращаться к соответствующим свойствам объектов как к элементам словаря. В примере ниже приведена тривиальная реализация на JavaScript:

```
// JavaScript
function getDynamicTable(objects, columnList) { ← JavaScript-функция, которая
    var table = "<table>";                               динамически строит таблицу <table>

    for(item in objects) { ← Внешний цикл; каждому объекту соответствует строка
        table += "<tr>";

                                Внутренний цикл; каждому свойству
        for (propertyName in columnList) { ← соответствует столбец
            var cellText = item[propertyName]; ← Динамически получаем значения свойств
            table += "<td>" + cellText + "</td>";
        }

        table += "</tr>";
    }

    table += "</table>";
    return table;
}
```

В Dart такой способ доступа к свойствам не поддерживается. Мы могли бы написать на Dart похожую функцию, которая принимает `List<Map>` вместо `List<Object>` (или `List<Expense>`), но для этого вызывающая функция должна была бы обойти все объекты `Expense`

в списке, преобразовать каждый из них в словарь и передать получившийся список словарей. Лучше поступить по-другому и наделять функцию `getDynamicTable()` возможностью извлекать значение свойства, только когда оно необходимо. Такой подход особенно привлекателен с точки зрения производительности, когда требуется реализовать разбиение списка на страницы – например, в цикле построения таблицы за раз обрабатывается не более 10 элементов.

И это можно сделать, если представить конфигурационные данные о столбцах в виде списка функций. Идея следующая: вызывающая программа определяет анонимные функции, которые знают, как получать значение одного столбца, и передает `getDynamicTable()` набор таких функций. Вместо того чтобы передавать список столбцов, мы передаем словарь, ключом которого является имя столбца (или свойства), а значением – функция, возвращающая это свойство. Подобная функция ожидает получить объект в качестве параметра, а возвращает строку. Определение соответствующего типа приведено ниже:

```
typedef String GetValueFunc(dynamic item);
```

Имея такой тип функции, мы можем переписать `refreshUI(List<Expense>)`, так чтобы она определяла конфигурацию столбцов и передавала ее функции `getDynamicTable()`:

<pre>refreshUi(List<Expense> expenses) { var columnConfig = new Map<String, GetValueFunc>(); columnConfig["detail"] = (expense) => expense.detail; columnConfig["type"] = (expense) => expense.type.name; columnConfig["date"] = (expense) => expense.date.toString(); columnConfig["amount"] = (expense) => expense.amount.toString(); viewElement = getDynamicTable(expenses, columnConfig); }</pre>	<code>columnConfig</code> объявляется как словарь, сопоставляющий ← функции строкам ← Для столбца "detail" возвращается свойство <code>expense.detail</code> ← Список расходов и словарь <code>columnConfig</code> передаются функции

Теперь функция `getDynamicTable()` может получить любое свойство объекта, используя ассоциированную с этим свойством функцию в переданном словаре:

```
var getValueFunc = columnConfig["detail"];
var detailText = getValueFunc(expense);
```

Итак, мы получили способ динамически извлекать значения произвольных свойств объекта, например свойства "detail". Полный код функции `getDynamicTable()` приведен в листинге 10.5.

Листинг 10.5. Функция `getDynamicTable()`, которая динамически строит HTML-таблицу

```
typedef String GetValueFunc(dynamic item); ← Определение типа функции GetValueFunc

TableElement getDynamicTable(List items,
↪      Map<String, GetValueFunc> columnConfig) {

    var table = new Element.tag("table"); ← Создаем пустой элемент table

    var header = new Element.tag("thead");
    for (String colName in columnConfig.keys) {
        header.children.add(new Element.html("<td>$colName</td>"));
    }
    table.children.add(header);

    for (var item in items) {
        var row = new Element.tag("tr");
        table.children.add(row);

        for (String colName in columnConfig.keys) { ← Внутренний цикл: для каждого столбца...

            var getValueFunc = columnConfig[colName]; ← ...находим функцию getValueFunc() и...

            var textValue = getValueFunc(item); ← ...с ее помощью получаем значение
                                                    свойства текущего объекта

            row.children.add(new Element.html("<td>$textValue</td>")); ←
        }
        Добавляем новый столбец в строку
    }

    return table;
}
```

Это эффективный способ обобщенного доступа к свойствам. Вместо того чтобы обращаться к свойству объекта напрямую, мы можем передать объект специфичной для свойства функции `getValueFunc()`, которую предоставила вызывающая программа. При таком подходе можно получать значения свойств динамически, не зная заранее, как

свойства называются. На рис. 10.9 показана таблица, созданная с помощью функции `getDynamicTable()`.

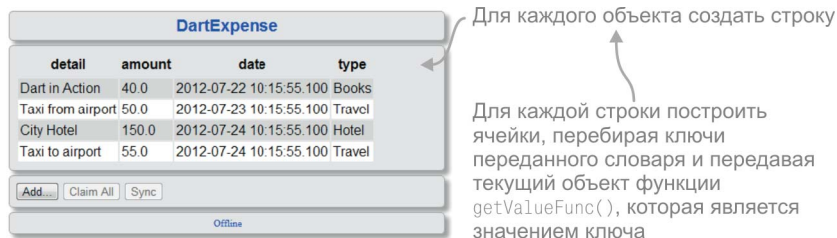


Рис. 10.9. Таблица, созданная обобщенной функцией генерации таблиц

В этом учебном примере функция `getDynamicTable()` довольно примитивна, но ценой небольших усилий можно наделить ее дополнительными возможностями. Оставляю в качестве упражнения для читателя следующие задачи: улучшить описание строки заголовка (не ограничиваясь именами ключей в словаре `columnConfig`), передать список операций, которые можно выполнять над строкой (например, изменить и удалить), и связать со строками и столбцами классы CSS.

Мы же вернемся к исходному варианту кода формирования `ListView`, потому что в нем создается также кнопка **Edit**, которая понадобится нам в следующем разделе, когда мы займемся навигацией.

Выше в этом разделе было показано, как присоединить к кнопке **Edit** обработчик события. В следующем разделе, когда мы добавим в приложение `DartExpense` представление для редактирования расходов, обработчики событий будут рассматриваться подробнее.

Памятка

- Интерфейс `Element` – основа библиотеки `dart:html`, он реализован свойствами `document` и `window`, находящимися на верхнем уровне библиотеки, а также всеми элементами HTML.
- Многим элементам HTML соответствуют специализированные подынтерфейсы, наследующие `Element`, например `ButtonElement` и `InputElement`.
- В интерфейсе `Element` определены два конструктора: `Element.tag()` создает один пустой элемент, в который можно программно добавлять потомков, а `Element.html()` создает из HTML-разметки один элемент, который уже может содержать потомков.
- В настоящее время в состав Dart SDK не входит библиотека виджетов, поэтому строить различные элементы пользовательского интерфейса, например списки и сетки, приходится самостоятельно – либо для конкретного случая, либо обобщенные, пригодные для повторного использования.

10.3. Обработка браузерных событий с помощью dart:html

В предыдущем разделе мы видели, как сконструировать представление и присоединить его к браузеру. Но это только половина истории. Функция `main()` должна не просто построить пользовательский интерфейс, но и связать с ним обработчики событий. Этой темой мы сейчас и займемся.

Обработчик события – это присоединяемая к HTML-элементу функция, которая реагирует на событие определенного вида. Самый типичный пример события – нажатие кнопки, например кнопки **Add** в списковом представлении из приложения `DartExpense`. Всем браузерным событиям соответствует свойство `on` интерфейса `Element`, являющееся реализацией интерфейса `ElementEvents`. Эта реализация предоставляет список событий по имени (полезно при автозавершении в редакторе `Dart Editor`) и переопределяет оператор `[]`, так чтобы можно было присоединять обработчики к заранее неизвестным событиям (быть может, для поддержки какой-то экспериментальной технологии в браузере), а также динамически. Каждое событие представляет собой реализацию интерфейса `EventListenerList`, который позволяет связывать с одним событием несколько обработчиков. В листинге 10.6 показаны два способа присоединения обработчика события к кнопке **Add**, созданной рассмотренным выше методом `_buildActions()` класса `ListView`. Код функций-обработчиков выделен полужирным шрифтом.

Листинг 10.6. Присоединение обработчика события к кнопке Add

```
class ListView {
  DivElement viewElement;
  DivElement actionsElement;

  ListView(List<Expense> expenses) { ← Конструктор ListView из предыдущего раздела
    refreshUI(expenses);
    _buildActions();
  }

  _buildActions() {
    actionsElement = new Element.tag("div");

    ButtonElement button = new Element.tag("button"); ← Создаем пустой элемент button
    button.text = "Add...";
```

```
actionsElement.children.add(button);
```

 ← Добавляем его в <div> actionsElement

```
button.onClick.add( (event) {
    navigate(ViewType.EDIT, null);
});
```

Добавляем
обработчик события
в именованное свойство

```
button.on["mouseover"].add( (event) {
    print("button mouseOver")
});
```

Добавляем обработчик
события, определяемого
строкой "mouseover"

```
}
```

```
...остальные методы опущены...
```

```
}
```

Функция, добавленная в список `EventListenerList` для указанного события, например `click` или `mouseover`, выполняется, когда браузер генерирует это событие для данного элемента. С одним событием одного элемента можно связать несколько обработчиков. Обработчики можно также связывать с разными событиями одного элемента. Возможно также перехватывать события, предназначенные для дочерних элементов; как это делается, мы увидим в следующем разделе.

В свойстве `on` типа `ElementEvents` из интерфейса `Element` определено примерно 50 именованных событий. Наиболее употребительные перечислены в табл. 10.3.

Таблица 10.3. Некоторые именованные события, доступные через свойство `on` типа `ElementEvents`

События	Назначение
<code>click, doubleClick</code>	Возникает, когда пользователь щелкает мышью по элементу
<code>focus, blur</code>	Возникает, когда элемент получает или теряет фокус
<code>keypress, keyDown, keyUp</code>	Возникает при нажатии клавиши в момент, когда элемент владеет фокусом
<code>mouseover, mouseMove, mouseUp</code>	Возникает, когда указатель мыши проходит над элементом
<code>mousedown, mouseUp</code>	Возникает, когда нажата или отпущена кнопка мыши
<code>touchStart, touchMove, touchEnd</code>	Возникает в момент касания сенсорного устройства

Это лишь часть событий, генерируемых браузером. В разделе, посвященном интерфейсу `ElementEvents`, на сайте <http://api.dartlang.org> их перечислено гораздо больше.

10.3.1. Управление порядком прохождения события в браузере

Когда пользователь нажимает кнопку **Add** в приложении `DartExpense`, браузер генерирует событие, которое распространяется от элемента верхнего уровня `document` к кнопке через все промежуточные родительские элементы. Достигнув конечной цели, событие начинает обратный путь по иерархии элементов (эта фаза называется *всплытием*), и при этом вызываются все обработчики данного события, присоединенные к помещаемым элементам. Таким образом, при обычных обстоятельствах родительские элементы могут увидеть и обработать событие, предназначенное потомку, но только после того, как потомок на него уже отреагировал. В приложении `DartExpense` кнопка **Add** является конечным узлом иерархии, изображенной на рис. 10.10. И `<div> actions`, и кнопка **Add** зарегистрировали обработчики события `click`, но обработчики, связанные с кнопкой, выполняются раньше.

Таков порядок прохождения события по умолчанию; распространение вниз по иерархии называется *фазой спуска* (*capture*), а вверх по иерархии – *фазой всплытия* (*bubbling*). Но у этого правила есть исключения: некоторые события, например `focus`, не всплывают. Полный перечень можно найти в разделе Википедии `DOM_events`. Однако в программе также можно предпринять некоторые шаги для модификации обычного порядка прохождения события.

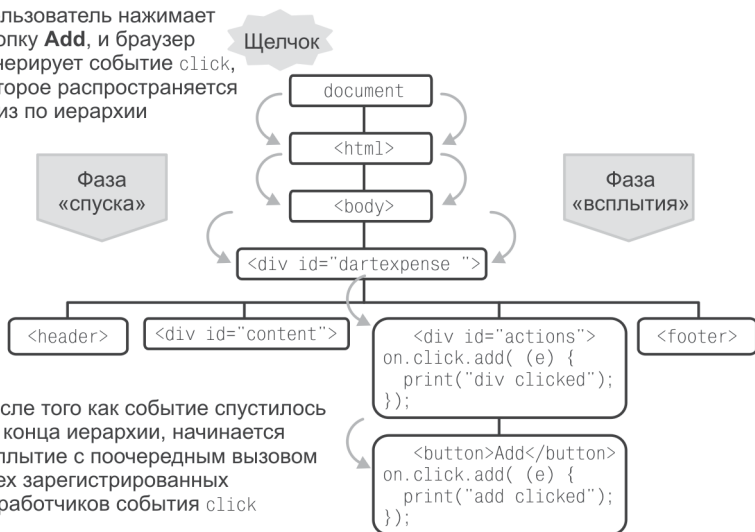
Модификация фазы всплытия

Чтобы вообще предотвратить всплытие события, мы можем вызвать метод `stopPropagation()` объекта `event`, который передается обработчику события в качестве параметра:

```
button.on.click.add( (event) {  
  print("add clicked");  
  event.stopPropagation();  ← Препятствует распространению события вверх по иерархии  
});
```

В результате событие `click` не приведет к вызову обработчиков `on.click`, присоединенных к другим элементам, однако все обработчики нажатия, сколько бы их ни было, присоединенные к тому же

1. Пользователь нажимает кнопку **Add**, и браузер генерирует событие `click`, которое распространяется вниз по иерархии



2. После того как событие спустилось до конца иерархии, начинается всплытие с поочередным вызовом всех зарегистрированных обработчиков события `click`

3. На отладочной консоли браузера печатаются сообщения:

```
"add clicked"
```

```
"div clicked"
```

Рис. 10.10. По умолчанию событие сначала доставляется самому глубоко вложенному элементу в иерархии, а затем всплывает, вызывая обработчики, присоединенные к посещаемым по пути элементам

самому элементу, все же будут вызваны. К той же кнопке можно присоединить второй обработчик события нажатия:

```
button.on.click.add( (event) => print("второй обработчик события"));
```

Таким образом, обработчик, присоединенный к `<div>`'у `actions`, не будет вызван, но оба обработчика, присоединенные к кнопке, вызываются. Впрочем, и это можно предотвратить: если вызвать метод `stopImmediatePropagation()` события, то все последующие обработчики, присоединенные непосредственно к данному элементу, не вызываются.

Модификация фазы спуска

Иногда желательно, чтобы родительские элементы могли обработать событие до того, как его увидит конечный адресат. Для этой цели в сигнатуре метода обработки события предусмотрен необязательный второй параметр:


```
add(eventHandler, [bool useCapture=false])
```

Если параметр `useCapture` равен `true`, то обработчик вызывается на пути события вниз, еще до того, как его увидят дочерние элементы. На рис. 10.11 показано, как модифицировать установку обработчика события `click` для `<div>`'а `actions`, чтобы он вызывался на фазе спуска.

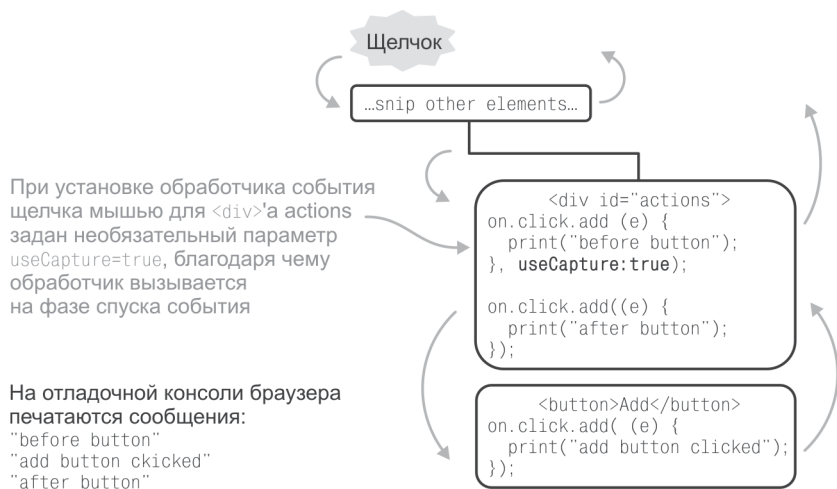


Рис. 10.11. Обработчики могут перехватывать событие на фазе его спуска, не дожидаясь вызова на фазе всплытия

Обработчики события, вызываемые на фазе спуска, также могут использовать методы `stopPropagation()` и `stopImmediatePropagation()` параметра `event`, чтобы предотвратить дальнейшее распространение события.

В последнем разделе этой главы мы рассмотрим интерфейс `Event`. С его методами `stopPropagation()` и `stopImmediatePropagation()` мы уже знакомы, но у интерфейса `Event`, как и у `Element`, есть ряд специализированных подтипов.

10.3.2. Наиболее распространенные типы событий

Вызывая функцию обработки события, браузер передает ей данные об этом событии в параметре `event` типа `Event`. Это обобщенный интерфейс, содержащий методы и свойства, общие для любых

событий, в том числе `stopPropagation()`, `timestamp` и `type`. Но для некоторых событий имеется дополнительная информация, поэтому базовому интерфейсу `Event` наследуют примерно 25 специализированных типов. Некоторые из них поддерживают такие современные технологии, как `SpeechRecognitionEvent` (для распознавания речи) и `DeviceOrientationEvent` (смена ориентации устройства), другие же более привычны. Так, в классе `MouseEvent` имеются координаты мыши в момент события, а в классе `KeyboardEvent` — код нажатой клавиши. В табл. 10.4 перечислены наиболее употребительные свойства класса `MouseEvent`.

Таблица 10.4. Наиболее употребительные свойства класса `MouseEvent`

Свойство	Назначение
<code>int button</code>	Номер нажатой кнопки мыши. Левая кнопка имеет номер 0
<code>int clientX</code> , <code>int clientY</code>	Координаты X и Y в клиентской области браузера в момент события
<code>int screenX</code> , <code>int screenY</code>	Экранные координаты X и Y в момент события
<code>bool altKey</code> , <code>bool ctrlKey</code> , <code>bool metaKey</code> , <code>bool shiftKey</code>	true, если в момент события была нажата одна из этих клавиш

Чтобы использовать эти сведения о конкретном событии, нужно обратиться к соответствующему свойству параметра `event`, переданного обработчику. Если вы заодно укажете тип этого параметра, то сможете воспользоваться всеми благами проверки типов со стороны инструментальных средств. В листинге ниже показано, как получить клиентские координаты мыши в момент события `click`.

Листинг 10.7. Использование специализированного типа `MouseEvent` для получения дополнительной информации о событии

		В списке параметров обработчика события
<code>button.on.click.add((MouseEvent event) {</code>		← задан тип параметра event
<pre> var x = event.clientX; var y = event.clientY; print("add button clicked, co-ordinates: \$x, \$y"); }); </pre>	Получаем координаты из event	

Порядок прохождения браузерного события — не какая-то особенность Dart; это часть поведения самого браузера, и, к сожалению, не во всех браузерах она реализована одинаково. Dart спроектирован

прежде всего для работы с современными браузерами, а это означает, что после трансляции на JavaScript большая часть приведенного здесь кода, скорее всего, не будет работать в IE6.

Совет. Если неполная обратная совместимость вас беспокоит, загляните на управляемый корпорацией Microsoft сайт www.ie6countdown.com, где написано «пришло время попрощаться с IE6».

Памятка

- Функции обработки событий можно добавлять в свойство `Element.on` либо по имени, например `on.click.add(...)`, либо по значению, например `on["click"].add(...)`.
 - Элемент может вызвать несколько обработчиков одного и того же события.
 - С одним элементом можно связать обработчики разных событий.
 - Возникшее событие сначала распространяется от элемента `document` вниз по иерархии родитель–потомок до конечного адресата. Это фаза спуска; по умолчанию на ней обработчики события не вызываются.
 - За фазой спуска следует фаза всплытия, когда событие распространяется вверх по иерархии; при этом поочередно вызываются все его обработчики.
 - Порядок прохождения события можно модифицировать в программе.
-

Это был длинный раздел, но без хорошего понимания браузерных событий невозможно строить интересные и отзывчивые пользовательские интерфейсы. В последующих главах мы рассмотрим также события, относящиеся к хранению данных в браузере и к сенсорным устройствам, но уже теперь вы, вероятно, представляете, как распространяются события. Дополнительные сведения об общих принципах работы с событиями в браузере см. на сайтах Mozilla Developer Network (https://developer.mozilla.org/en-US/docs/DOM/DOM_event_reference) и w3.org (www.w3.org/TR/DOM-Level-3-Events/#event-flow), где приведена полезная справочная информация.

10.4. Резюме

Глава получилась довольно длинной, мы рассмотрели дизайн одностраничного веб-приложения, способного работать в автономном режиме, что, с одной стороны, упрощает разработку, а с другой – предоставляет дополнительные возможности пользователю. Затем мы обсудили функцию `main()`, с которой начинается выполнение

Dart-приложения; она отвечает за конструирование пользовательского интерфейса и присоединение обработчиков событий.

Мы рассмотрели процедуру построения ГИП с помощью библиотеки `dart:HTML`, являющейся аналогом библиотеки `jQuery` и предназначенной для манипулирования моделью `DOM`. На момент написания этой книги в составе Dart SDK еще не было библиотеки виджетов, поэтому для работы с элементами HTML приходится прибегать к различным классам Dart, наследующим интерфейсу `Element`. По ходу конструирования интерфейса своего приложения мы поговорили о том, как можно было бы реализовать обобщенную сетку, которая принимает список произвольных объектов и представляет его в виде HTML-таблицы.

Наконец, мы познакомились с моделью событий в браузере и видели, как в Dart используются обработчики, позволяющие программе реагировать на события, и как модифицировать принятый по умолчанию порядок прохождения событий – сначала вниз, а потом вверх по иерархии родитель–потомок.

В следующей главе речь пойдет о хранении данных в браузере с помощью формата `JSON` и клиентского хранилища. Мы также изучим навигацию между представлениями, в том числе вопрос о реагировании на нажатие встроенных кнопок браузера **Вперед** и **Назад** посредством манипулирования историей навигации.



Глава 11. Навигация и локальное хранение данных

В этой главе

- ❑ Браузерная навигация в одностраничном приложении.
- ❑ Запоминание пользовательских настроек в куках.
- ❑ Сериализация объектов в формате JSON.
- ❑ Локальное хранение данных в браузере.

Пользователи привыкли к встроенным в браузер средствам навигации – кнопкам **Вперед** и **Назад** и закладкам – и хотят работать с веб-приложениями точно так же, как всю жизнь работали с обычными сайтами. Кроме того, пользователи ожидают от приложения быстрого отклика, что достигается за счет сохранения данных в локальной файловой системе.

Современные браузерные технологии позволяют не разочаровывать пользователя. И сейчас мы добавим соответствующие возможности в приложение, которое начали разрабатывать в предыдущей главе.

Когда пользователь одностраничного приложения переходит от одного представления к другому, например от списка к редактированию, он ожидает, что нажатие кнопки **Назад** вернет его туда, откуда он ушел. Поэтому наше приложение должно каким-то образом уведомить браузер о смене представления. Мы покажем, как с помощью манипуляций историей можно реализовать навигацию с помощью кнопки **Назад** и поставить закладку на конкретное представление.

Рассмотрев механизм навигации с помощью стандартных средств браузера, мы обратимся к хранению пользовательских настроек в куках. Это даст возможность возвращаться к тому представлению, которое было на экране в момент закрытия браузера.

В предыдущей главе мы работали с подставными локальными данными, зашитыми в код приложения. Теперь мы вместо этого воспользуемся механизмом сохранения данных в браузере, основанным на использовании HTML5 Web Storage API.

Напомним, что сейчас в приложении DartExpense есть два пред-

ставления: список и редактирование. При нажатии пользователем какой-нибудь кнопки вызывается функция `navigate()`, работающая, как показано на рис. 11.1.

Если передать функции `getExpenseById()` значение `null` в качестве

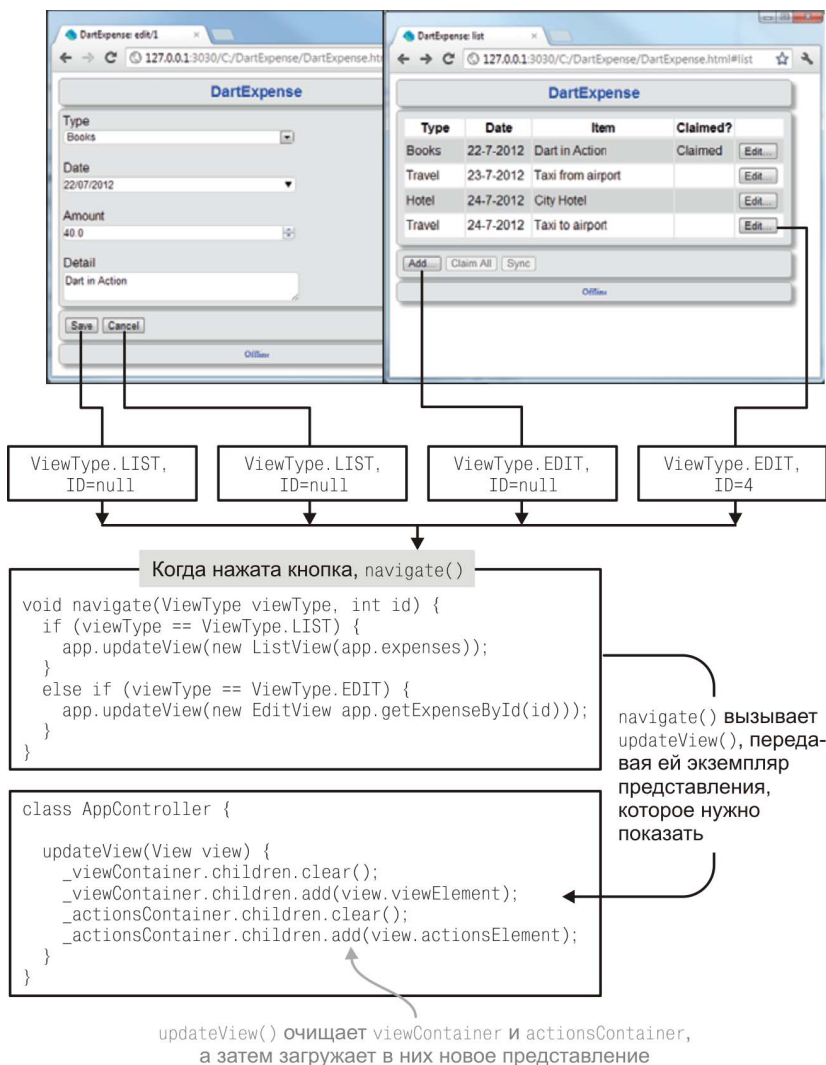


Рис. 11.1. Когда пользователь нажимает кнопку, текущее представление сменяется новым, переданным функции `navigate()`

параметра `id`, то вместо существующего объекта статьи расхода она вернет новый. Таким образом, одно и то же представление используется для двух целей: *добавления* и *редактирования*.

Функция `updateView()`, рассмотренная в предыдущей главе, очищает `<div>`'ы `_viewContainer` и `_actionContainer`, ссылки на которые хранятся в объекте `AppController`, и строит в них новое представление.

Проблема в том, что браузер обо всех этих манипуляциях ничего не знает. Приложение начинает работать при заходе на страницу `DartExpense.html`, и сколько бы раз пользователь ни переключался потом между представлениями, нажимая кнопки, с точки зрения браузера он остается на той же самой странице – в истории посещения страниц эти действия не отражаются. Это означает, что встроенные в браузер кнопки **Вперед** и **Назад** не помнят о перемещениях пользователя по приложению. Пора это исправить.

11.1. Интеграция навигации с браузером

Когда пользователь нажимает кнопку **Edit** в списковом представлении, браузер знает лишь, что выполняется какой-то код. Если мы хотим, чтобы браузер добавил новый URL-адрес в свою историю, то должны сообщить ему о том, что адрес изменился. К счастью, это нетрудно сделать с помощью появившейся в HTML5 функции `pushState()`, которая добавляет информацию в браузерную историю навигации.

Начальный URL-адрес приложения `DartExpense`, запущенного из редактора Dart в браузере Dartium, показан на рис. 11.2. Мы должны известить браузер о том, что этот адрес изменился, но при этом никуда не переходить.

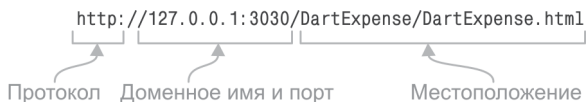


Рис. 11.2. Типичный URL-адрес, состоящий из протокола, доменного имени, порта и местоположения

11.1.1. Применение функции `pushState()` для добавления элементов в браузерную историю навигации

Когда пользователь переходит на другую страницу со сменой URL-адреса, браузер помещает новый адрес в список, который называется историей навигации. Функция `pushState()` позволяет имитировать эту операцию, никуда не переходя, например после нажатия кнопки **Edit**. Функция принимает три параметра: произвольные данные о состоянии приложения, заголовок (в настоящее время большинство браузеров его не используют) и новое местоположение, отражаемое в URL-адресе. Изменить протокол, доменное имя или порт мы не можем, но вот в часть «местоположение» будет записано переданное значение. Например, на рис. 11.3 показан вызов `pushState()`, при котором местоположение в URL-адресе изменяется на `"/edit/1"`.

`window.history.pushState("edit/1", "", "/edit/1");`

Данные
о состоянии

Заголовок
(не используется)

Новое
местоположение

Рис. 11.3. Вызов функции `pushState()`

При таком вызове значение первого параметра — `"edit/1"` — сохраняется в истории навигации как информация о состоянии. В приложении `DartExpense` эта строка интерпретируется как `viewName/idValue`, но в общем случае допустимы любые текстовые данные, описывающие текущее состояние приложения. `DartExpense` понимает, что `"edit/1"` означает, что мы находимся в представлении редактирования и редактируем запись о расходе с идентификатором 1. Когда пользователь нажмет кнопку **Save** или **Cancel**, мы снова вызовем `pushState()`, передав ей в качестве данных о состоянии строку `"list"` без идентификатора.

Второй параметр, заголовок, большинством браузеров пока не используется. Чтобы изменить текст в строке заголовка браузера, нужно установить свойство `document.title`, например:

```
document.title = "DartExpense: Editing expense with ID:1";
```

Последний параметр – новое местоположение – особенно интересен. Благодаря ему URL, отображаемый в адресной строке браузера, заменяется фиктивным, например вместо

```
http://127.0.0.1:3030/DartExpense/DartExpense.html
```

появляется

```
http://127.0.0.1:3030/edit/1
```

хотя реально мы никуда не переходили. Изменяется лишь местоположение, модифицировать таким образом доменное имя не получится. Однако этот фиктивный URL пока никуда не ведет, попытка перейти по адресу `http://127.0.0.1:3030/edit/1`, скорее всего, приведет к сообщению «404 page not found» (страница не найдена). Однако же параметр `location` подчиняется всем стандартным соглашениям о местоположении, принятым в браузере. В табл. 11.1 показаны значения `location` и соответствующие им URL-адреса.

Таблица 11.1. Типичные способы изменить местоположение в URL с помощью функции `pushState()`

Местоположение	Результирующий URL	Тип
<code>/edit/1</code>	<code>http://127.0.0.1:3030/edit/1</code>	Абсолютный путь
<code>edit/1</code>	<code>http://127.0.0.1:3030/DartExpense/DartExpense.html/edit/1</code>	Относительный путь
<code>?edit/1</code>	<code>http://127.0.0.1:3030/DartExpense/DartExpense.html?edit/1</code>	Строка запроса
<code>#edit/1</code>	<code>http://127.0.0.1:3030/DartExpense/DartExpense.html#edit/1</code>	Фрагмент пути

На этом этапе информация о местоположении нужна в основном для организации обратной связи с пользователем; приложению для выполнения навигации она ни к чему. В следующем разделе мы покажем, как воспользоваться данными, хранящимися в реальном URL, чтобы поставить закладку на конкретное местоположение в приложении.

В приложении DartExpense лучше не связывать вызовы `pushState()` с каждым нажатием кнопки, а поместить единственный вызов в уже существующую функцию `navigate()`, обслуживающую всю навигацию. В листинге ниже приведен модифицированный код функции `navigate()`, в котором тип представления и значение идентификатора

преобразуются в данные о состоянии и присоединяются к текущему URL в виде фрагмента пути.

Листинг 11.1. Модифицированная функция `navigate()`, интегрированная с браузерной историей навигации

```
void navigate(ViewType viewType, int id) {
  String state = id == null ?
  ↳      "${viewType.name}" :
  ↳      "${viewType.name}/${id.toString()}";

  window.history.pushState(state, "", "#$state");
  document.title = "DartExpense: $state";

  if (viewType == ViewType.LIST) {
    app.updateView(new ListView(app.expenses));
  } else if (viewType == ViewType.EDIT) {
    app.updateView(new EditView(app.getExpenseById(id)));
  }
}
```

Объединяем имя представления и идентификатор, если он не равен null
 Вызываем `pushState()`, чтобы сохранить состояние и фрагмент пути

← Обновляем строку заголовка в окне браузера

Презжий код

Если вызвать эту функцию с параметрами `navigate(ViewType.EDIT, 1)`, то она выполнит следующие действия:

1. Записать в переменную `state` значение `"edit/1"`.
2. Вызвать `pushState("edit/1", "", "#edit/1")`, что породит URL `http://127.0.0.1:3030/DartExpense/DartExpense.html#edit/1`.
3. Записать в заголовок документа строку `"DartExpense: edit/1"`.
4. Выполнить прежний код навигации, который покажет представление редактирования записи о расходе с `ID=1`.

В результате в адресной строке появится новый URL. В пользовательском интерфейсе будет показано представление редактирования, а старый URL-адрес сохранен в истории навигации (см. рис. 11.4).

Но возможность сохранить информацию о состоянии в истории — только полдела. Далее нужно заставить браузер уведомлять DartExpense о том, что пользователь нажал кнопку **Вперед** или **Назад**.

11.1.2. Подписка на событие `popState`

Итак, DartExpense использует `pushState()`, чтобы сообщить браузеру о состоявшемся переходе в другое состояние, но нужно еще, чтобы браузер уведомлял DartExpense о нажатии пользователем встроенной кнопки навигации. Когда пользователь нажимает кнопку

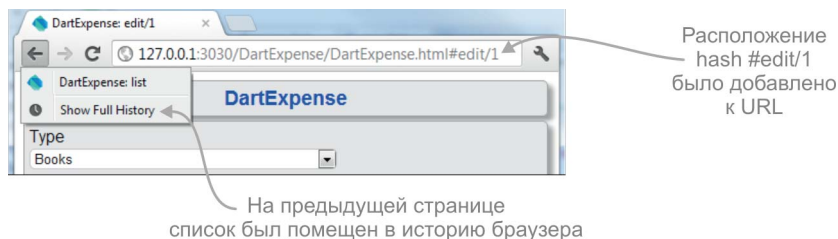


Рис. 11.4. Вызов `pushState()` в функции навигации помещает предыдущее местоположение в историю браузера

Вперед или **Назад** либо переходит на конкретную страницу в истории, браузер генерирует событие `popState`. Приложение `DartExpense` должно в ответ на него вызвать функцию, которая и выполнит заданную навигацию.

Когда пользователь нажимает кнопку **Назад**, браузер передает приложению данные о состоянии, сохраненные ранее функцией `pushState()`. Зная их, мы можем перевести приложение в нужное состояние. Например, если обработчику события `popState` была передана строка `"edit/1"`, то мы вызовем функцию `navigate()`, передав ей тип представления `ViewType.EDIT` и идентификатор `id=1`. Далее функция `navigate()` позаботится о том, чтобы показать пользовательский интерфейс для редактирования объекта `Expense` с идентификатором 1. Вызывая `navigate()` из обработчика события `popState`, ни в коем случае не следует допускать, чтобы она снова поместила данные о состоянии в браузерную историю (вызвав `pushState()`), — ведь браузер только что извлек их оттуда! Для этого мы заведем в `navigate()` необязательный параметр `isFromPopState`, который будем проверять перед вызовом `pushState()`:

```
navigate(ViewType viewType, int id, [bool isFromPopState=false])
```

На рис. 11.5 показана ожидаемая последовательность вызовов `pushState()` и обработчика `popState`.

Чтобы приложение `DartExpense` реагировало на событие `popState`, мы должны на этапе инициализации подключить обработчик этого события — `onPopState()`. В предыдущей главе было сказано, что за конструирование пользовательского интерфейса и настройку обработчиков событий отвечает функция `main()`; в листинге ниже приведена функция `onPopState()`, которая выделяет `ViewType` и `ID` из данных

1. Пользователь нажимает кнопку **Edit** для редактирования объекта Expense 1

```
navigate(ViewType.EDIT, 1) {
  pushState("edit/1", "", #edit/1")
  app.updateView(...);
}
```

Предыдущее состояние: "list"
Текущее состояние: "edit/1"

2. Пользователь нажимает кнопку **Save** и переходит в списковое представление

```
navigate(ViewType.LIST, null) {
  pushState("list", "", #list)
  app.updateView(...);
}
```

Предыдущее состояние: "list"
Предыдущее состояние: "edit/1"
Текущее состояние: "list"

3. Пользователь нажимает кнопку **Назад**, чтобы вернуться к предыдущему представлению edit/1

```
onPopState(PopStateEvent event) {
  var currentState = event.state;
  // ...snip convert state to
  // viewType and id
  navigate(viewType, id);
}
```

Свойство event.state
объекта типа PopStateEvent
содержит строку "edit/1"

Предыдущее состояние: "list"
Текущее состояние: "edit/1"

```
navigate(ViewType.EDIT, 1) {
  // do not pushState this time
  app.updateView(...);
}
```

Рис. 11.5. При вызове `pushState()` в историю навигации добавляется переданное состояние. Когда пользователь нажимает кнопку **Назад** или **Вперед**, браузер генерирует событие `popState`

о состоянии, а также модифицированная функция `main()`, которая присоединяет этот обработчик к событию `popState`.

Листинг 11.2. Функция `main()`, модифицированная для прослушивания событий `popState`

```
void main() {
  var uiContainer = query("#dartexpense");
  var dataSource = new MockData();

  _app = new AppController(uiContainer, dataSource);
  app.buildUI();
  app.loadFirstView();
  window.on.popState.add(onPopState); ← Добавляем функцию onPopState() в список
}                                       прослушивателей событий, хранящийся в браузере
```

```

void onPopState(PopStateEvent event) { ← Определяем onPopState()
    if (event.state != null) {
        List<String> stateData = event.state.split("/"); ← event.state содержит "edit/1" или "list"

        var viewName = stateData[0];
        var viewType = new ViewType(viewName); | Выделяем тип представления viewType

        var id = stateData.length == 2 ? int.parse (stateData[1]) : null; ← ID – это int или null

        navigate(viewType, id, isFromPopState:true); ← Вызываем navigate(), передавая true в параметре isFromPopState
    }
}

```

Теперь пользователь может перемещаться по приложению, используя предоставленные нами кнопки **Add**, **Edit** и **Save**, а также встроенные в браузер кнопки **Вперед** и **Назад** и историю навигации.

Памятка

- Функция `window.history.pushState(data,title,url)` добавляет записи в браузерную историю навигации.
- Обработчик события `window.on.popState.add( ... )` вызывается, когда пользователь нажимает встроенные в браузер кнопки **Вперед** и **Назад**.

Чтобы пользователю было приятнее работать, мы можем сделать еще один шаг: сохранять данные о состоянии не только в истории навигации с помощью функции `pushState()`, но и в куке браузера. Тогда при возврате в приложение после перерыва пользователь увидит то представление, на котором прервал работу.

11.2. Использование куков браузера для повышения удобства работы

Кук браузера – это небольшой текстовый файл, относящийся к конкретному сайту и сохраняемый браузером на компьютере пользователя. Приложение может записать в один кук примерно 4096 байтов данных. Для одного сайта можно хранить от 80 до 140 куков. Впрочем, точные значения зависят от браузера, так что рекомендуем не слишком увлекаться – чем меньше, тем лучше. Данные в кук представляют собой пару ключ–значение, а также дату, после которой кук перестает быть действительным. Кук ассоциирован с конкретным доменом, поэтому куки, созданные вашим приложением, другим сайтам недоступны.

Предупреждение. Поскольку куки хранятся в открытом виде, не следует записывать в них конфиденциальные данные, например пароли. Пользователь может вообще отключить куки, поэтому использовать их следует только как вспомогательное средство, не полагаясь на то, что они обязательно существуют.

Чтобы войти в приложение DartExpense, пользователь должен загрузить файл DartExpense.html. Но после этого мы можем показать не списковое представление, а то, которое находилось на экране, когда пользователь в последний раз работал с приложением.

Примечание. Куки связаны с конкретным браузером. Так, куки, сохраненные приложением DartExpense в браузере Chrome, будут недоступны при запуске того же приложения на той же машине, но в браузере Firefox.

Как и в случае `pushState()` и `popState`, работа с куками имеет две стороны. Во-первых, нужно сохранять в кукe информацию о состоянии при переходах, а во-вторых, извлекать ее на этапе инициализации приложения (если кук существует).

11.2.1. Сохранение данных в кукe

Идеальным местом для сохранения данных о текущем представлении в кукe снова является функция `navigate()`. Для создания кука нужно записать пару ключ–значение в свойство `document.cookie`, например:

```
document.cookie = "stateData=edit/1";
```

Свойство `cookie` несколько отличается от других свойств. При его установке создается или изменяется хранимый набор данных. Если кук `stateData` уже существует, то в результате показанного выше присваивания свойству `cookie` он перезаписывается. Если кук не существует, то он создается. Если задать другой ключ кука, как в случае `expenseType` ниже, то браузер создаст еще один кук. Например, если два раза присвоить значение свойству `document.cookie`:

```
document.cookie = "stateData=edit/1";  
document.cookie = "expenseType=books";
```

то будет сохранено два кука. Предложение `print(document.cookie)` вернет строку `"stateData=edit/1; expenseType=books"`.

Показанный выше способ установки куков не даст желаемого эффекта, потому что такие куки существуют только на протяжении сеанса. То же самое приложение, открытое в другой вкладке браузера, их увидит, но после закрытия окна браузера куки уничтожаются. Если задан срок хранения, то кук считается сеансовым. Чтобы кук пережил закрытие сеанса, необходимо задать для него свойство `expires` или `max-age`.

Дата окончания срока хранения (`expires`) должна быть задана в определенном формате. В JavaScript это формат "Wdy, DD Mon YYYY HH:MM:SS GMT", в котором возвращает дату функция `toGMTString()`, например:

```
document.cookie="stateData=edit/1;expires=Fri, 3 Aug 2012 20:47:11 GMT";
```

К сожалению, в Dart нет эквивалентной функции, которая выводит дату именно в таком формате. Поэтому ее нужно написать самостоятельно или воспользоваться гораздо более простым свойством `max-age`.

В свойстве `max-age` задается количество секунд до окончания срока хранения кука. Например, чтобы кук хранился неделю, нужно создать его следующим образом:

```
document.cookie="stateData=edit/1;max-age=${60*60*24*7}";
```

В браузерах Chrome и Dartium куки можно просмотреть в разделе **Cookies** на вкладке **Elements** консоли инструментов разработчика, как показано на рис. 11.6.

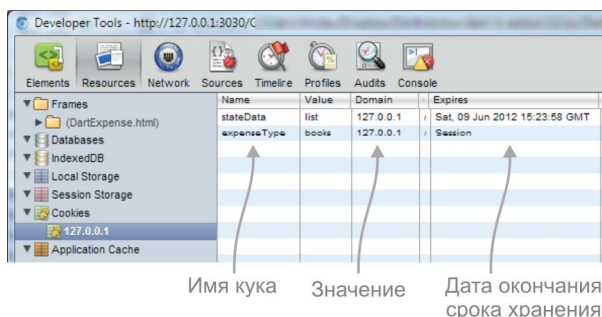


Рис. 11.6. Куки, установленные приложением, можно увидеть в разделе **Cookies** консоли инструментов разработчика в браузерах Chrome и Dartium

Добавьте в функцию `navigate()` строку

```
document.cookie="stateData=$state;max-age=${60*60*24*7*4}";
```

рядом с местом, где изменяется свойство `document.title`. При любой операции навигации предыдущее значение перезаписывается, переменная `state` хранится в кукке четыре недели.

11.2.2. Чтение данных из кука

Научившись сохранять данные в кукке, мы должны модифицировать приложение, так чтобы оно доставало эти данные, извлекало из них имя представления и идентификатор и передавало их функции `navigate()` – точно так же, как это делается в функции `onPopState()`. К сожалению, свойство `document.cookie` возвращает строку со списком значений всех кукков через запятую, например:

```
"stateData=edit/1; expenseType=books"
```

Нам необходима вспомогательная функция, которая извлечет из этой строки значение заданного ключа, если таковой существует. В листинге 11.3 показано, как с помощью функции `string.split()` разбить возвращенную строку на список строк вида `ключ=значение`, считая точку с запятой символом-разделителем. Затем каждая строка `ключ=значение` еще раз разбивается на части – на этот раз по знаку равенства, и анализируется ключ. Если нужного ключа не существует – потому что кук еще не был установлен или срок его хранения уже истек, – то функция возвращает пустую строку.

Листинг 11.3. Вспомогательная функция для получения значения кука с указанным ключом

```
String getValueFromCookie(String key) {                                     Для каждого кука
  for (String cookieKV in document.cookie.split(";")) { ← в свойстве document.cookie...
    List cookie = cookieKV.split("="); ← ... разбить на пары ключ–значение
    if (cookie.length > 1) {
      var cookieKey = cookie[0].trim(); ← Если значение есть...
      if (cookieKey == key && cookie.length > 1) {
        return cookie[1]; ← ...вернуть его, если ключ совпадает с заданным
      }
    }
  }
  return ""; ← Вернуть пустую строку, а не значение null, подразумеваемое по умолчанию
}
```

Теперь, имея функцию, которая получает значение нужного кука, мы можем внести изменения в процедуру инициализации приложения. Функция `main()` вызывает метод `loadFirstView()` объекта `AppController`, поэтому именно в него было бы естественно поместить проверку существования кука и настройку начального представления. Для выделения `ViewType` и `ID` из кука я просто скопирую код из функции `onPopState()`, а полноценный рефакторинг оставлю читателю в качестве упражнения. В листинге ниже приведен модифицированный код метода `loadFirstView()`.

Листинг 11.4. Изменение начального представления в случае, когда имеется кук `stateData`

```
class AppController {
    // ...остальные методы и свойства опущены...

    loadFirstView() {
        var viewType = ViewType.LIST; | По умолчанию устанавливается
        var id = null;                | списковое представление          Пытаемся получить
                                     |                                   значение кука
        var stateCookieValue = getValueFromCookie("stateData"); ← stateData

        if (stateCookieValue != null && stateCookieValue.length > 0) { ←
            List<String> stateData = stateCookieValue.split("/");
            var viewName = stateData[0]; | Если значение успешно получено, то раз-
            var viewType = new ViewType(viewName); | бираем его так же, как в функции onPopState()
            var id = stateData.length == 2 ? int.parse(stateData[1]) : null;
        }
        navigate(viewType, id); ← Передаем функции navigate() значения, подразумеваемые
    }                               по умолчанию или извлеченные из кука
}

```

Теперь приложение пользуется куками (формой локального хранилища) для сохранения информации между сеансами. А для перехода между различными представлениями приложения к услугам пользователя встроенная в браузер история навигации.

Памятка

- Для сохранения небольших объемов данных между сеансами работы с браузером можно использовать куки.
 - Куки небезопасны и могут быть отключены пользователем, поэтому применять их следует только как вспомогательное средство.
-

В следующем разделе мы научимся сохранять больше данных с помощью HTML5 Web Storage API, реализованного в современных браузерах. И наконец-то перейдем от подставных данных к реальным.

11.3. Локальное хранение данных с помощью Web Storage API

Современные браузеры предлагают различные механизмы локального хранения данных с большим или меньшим уровнем поддержки. Простейший вариант, поддерживаемый максимальным числом браузеров, – Web Storage API, позволяющий хранить список пар ключ–значение, как в куках, но без ограничения срока хранения и в куда большем объеме – 5 Мб на домен. Далее идет IndexedDB – индексированное объектное хранилище, концептуально похожее на серверные NoSQL-базы типа MongoDB и CouchDB. Эта система позволяет искать подходящие записи по индексу, что дает гораздо более высокую производительность, чем Web Storage API, в случае когда требуется найти некоторое подмножество записей. Наконец, есть еще Web SQL Database – встроенная база данных на основе SQL. Но она поддержана не очень широко, отчасти из-за решения Mozilla отказаться от ее поддержки в пользу развития API IndexedDB. Сопровождение спецификации Web SQL остановлено, поэтому браузеры, которые еще поддерживают ее, в скором времени могут это прекратить. В табл. 11.2 показано состояние поддержки вышеупомянутых технологий летом 2012 года.

Таблица 11.2. Поддержка технологий локального хранения браузерами

Браузер и версия	Web Storage	IndexedDB	Web SQL DB
Chrome 17+	Поддерживается	Поддерживается	Поддерживается
Android Browser 2.1+	Поддерживается	Не поддерживается	Поддерживается
Firefox 3.6	Поддерживается	Поддерживается	Поддерживается
Firefox 8+	Поддерживается	Поддерживается	Поддерживается
Internet Explorer 8	Поддерживается	Поддерживается	Поддерживается
Internet Explorer 8	Поддерживается	Не поддерживается	Не поддерживается
Internet Explorer 8	Поддерживается	Поддерживается	
Safari 5+	Поддерживается	Не поддерживается ¹	Поддерживается
iOS Safari 3.2+	Поддерживается	Не поддерживается	Поддерживается
Opera 11.6+	Поддерживается	Не поддерживается ¹	Поддерживается
Opera Mobile 11+	Поддерживается	Не поддерживается	Поддерживается

¹ Планы поддержки IndexedDB в браузерах Safari и Opera в настоящее время неизвестны.

Как видите, все современные браузеры поддерживают Web Storage, но IndexedDB не поддерживается в Safari и Opera, а также в более старых версиях IE и Firefox. Что касается поддержки Web SQL, то она была добавлена в браузеры Chrome и Safari, пока спецификация еще активно разрабатывалась, но IE и Firefox не поддерживали ее никогда.

Примечание. Локальное хранилище, как и куки, связано с конкретным браузером. Это означает, что данные DartExpense, находящиеся в локальном хранилище Chrome, не будут доступны этому же приложению, запущенному в браузере Firefox на той же самой машине.

В этом разделе мы научим DartExpense пользоваться самым распространенным хранилищем на основе Web Storage API. В подобном случае ключи и значения должны быть строками, то есть сохранять расходы непосредственно не получится. Сначала их необходимо преобразовать в строковое представление, например в формат JSON (JavaScript Object Notation). В библиотеке Dart для работы с JSON есть две функции: `JSON.stringify()` и `JSON.parse()`. Первая преобразует словари и списки в строки, вторая — наоборот. Нам необходимо каким-то образом представить объекты типа `Expense` и `ExpenseType` в виде словаря.

11.3.1. Преобразование объектов Dart в формат JSON

В JavaScript объекты представляют собой словари, то есть множества пар ключ–значение, а не экземпляры реальных классов, как в Dart. Формат JSON, распространенный в веб повсеместно, ориентирован на представление структур данных JavaScript и напрямую к модели классов, принятой в Dart, неприменим. Однако его можно преобразовать в объект класса `Map`, содержащий список пар ключ–значение. Чтобы представить объекты классов `Expense` и `ExpenseType` в формате JSON, мы должны создать словарь из списка свойств объекта. Затем этот словарь можно передать функции `JSON.stringify()`, которая преобразует его в строку в формате JSON.

На рис. 11.7 показано, как словарь, описывающий статью расхода и содержащий среди прочего вложенный словарь `expenseType`, преобразуется в JSON-строку функцией `JSON.stringify()`. Парная функция `JSON.parse(string)` строит из JSON-строки словарь.

Словарь в Dart...

```
var exMap = new Map();
exMap["amount"] = 40;

exMap["expenseType"] = new Map();
exMap["expenseType"]["name"] = "Travel";
exMap["expenseType"]["code"] = "TRV";

exMap["date"] = new Date.now().toString();
exMap["detail"] = "Taxi from Airport";
exMap["id"] = 1;
exMap["isClaimed"] = false;
```

... преобразуется в JSON-строку

```
{
  "amount": 40,
  "expenseType": {
    "name": "Travel",
    "code": "TRV"
  },
  "date": "2012-06-25 00:00:00.000",
  "detail": "Taxi from Airport",
  "id": 1,
  "isClaimed": false
}
```

Преобразовать словарь в JSON-строку

```
var jsonString = JSON.stringify(exMap);
```

Преобразовать JSON-строку в словарь

```
var exMap = JSON.parse(jsonString);
```

Рис. 11.7. Преобразование словаря
в JSON-строку с помощью `JSON.stringify()`

Поскольку в библиотеках Dart нет готовых функций, умеющих преобразовывать произвольные объекты в формат JSON и обратно, эту функциональность придется добавить в классы `Expense` и `ExpenseType` вручную.

Версия Milestone 1

В версию языка Milestone 1 планируется добавить поддержку отражения, в результате чего станет возможен динамический анализ свойств объекта во время выполнения. Тогда разработчики библиотек Dart смогут среди прочего реализовать преобразование Dart-объектов в формат JSON и наоборот. Но иногда требуется преобразовать лишь некоторые (а не все) поля, поэтому описываемая ниже техника все равно останется полезной.

Есть два способа преобразовать Dart-объект в словарь, пригодный для функции `JSON.stringify()`. Самое простое – включить в класс функцию `toMap()`, возвращающую словарь. Чуть сложнее реализовать в классе интерфейс `Map`, зато этот подход более гибкий, а код получается более понятным.

Написание специальной функции `toMap()`

Этот способ позволяет писать такой код:

```
var jsonString = JSON.stringify(expense.toMap());
```

в большинстве случаев вполне приемлемый. Ниже показано, как реализовать метод `toMap()` в классах `Expense` и `ExpenseType`.

Листинг 11.5. Реализация метода `toMap()` в классах `Expense` и `ExpenseType`

```
class Expense {
    int _id;
    int get id => _id;
    ExpenseType type;
    Date date;
    num amount = 0;
    String detail;
    bool isClaimed = false;

    Map toMap() { ← Функция Expense.toMap()
        final map = new Map<String, Object>();
        map["id"] = _id;
        if (date != null) {
            map["date"] = date.toString();
        }
        map["amount"] = amount;
        map["detail"] = detail;
        map["isClaimed"] = isClaimed;
        if (type != null) {
            map["expenseType"] = type.toMap();
        }
        return map;
    }
}

class ExpenseType {
    final String name;
    final String code;

    const ExpenseType(this.name, this.code);

    Map toMap() {
        final map = new Map<String, Object>();
        map["name"] = name;
        map["code"] = code;
        return map;
    }

    bool operator ==(other) {
        if (other == null) return false;
        return this.name == other.name && this.code == other.code;
    }
}
```

Строковое, числовое
и булевское свойства
записываются в словарь

Вложенный словарь,
представляющий свойство
`expenseType`

Функция `ExpenseType.toMap()`

При использовании этой техники всякий объект верхнего уровня возвращает свое представление в виде строки или словаря. Библиотечные функции умеют преобразовывать в формат JSON только типы `String`, `num`, `bool`, `List`, `Map` и `null`. А значит, все остальные типы, например `Date`, нужно преобразовывать в строки (с помощью метода `toString()`) или в словари, как мы поступили в функции `toMap()` из класса `ExpenseType`.

Реализация интерфейса `Map`

Второй способ заключается в том, чтобы реализовать в классе интерфейс `Map`, который позволит передавать объект непосредственно функции `JSON.stringify()`. Правда, придется написать больше стереотипного кода (для реализации интерфейса), хотя устроен он примерно так же, как функция `toMap()`. Зато в результате получается более элегантный и читабельный код, например:

```
var jsonString = JSON.stringify(expense);
```

У реализации интерфейса `Map` есть и еще одно преимущество: функции `stringify()` можно передавать не только одиночный объект `Expense`, но и список таких объектов. При этом функция будет считать каждый элемент списка словарем и соответственно преобразовывать его. С методом `toMap()` такой фокус не пройдет, потому что `stringify()` не знает, что нужно вызвать его для каждого элемента списка.

Ниже перечислены важнейшие свойства и методы интерфейса `Map`, которые необходимо реализовать:

- ❑ `keys` – возвращает список свойств класса `Expense` в виде списка строк;
- ❑ перегруженный оператор `[]` – возвращает значение свойства, имя которого совпадает с заданным ключом (преобразованное к типу `String`, `num`, `bool`, `List`, `Map` или `null`);
- ❑ `forEach()` – принимает функцию в качестве параметра. Эта функция вызывается для каждого ключа, причем ей передаются сам ключ и ассоциированное с ним значение.

Запрос ключа, отсутствующего в коллекции `keys`, должен возвращать `null`. Все остальные методы возбуждают исключение `UnimplementedError`, поскольку в нашем случае они не требуются. В листинге ниже показана реализация интерфейса `Map` в классе `Expense`.

Листинг 11.6. Реализация интерфейса Map для поддержки функции JSON.stringify()

```

class Expense implements Map {  ← Реализация интерфейса Map
    //...остальные свойства и методы опущены...

    Collection get keys {
        return ["id", "amount", "expenseType", "date", "detail", "isClaimed"];
    }

    operator [] (key) {  ← Возвращается список имен ключей фиксированного размера
        if (key == "id") {
            return this.id;
        } else if (key == "amount") {
            return this.amount;
        } else if (key == "expenseType") {
            return this.type.toMap();
        } else if (key == "date") {
            return date == null ? null : date.toString();
        } else if (key == "detail") {
            return this.detail;
        } else if (key == "isClaimed") {
            return this.isClaimed;
        } else {
            return null;  ← ...или null
        }
    }

    forEach(funcParam(key,value)) {
        for (var k in keys) {
            funcParam (k, this[k]);
        }
    }

    bool get isEmpty(){ throw new UnimplementedError(); }
    bool containsValue(value) { throw new UnimplementedError(); }
    bool containsKey(value) { throw new UnimplementedError(); }
    Collection get keys { throw new UnimplementedError(); }
    Collection get values { throw new UnimplementedError(); }
    int get length { throw new UnimplementedError(); }
    void clear() { throw new UnimplementedError(); }
    void remove(key) { throw new UnimplementedError(); }
    putIfAbsent(key, ifAbsent) { throw new UnimplementedError(); }
    void operator []=(key, value) { throw new UnimplementedError(); }
}

```

Возвращается значение для заданного ключа...

Для каждого ключа в коллекции keys функции funcParam() передаются сам ключ и его значение

Нереализованные методы интерфейса Map

Для реализации интерфейса `Map` приходится писать много стеореотипного кода, но использовать такой класс проще, потому что он позволяет прозрачно преобразовывать одиночные объекты или списки объектов типа `Expense` в JSON-строку с помощью функции `JSON.stringify()`.

11.3.2. Преобразование JSON-строк в Dart-объекты

Разработав механизм для преобразования класса в строку, мы должны придумать, как выполнить обратное преобразование с использованием функции `JSON.parse()`. К сожалению, в имеющихся на данный момент библиотеках готового решения нет. И хотя класс `Expense` реализует интерфейс `Map`, функция `JSON.parse()` возвращает словарь, а не объект класса `Expense`, потому что об этом классе она ничего не знает.

Стандартный способ решить эту задачу в Dart заключается в том, чтобы включить в класс именованный конструктор `fromMap()`, который инициализирует свойства объекта. В таком случае объект можно будет создать следующим образом:

```
var jsonString = ...преобразование статьи расхода в JSON-строку опущено...
var map = JSON.parse(jsonString);
var expense = new Expense.fromMap(map);
```

Конструктор `fromMap()`, показанный в листинге 11.7, читает все значения из словаря и инициализирует ими свойства объекта класса `Expense`, в том числе и свойство типа `ExpenseType`, которое конструируется путем чтения его свойств `name` и `code` из внутреннего словаря `expenseType`. Помимо именованного конструктора `fromMap()`, в классе остается также конструктор по умолчанию, поэтому создавать пустые объекты можно, как и раньше.

Листинг 11.7. Конструктор `fromMap()`: инициализирует объект на основе словаря его свойств

```
class Expense implements Map {
  //... остальные методы и реализация интерфейса Map опущены...
  int _id;
  int get id => _id;
  ExpenseType type;
  Date date;
```



```

num amount = 0;
String detail;
bool isClaimed = false;

Expense() {
  _id = _getNextId();
};

```

Существующий
конструктор
по умолчанию

```

Expense.fromMap(Map map) {
  _id = map["id"];
  date = new Date.fromString(map["date"]);
  amount = map["amount"];
  detail = map["detail"];
  isClaimed = map["isClaimed"];
  if (map.containsKey("expenseType")) {
    var expenseTypeName = map["expenseType"]["name"];
    var expenseTypeCode = map["expenseType"]["code"];
    type = new ExpenseType(expenseTypeName, expenseTypeCode);
  }
}

```

← Именованный конструктор fromMap

Инициализируем
свойства значениями
из словаря

Читаем значения
expenseType из внутреннего
словаря

Записываем в свойство type
новый объект типа ExpenseType

Примечание. Недавно в библиотеку `dart:json` была добавлена новая возможность, помогающая преобразовывать объекты в формат JSON: теперь библиотека пытается вызвать метод `toJson()`, если преобразуемый тип отличается от стандартных. Вы можете реализовать этот метод в своем классе, так чтобы он возвращал строковое представление объекта.

Итак, у нас есть механизм преобразование объектов, представляющих расходы, в формат JSON и обратно. Следующий шаг – сохранить их в браузере, воспользовавшись Web Storage API.

11.3.3. Сохранение данных в браузере

Программный интерфейс Web Storage API имеет две реализации с одинаковым синтаксисом: сеансовое хранилище и локальное хранилище. В примерах ниже мы будем использовать локальное хранилище, которое позволяет сохранять данные между сеансами. Во втором – сеансовом – хранилище данные хранятся только на протяжении одного сеанса (до момента закрытия браузера), обратиться к нему можно с помощью свойства `window.sessionStorage`.

Примечание. Хотя синтаксис доступа к обоим хранилищам одинаков, между ними есть важное различие. Объем локального хранилища не может превышать 5 Мб на один домен, тогда как объем сеансового хранилища ограничен только системными ресурсами.

Для доступа к локальному хранилищу служит свойство `window.localStorage`. Тип этого свойства реализует интерфейс `Map`, что позволяет читать и записывать данные по индексу:

```
window.localStorage["expense"] = JSON.stringify(expense);  
var jsonString = window.localStorage["expense"];  
expense = new Expense.fromJson(jsonString);
```

Как и куки, пары ключ–значение в локальном хранилище связаны с конкретным доменом, поэтому данные, созданные приложением `DartExpense`, хранятся отдельно от данных других сайтов. Как и куки, данные в локальном хранилище представлены в открытом виде, поэтому помещать туда конфиденциальные данные, например пароли, не стоит. На рис. 11.8 показано, как выглядит локальное хранилище на панели инструментов разработчика в браузере Chrome или Dartium.

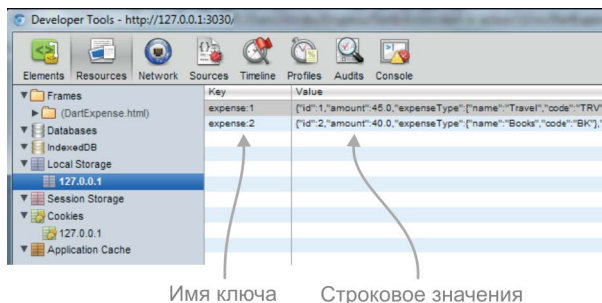


Рис. 11.8. На панели инструментов разработчика можно просмотреть локальное хранилище ключей и значений

Предупреждение. Все данные, находящиеся в локальном хранилище, не защищены, их может просмотреть любой человек, получивший доступ к браузеру. Приложение должно либо зашифровать их, либо – что еще лучше – вообще не хранить в браузере секретные данные.

Вставка данных в локальное хранилище

Поскольку приложению DartExpense требуется хранить несколько объектов Expense, оно будет строить ключ в виде строки `expense:${expense.id}`, то есть ключи будут называться `expense:1`, `expense:2` и т. д. Это дает возможность получать и модифицировать каждую запись о расходе по отдельности. Альтернативно можно было бы хранить весь преобразованный в формат JSON список расходов в одном свойстве:

```
List<Expense> expensesList = ...построение списка опущено...
window.localStorage["expenses"] = JSON.stringify(expensesList);
```

Но у такого способа есть недостаток: даже если нужно изменить всего одну запись о расходе, модифицировать придется весь список.

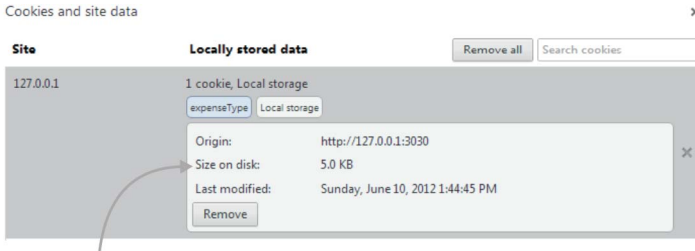
Когда некоторому ключу, например `expense:1`, присваивается JSON-строка, в локальном хранилище либо создается новая запись, либо перезаписывается существующая. Иногда желательно знать, что именно произошло, например чтобы вывести одно из двух сообщений: «Запись о расходе создана» или «Запись о расходе изменена». В интерфейсе Map имеется метод `containsKey()`, который позволяет произвести такую проверку перед вставкой значения, например:

```
var isNewValue = window.localStorage.containsKey("expense${expense.id}");
window.localStorage["expense${expense.id}"] = JSON.stringify(expense);
```

В локальном хранилище для одного домена может храниться не более 5 Мб данных (в сеансовом хранилище место не ограничено). Чтобы узнать, сколько места реально занято, нужно зайти в раздел **Cookies and Site Data** (Файлы cookie и данные сайта) на странице дополнительных настроек Chrome, как показано на рис. 11.9.

Пользователи, а также установленные подключаемые модули и расширения могут отключать хранилище Web Storage, поэтому приложение должно обрабатывать такую ситуацию. Попытка записать данные в отключенное локальное хранилище приведет к исключению `DOMException`. Лучше всего помещать все операции записи в локальное хранилище внутри блока `try/catch`:

```
try {
    window.localStorage["expense:${expense.id}"] = JSON.stringify(expense);
}
on DOMException catch (ex) {
    window.alert("Локальное хранилище недоступно");
}
```



Для любого сайта можно узнать размер места, занятого куками или данными в локальном хранилище. Для этого нужно перейти в раздел **Cookies and Site Data** (Файлы cookie и данные сайта) на странице дополнительных настроек Chrome

Рис. 11.9. Сколько данных хранится для конкретного сайта

Изменим функцию `addOrUpdate()`, так чтобы она помещала данные в локальное хранилище.

Листинг 11.8. Добавление или обновление записи о расходе в локальном хранилище

```
bool addOrUpdate(Expense expense) {
    var localStorageKey = "expense:${expense.id}";
    bool isNew = !window.localStorage.containsKey(localStorageKey);

    try {
        window.localStorage[localStorageKey] = JSON.stringify(expense);
    }
    on DOMException catch (ex) {
        window.alert("Локальное хранилище отключено");
    }

    return isNew;
}
```

Чтение данных из локального хранилища

Прочитать данные из локального хранилища еще проще. Поскольку класс свойства `localStorage` реализует интерфейс `Map`, для чтения достаточно передать аксессуару имя ключа:

```
var jsonString = window.localStorage["expense:1"];
```

Приложению DartExpense нужно загрузить список всех расходов. Для этого прочитаем из `localStorage` все значения, ключи которых начинаются строкой `expense:.` Это просто – достаточно воспользоваться свойством `keys`, как показано в листинге ниже. Здесь мы извлекаем JSON-представление расхода и преобразуем его в объект `Expense`, вызывая именованный конструктор `fromMap()`.

Листинг 11.9. Чтение списка расходов из локального хранилища

```
List<Expense> loadExpenses() {
    var expensesList = new List<Expense>();    ← Создаем пустой список

    for (var key in window.localStorage.keys) {    ← Перебираем все ключи
                                                в локальном хранилище
        if (key.startsWith("expense:")) {    ← Если имя ключа начинается строкой expense:...
            String value = window.localStorage[key];    ..., то преобразуем строку
            var map = JSON.parse(value);                в словарь, а затем
            var expense = new Expense.fromMap(map);      в объект Expense
            expensesList.add(expense);    ← Добавляем запись о расходе в список
        }
    }

    return expenses;    ← Возвращаем список
}
```

А теперь все вместе: замена подставных данных локальным хранилищем

Научившись сохранять данные в локальном хранилище и читать их оттуда, мы можем заменить класс `MockData` в функции `main()`. Этот класс реализует интерфейс `DataAccess`:

```
interface DataAccess {
    List<Expense> get expenses;
    bool addOrUpdate(Expense expense);
}
```

Нам нужен новый класс `LocalStorageData`, в котором будут реализованы обе функции: возврат списка расходов и обновление одной записи о расходе, но только применительно к данным в локальном хранилище. Решив эту задачу, мы сможем изменить функцию `main()`, как показано ниже.

Листинг 11.10. Чтение списка расходов из локального хранилища

```
void main() {  
  var uiContainer = query("#dartexpense");  
  var dataSource = new MockData();  
  var dataSource = new LocalStorageData();  
  _app = new AppController(uiContainer, dataSource);  
  app.buildUI();  
  app.loadFirstView();  
  
  window.on.popState.add(onPopState);  
}
```

Заменяем тип `dataSource`
← классом `LocalStorageData`
← Передаем `dataSource`
объекту `AppController`

Теперь приложение `DartExpense` превратилось в настоящее автономное клиентское приложение, которое между сеансами сохраняет данные в локальном хранилище.

Памятка

- Из всех технологий локального хранения `Web Storage API` поддерживана наиболее широко и проще всего в использовании.
- Хранилище `window.localStorage` позволяет хранить данные между сеансами, но не более 5 Мб на один домен.
- Хранилище `window.sessionStorage` позволяет хранить сколько угодно данных (объем ограничен только системными ресурсами), но уничтожается по завершении сеанса.

11.4. Резюме

В этой главе мы дополнили приложение `DartExpense`, сделав работу пользователя более удобной. Подключив браузерную историю навигации с помощью функции `pushState()` и события `popState`, мы дали пользователю возможность переходить от одного представления к другому, применяя стандартные средства браузера. Мы также воспользовались куками, чтобы после перезагрузки браузера показывать то представление, которое пользователь видел в последний раз.

Мы научились преобразовывать Dart-объекты в JSON-строки и наоборот с помощью функций `JSON.stringify()` и `JSON.parse()`. Эти функции умеют работать только с типами `Map`, `List`, `String`, `bool`, `num` и `null`; механизм преобразования собственного класса, например `Expense`, в словарь, нужно написать самостоятельно. Мы сделали это, реализовав в классе интерфейс `Map`. Для обратного преобразования

из словаря JSON мы написали именованный конструктор `fromMap()`, который инициализирует свойства объекта.

Наконец, мы узнали, как сохранять данные в формате JSON в браузерном локальном хранилище, если пользоваться свойством `localStorage`, которое дает доступ к хранилищу ключей и значений, поддерживаемому всеми современными браузерами. Это позволило нам заменить подставные данные в `DartExpense` реальными и сохранять их между сеансами браузера.

В четвертой части книги мы рассмотрим применение Dart на сервере и включим в приложение `DartExpense` средства синхронизации данных с сервером. Но сначала, в следующей главе, обсудим взаимодействие с функциями JavaScript и вопрос о том, как Dart и JavaScript, действуя совместно, могут запрашивать JSON-данные с общедоступных серверов.



Глава 12. Взаимодействие с другими системами и языками

В этой главе

- ☐ Взаимодействие с JavaScript.
- ☐ Получение данных с внешних серверов с помощью технологии JSONP.
- ☐ Эксплуатация приложения без сервера.
- ☐ Создание допускающего установку пакета приложения для Chrome.

Хотя приложение и можно написать целиком на Dart, история на этом не заканчивается. Dart-приложения обитают в более широкой экосистеме, охватывающей браузерные приложения, существующие каркасы и библиотеки, написанные на JavaScript, и API доступа к сторонним серверам. И везде есть проблемы, связанные с доступностью серверов, безопасностью браузеров и взаимодействием между различными виртуальными машинами.

В этой главе мы научим приложение DartExpense взаимодействовать с JavaScript путем обмена сообщениями между VM Dart и JavaScript. Это даст нам возможность вызывать Dart-код из JavaScript-кода и наоборот.

Благодаря данной возможности мы сможем использовать основанный на JavaScript механизм взаимодействия со сторонними серверами, например публичными API Google, применяя технологию обратных вызовов JSONP (JSON с набивкой), которая позволяет получать данные от стороннего сервера, обходя ограничения безопасности, действующие для AJAX-вызовов.

Затем мы расширим возможности DartExpense работать автономно, воспользовавшись HTML5-технологией AppCache, которая позволяет браузеру запускать приложение, даже когда сервер недоступен — идеальное решение для мобильных приложений, когда гарантировать наличие сети не всегда возможно.

Наконец, мы покажем, как сформировать пакет приложения для Chrome, который можно запустить полностью автономно или установить в Google Chrome из магазина Chrome Web Store. Допускающие установку приложения для Chrome – это способ представить ваш продукт широкой аудитории с помощью Web Store.

Мы начнем с добавления в DartExpense круговой диаграммы типов расходов, для чего воспользуемся сторонней JavaScript-библиотекой рисования графиков. Это первая внешняя система, с которой мы интегрируем Dart (см. рис. 12.1).

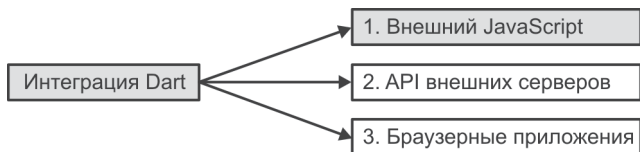


Рис. 12.1. Интеграция Dart с JavaScript

12.1. Взаимодействие с JavaScript

На JavaScript написаны тысячи библиотек, и многие разработчики хотели бы воспользоваться некоторыми из них в своих приложениях. По мере роста популярности языка Dart будут появляться написанные на нем версии известных библиотек, но до тех пор необходимо организовывать совместную работу виртуальных машин – вызов JavaScript из Dart и наоборот. Это относится и к случаю, когда Dart транслируется на JavaScript, – хотя при этом существует только VM JavaScript, конвертированный код не так-то легко использовать совместно с другими написанными на JavaScript программами из-за различных видов оптимизации.

Примечание. Имейте в виду, что вызов JavaScript из Dart отличается от принципа работы применяемого в Google Web Toolkit (GWT) интерфейса JavaScript Native Interface (JSNI), который позволяет внедрять JavaScript-код прямо в код, написанный на Java. В Dart имеется четкая граница, на которой заканчивается Dart-код и начинается JavaScript-код. Но вполне вероятно, что эта граница будет постепенно стираться, по мере того как команда Dart разработает прокси-объекты, улучшающие интероперабельность Dart и JavaScript.

Мы добавим в приложение DartExpense третье представление – диаграмму. В нем будет пустой `<div>`, который мы заполним с по-

мощью библиотеки Google Chart, написанной на JavaScript. Мы будем передавать из Dart в JavaScript данные о расходах, на основе которых построим круговую диаграмму по типам расходов, показанную на рис. 12.2.

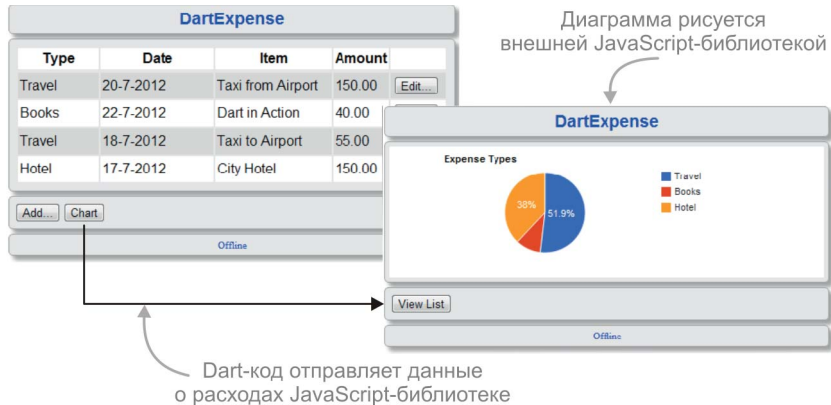


Рис. 12.2. Добавление кода генерации круговой диаграммы, написанного на JavaScript, в приложение DartExpense

Взаимодействие между JavaScript и Dart осуществляется в виде передачи сообщений, как правило, в формате JSON, между разными виртуальными машинами. Приложение DartExpense, исполняемое VM Dart, отправляет сообщение функции, написанной на JavaScript, в котором требует нарисовать диаграмму в указанном элементе `<div>`. Наш JavaScript-код получает это сообщение и вызывает библиотеку, которая строит и отрисовывает диаграмму.

Примечание. Далее в тексте, говоря о VM Dart, мы имеем в виду как код, написанный на Dart, так и Dart-код, транслированный на JavaScript, а говоря о VM JavaScript, – код, изначально написанный на JavaScript. Dart-код, транслированный на JavaScript, следует рассматривать именно как Dart-код и отделять от «родного» JavaScript-кода.

В основе механизма передачи данных между Dart и JavaScript лежит метод `postMessage()` объекта браузера `window`, предоставляемого библиотекой `dart:html`. При таком подходе принимающая сторона должна просто добавить прослушиватель событий, которому будут передаваться полученные сообщения. Браузер принимает сообщения от одной VM и переправляет их через границу, за которой их

может получить другая ВМ. На рис. 12.3 изображен поток данных между двумя ВМ.

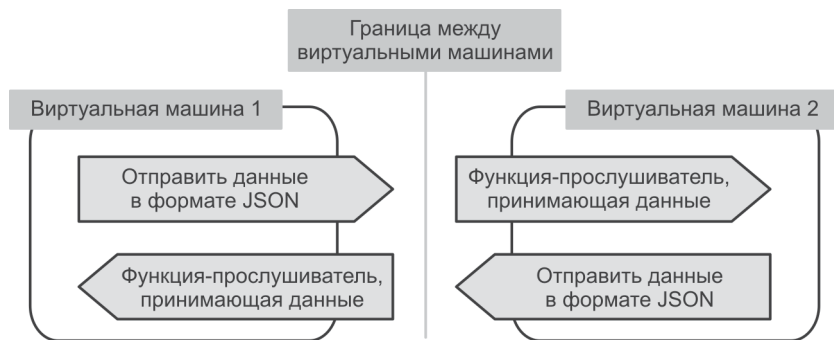


Рис. 12.3. Отправка данных от одной ВМ другой. На принимающей стороне должна существовать функция-прослушиватель

Функция `postMessage()` посылает сообщения всем прослушивателям – в Dart-приложении, в JavaScript-коде или зарегистрированным в сторонних импортированных скриптах. Каждый прослушиватель должен проанализировать полученные данные и решить, адресованы они ему или нет.

Чтобы реализовать такое взаимодействие, мы будем передавать функции `postMessage()` данные в формате JSON и добавим поле `type`, равное `dart2js` или `js2dart`, чтобы принимающая функция могла понять, адресовано ли сообщение ей. Мы добавим также поле `action`, чтобы можно было зарегистрировать несколько принимающих функций. Пока что у нас будет только одно действие – `chart`, но впоследствии появятся и другие. Сводные данные о расходах передаются в поле `payload`. Формат и поток данных от Dart к JavaScript показаны на рис. 12.4.

Браузерная функция `postMessage(data, targetOrigin)` принимает два параметра: посылаемые данные в формате JSON и `targetOrigin` – URL-адрес принимающей страницы. Иначе говоря, если данные посылаются странице `http://localhost:8080/DartExpense.html`, то `targetOrigin` тоже равен `http://localhost:8080/DartExpense.html`. В большинстве случаев можно использовать значение `window.location.href`, потому что данные посылаются той же самой странице, но возможность задавать это значение позволяет передавать сообщения меж-

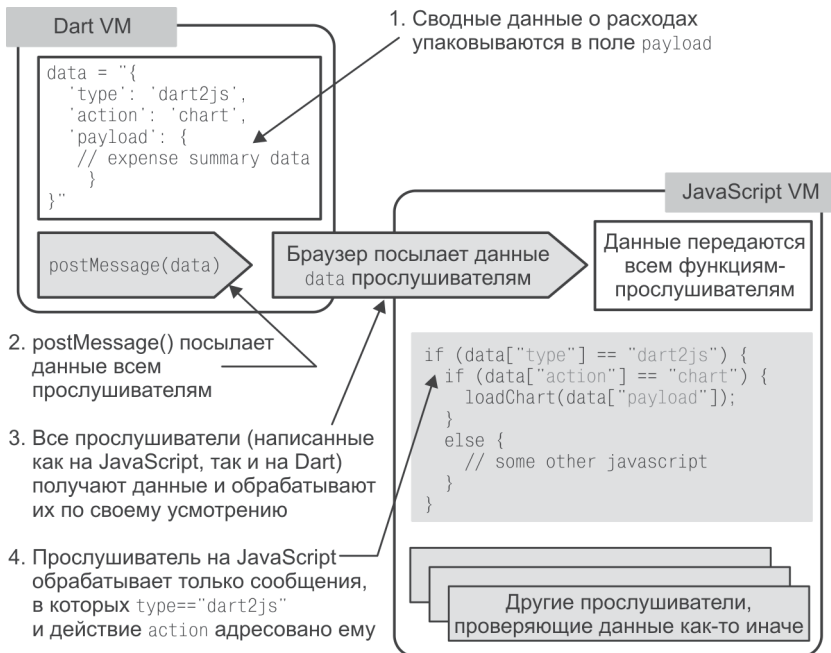


Рис. 12.4. Функция `postMessage()` посылает данные всем прослушивателям, поэтому каждый прослушиватель должен анализировать, предназначены ли данные именно ему

ду разными IFRAME (независимыми веб-страницами, возможно, с другого сервера, внедренными в другую страницу). Для простоты создадим вспомогательную функцию `sendToJavaScript()`, которая принимает действие (action) и полезную нагрузку (payload).

12.1.1. Отправка данных из Dart в JavaScript

Для отправки данных из Dart в JavaScript мы написали вспомогательную функцию `sendToJavaScript()` (см. листинг 12.1), которая помещает в поле `type` словаря `data` строку `dart2js`, в поля `action` и `payload` – переданные ей параметры (последний содержит сводные данные о расходах), преобразует словарь `data` в формат JSON, а затем вызывает функцию `window.postMessage()`. Поместите эту функцию в исходный файл `app.dart`, содержащий приложение `DartExpense`.

Листинг 12.1. app.dart: функция `sendToJavaScript()`, вызывающая `postMessage()`*// остальные функции и классы опущены*

```

sendToJavaScript(String action, var payload) {
  var data = new Map<String, Object>(); ← Создаем словарь для хранения данных
  data["type"] = "dart2js";           ← Указываем тип,
  data["action"] = action;             ← действие
  data["payload"] = payload;           ← и полезную нагрузку
  var jsonData = JSON.stringify(data); ← Преобразуем словарь в формат JSON
  window.postMessage(jsonData, window.location.href); ← С помощью postMessage()
}                                     отправляем данные всем
                                     прослушивателям

```

Разобравшись с тем, как отправлять данные JavaScript-функции, мы должны подготовить данные в приложении DartExpense. Нам необходима сводка всех расходов, в которой для каждого типа расхода будет содержаться общая сумма. На рис. 12.5 показано, как из исходных данных формируется агрегат, необходимый для построения круговой диаграммы.

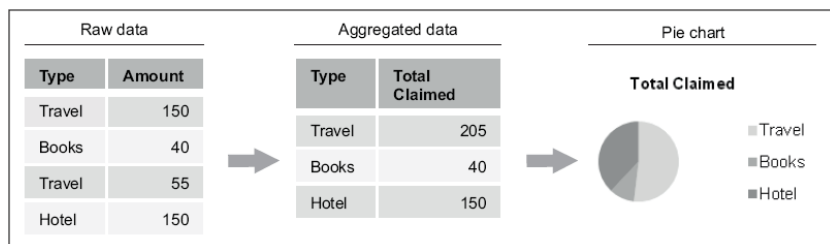


Рис. 12.5. Как данные приложения DartExpense преобразуются из исходной формы в форму, пригодную для построения круговой диаграммы

Для подготовки агрегированных данных мы напишем стереотипную функцию `getAggregatedData()`. Она будет возвращать словарь `Map<ExpenseType, double>`, в котором каждому типу расхода соответствует суммарная величина расходов этого типа. Для библиотеки построения графиков сводные данные должны быть представлены в четко определенном формате: список пар ключ–значение, в котором каждая пара сама представляет собой список. Первая пара описывает заголовок. Ниже приведен пример данных:

```
[
  ['Type', 'Amount'],
  ['Travel', 205.0],
  ['Books', 40.0],
  ['Hotel', 150.0]
]
```

← Внешний список пар ключ–значение

← Информация для заголовка

Сводные данные о расходах

При нажатии на кнопку **View Chart** приложение вызывает функцию `getAggregatedData()`, которая генерирует сводные данные о расходах и передает результат классу `ChartView`.

Класс `ChartView`, приведенный в листинге 12.2, реализует интерфейс `View`, — точно так же, как уже имеющиеся представления списка и редактирования. В нем есть ссылки на элемент `rootElement`, в котором размещается содержимое представления, и на элемент `actions` для ассоциированных с ним кнопок. Конструктор `ChartView` создает `<div>` с `id=chartView`, в котором JavaScript-библиотека строит круговую диаграмму. Типы и общие суммы расходов помещаются в список `payload`, который должен быть передан библиотеке построения графиков. И наконец, мы вызываем написанную ранее функцию `sendToJavaScript()`, передавая в качестве действия `chart`. Она преобразует полезную нагрузку `payload` в формат JSON и отправляет данные всем прослушивателям с помощью `postMessage()`.

Листинг 12.2. `ui_chart.dart`: класс `ChartView`, который вызывает JavaScript-библиотеку построения графиков

```
class ChartView implements View {
  DivElement rootElement;
  DivElement actions;

  ChartView(Map<ExpenseType, double> expenseSummary) {
    _buildView(expenseSummary);
    _buildActions();
  }

  _buildView (var expenseSummary) {

    rootElement = new Element.html("""
      <div id='chartView'
        style='width:500px;
        height:150px'></div>""");

    List payload = new List();
    payload.add(["Type", "Amount"]);
```

Конструктор принимает
сводные данные
expenseSummary

Создаем <div>
с идентификатором chartView,
необходимый
JavaScript-библиотеке

Создаем список для хранения полезной нагрузки
и добавляем в него строку заголовка

```
for (var expenseType in expenseSummary.getKeys()) {  
    var totalAmount = expenseSummary[expenseType];  
    payload.add([expenseType.name, totalAmount]);  
}  
  
var action = "chart";  
sendToJavaScript(action, payload); ← Отправляем данные JavaScript-функции  
}  
  
_buildActions() {  
    // добавление кнопки "Return to List" опущено  
}  
}
```

Но это только полдела. Dart-программа отправила данные, но предстоит еще написать на JavaScript функцию-прослушиватель, которая эти данные получит.

12.1.2. Получение в JavaScript данных, посланных из Dart

Чтобы сделать что-то полезное с данными, отправленными функцией `postMessage()`, необходимо написать JavaScript-функцию, которая их примет. Эту функцию мы затем присоединим к объекту `window` с помощью функции `window.addEventListener()`. Последняя принимает два параметра: тип прослушиваемого события (в данном случае `message`) и JavaScript-функцию обратного вызова.

Примечание. Весь написанный в этом разделе JavaScript-код размещен непосредственно в тегах `<script>` в файле `DartExpense.html`. Внешне Dart и JavaScript похожи.

В листинге ниже приведен JavaScript-код, который получает сводные данные о расходах и печатает полезную нагрузку на консоли разработчика в браузере.

Листинг 12.3. `DartExpense.html`: JavaScript-код, принимающий данные от функции `postMessage()`

```
<!DOCTYPE html>
```

```
<html>  
  <head>
```

```

<!-- остальные теги в секции заголовка опущены -->
<script type="text/javascript">  ← Тер <script>, содержащий JavaScript-код
    function receiveFromDart(event) {  ← Объявляем JavaScript-функцию receiveFromDart()
        var data = JSON.parse(event.data);  ← Преобразуем данные из JSON в JavaScript
        var type = data["type"];           | Получаем
        var action = data["action"];       | данные
        var payload = data["payload"];     | из JavaScript-словаря
        console.log(type + ", " + action + ", " + payload);  ← Выводим их
                                                на консоль
    }                                       Добавляем
    window.addEventListener("message", receiveFromDart);  ← функцию-прослушиватель
                                                         в объект window
</script>
</head>
<body>
    <div id="dartexpense"></div>
    <script type="application/dart" src="DartExpense.dart"></script>
    <script src="dart.js"></script>
</body>
</html>

```

Стандартный HTML-код
для размещения Dart-приложения

Если вызвать в Dart-коде функцию `sendToJavaScript()`, то эта функция распечатает сводные данные о расходах на консоли разработчика. Убедившись, что мы можем получить данные в JavaScript, можно приступить к использованию библиотеки Google Chart. Тут есть две стороны: собственно API библиотеки и функция, из которой API вызывается. Сам API мы добавляем на страницу с помощью ссылки на скрипт `jsapi`, размещенный на сайте Google:

```

<script type="text/javascript" src="http://www.google.com/jsapi"></script>

```

В результате подгружается код, необходимый для загрузки именно Chart API из всех доступных API Google. Чтобы использовать Chart API, нам понадобится добавить еще одну JavaScript-функцию, `loadPieChart(chartData)`, которая принимает сводные данные о расходах и строит из них круговую диаграмму. Эта функция загружает библиотеку построения графиков и принимает функцию `drawChart()`, которая нарисует диаграмму, после того как библиотека будет полностью загружена. В листинге 12.4 приведены обе эти функции, а также модифицированная JavaScript-функция `receiveFromDart()`, которая проверяет, что поступившие от `postMessage()` предназначены именно ей, и передает полезную нагрузку функции `loadPieChart()`.

Листинг 12.4. Вызов библиотеки Chart из функции receiveFromDart()

```

<!-- прочий html-код опущен -->
<head>
    <script type="text/javascript" src="http://www.google.com/jsapi">
    </script>

    <script type="text/javascript">

        function receiveFromDart(event) {
            var data = JSON.parse(event.data);
            if (data["type"] == "dart2js") {
                if (data["action"] == "chart") {
                    var chartData = data["payload"];
                    loadPieChart(payload);
                }
            }
        }

        window.addEventListener("message", receiveFromDart);

        function loadPieChart(chartData) {
            var packageData = { packages:["corechart"], callback: drawChart};
            google.load("visualization", "1", packageData);

            function drawChart() {
                var data = google.visualization.arrayToDataTable(chartData);
                var options = {title: 'Expense Types'};

                var divElement = document.getElementById("chartView");
                var chart = new google.visualization.PieChart(divElement);

                chart.draw(data, options);
            }
        }
    </script>
</head>
<!-- прочий html-код опущен -->

```

Включается Google API

Извлекаем полезную нагрузку –
 ← данные о расходах
 ← Передаем данные функции loadPieChart()

После загрузки пакета corechart производится
 обратный вызов функции drawChart()

Преобразуем chartData в нужный формат
 и задаем параметры

Создаем объект PieChart внутри
 указанного элемента <div>

← Рисуем
 круговую
 диаграмму

Можете поэкспериментировать с другими параметрами, например задать `is3d:true`, чтобы получилась трехмерная диаграмма, или поиграть с размерами и цветами. Поиск в сети по запросу «Google Chart API» выдаст полную информацию о параметрах и других ви-

дах графиков, например столбчатых диаграммах и линейных графиках. Все это находится на сайте <http://developers.google.com>.

Научившись вызывать JavaScript из Dart, мы должны теперь решить обратную задачу – вызвать Dart из JavaScript. Идея та же: JavaScript-код посылает данные с помощью функции `postMessage()`, а Dart-код прослушивает событие, генерируемое браузером.

12.1.3. Отправка данных из JavaScript в Dart

Сейчас приложение DartExpense умеет посылать данные JavaScript-библиотеке Chart. Но надо довести дело до конца и научить JavaScript-функцию `drawChart()` отправлять обратное сообщение Dart-коду, когда диаграмма будет нарисована. В ответ мы сделаем доступной кнопку **Return to List** (Возврат к списку) на панели действий. Поток данных от Dart к JavaScript и обратно показан на рис. 12.6.

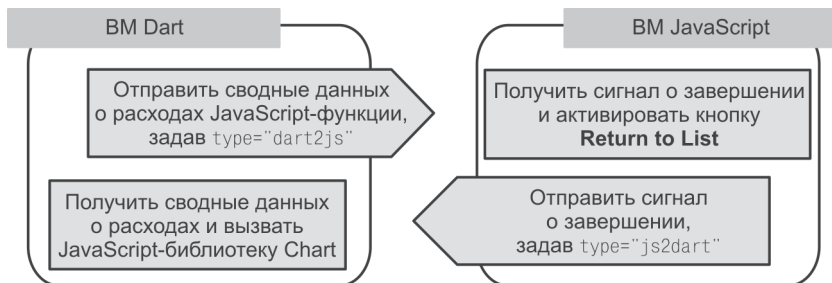


Рис. 12.6. Поток данных от Dart к JavaScript и обратно

Точно так же, как мы написали функцию `sendToJavaScript()` на Dart, напомним функцию `sendToDart(action, payload)` на JavaScript. Она упакует действие и полезную нагрузку в JSON-строку и отправит ее браузеру с помощью функции `window.postMessage()`, как показано в листинге ниже. Необходимо также модифицировать функцию `drawChart()`, чтобы она вызывала `sendToDart()`, когда диаграмма будет нарисована.

Листинг 12.5. Вспомогательная JavaScript-функция `sendToDart()`

```
<script type="text/javascript">
  function loadPieChart(chartData) {
```

```

var packageData = { packages:["corechart"], callback: drawChart};
google.load("visualization", "1", packageData);

function drawChart() {
    var data = google.visualization.arrayToDataTable(chartData);
    var options = {title: 'Expense Types'};

    var divElement = document.getElementById("chartView");
    var chart = new google.visualization.PieChart(divElement);

    chart.draw(data, options);
    sendToDart("chartComplete", "finished"); ← Уведомить Dart о том, что диаграмма
    }                                         успешно нарисована
}

function sendToDart(action, payload) {
    var data = {};
    data["type"] = "js2dart";
    data["action"] = action;
    data["payload"] = payload;
    var jsonData = JSON.stringify(data); ← Преобразуем данные в формат JSON
    window.postMessage(jsonData,window.location.href); ← Отправляем данные с помощью
    }                                                     функции postMessage()
}
</script>

```

Теперь JavaScript-код посылает данные всем прослушивателям события `postMessage()`, поэтому осталось только написать такой прослушиватель для Dart-части приложения. С функцией `postMessage()` связана только одна очередь, поэтому любые сообщения – отправленные как из Dart, так и из JavaScript, – доставляются всем прослушивателям. Поэтому каждый должен проверить, ему ли адресовано сообщение. Для этого предназначено поле `data["type"]`. JavaScript-прослушиватель обрабатывает только сообщения, в которых `data["type"]` равно `dart2js`, а Dart-прослушиватель – только сообщения, в которых `data["type"]` равно `js2dart`.

Примечание. Начиная с этого места, весь код написан только на Dart.

В следующем листинге приведен модифицированный класс `ChartView`. Он вызывает внешнюю библиотеку для построения диаграммы, а затем ждет ответа JavaScript о завершении работы, после чего можно сделать доступной кнопку **Return to List**.

Листинг 12.6. ui_chart.dart: получение данных в Dart от JavaScript

```

class ChartView implements View {
  DivElement rootElement;
  DivElement actions;
  ButtonElement returnToListButton;

  // конструктор опущен

  _buildView (var expenseSummary) {
    // построение полезной нагрузки опущено

    var action = "chart";
    sendToJavaScript(action, payload);

    onFinishedListener(event) {
      var data = JSON.parse(event.data); ← Извлекаем данные из сообщения,
      if (data["type"] == "js2dart") {      ← отправленного postMessage()
        if (data["action"] == "chartComplete") {
          window.on.message.remove(onFinishedListener); ← Удаляем этот прослушива-
          returnToListButton.disabled = false; ← тель событий, чтобы
        }                                     больше он не вызывался
      }                                     Активируем кнопку
    }                                     returnToListButton
  };

  window.on.message.add(onFinishedListener); ← Добавляем функцию, прослушивающую
}                                             события, связанные с postMessage()

  _buildActions() {
    // добавление кнопки "Return to List" опущено
  }
}

```

Итак, мы рассмотрели передачу данных из Dart в JavaScript и обратно. Из-за наличия нескольких ВМ с собственной памятью и необходимости взаимодействовать с браузером схема оказывается сложнее, чем внедрение JavaScript-кода в Dart-код. Единственный способ взаимодействия между разными ВМ – передача сообщений.

Памятка

- Данные можно передавать из Dart-кода наружу с помощью функции `window.postMessage()`.
- JavaScript-код может прослушивать событие `message` и таким образом получать данные от Dart-кода; прослушиватель присоединяется с помощью функции `window.addMessageListener()`.

- Данные, отправленные функцией `postMessage()`, могут быть доставлены нескольким прослушивателям, поэтому каждый должен проверить, ему ли адресовано сообщение.

В следующем разделе мы воспользуемся функцией `sendToDart()` для получения данных от внешних серверов с помощью технологии JSONP, которая давно применяется в JavaScript для обхода ограничений безопасности при отправке AJAX-запросов.

12.2. Взаимодействие с внешними серверами

Одна из замечательных особенностей веб-разработки состоит в том, что мы можем обращаться к открытым API внешних серверов и получать от них нужные приложению данные. Это будет вторая внешняя система, с которой мы интегрируем Dart (см. рис. 12.7).

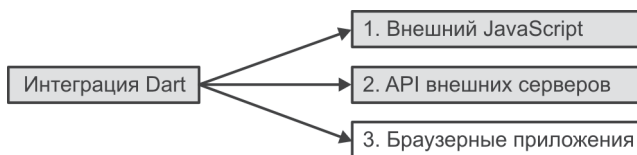


Рис. 12.7. Интеграция Dart с внешним сервером

Снова изменим приложение DartExpense, добавив функцию, которая будет конвертировать сумму расходов из долларов США в английские фунты. Для этого нам необходим текущий обменный курс, который любезно предоставляет сайт <http://openExchangeRates.org>, если обратиться по адресу <http://openexchangerates.org/api/latest.json>.

В представление редактирования мы поместим дополнительную кнопку **Convert to GBP**, которая получит текущий обменный курс и заменит введенное значение в долларах конвертированным, как показано на рис. 12.8.

Информация о курсах валют возвращается в формате JSON, как показано в следующем фрагменте, содержащем только четыре курса:

```
{
  "rates": {
    "EUR": 0.816139,
    "FJD": 1.820003,
```

```
"FKP": 0.639819,  
"GBP": 0.639819  
}  
}
```

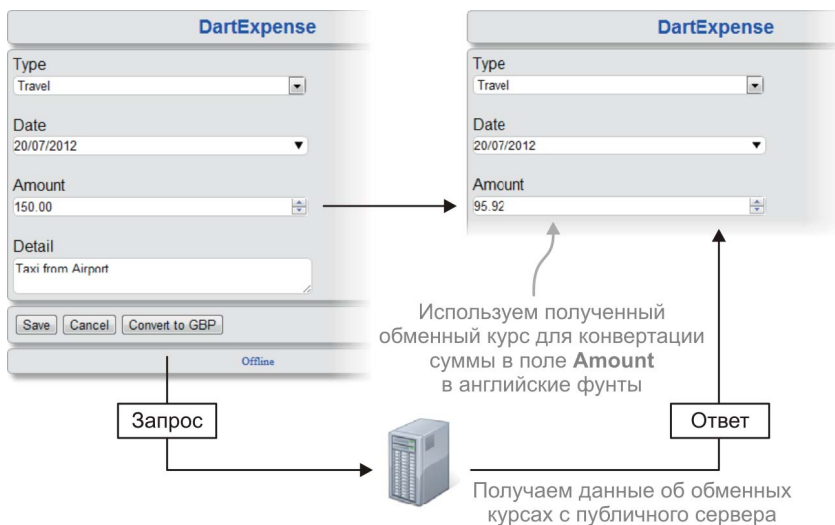


Рис. 12.8. Использование информации об обменных курсах, полученной от открытого API, для конвертации суммы из долларов США в английские фунты

Получив эти данные, программа должна найти нужный курс и умножить его на величину суммы в долларах:

```
var data = JSON.parse(exchangeRateData);  
var conversionRate = data["rates"]["GBP"];  
var gbpAmount = dollarAmount * conversionRate;
```

После этого можно записать в поле **Amount** сумму в английских фунтах. К сожалению, добраться до данных о курсах не так-то просто из-за встроенных в браузер ограничений безопасности, смысл которых – в том, чтобы запретить отправку AJAX-запросов сайту, отличающемуся от того, с какого получена страница.

12.2.1. Правило одного домена

Стандартный способ асинхронного получения данных от сервера – запрос `HttpRequest`. Этот API встроен в браузер, ему соответствует первая буква в аббревиатуре AJAX (А – асинхронный). Технология `HttpRequest`, известная также под названием XHR (от предоставляемой браузером функции `XMLHttpRequest()`), появилась в популярных браузерах примерно в 2005 году. Предназначалась она для того, чтобы веб-страница могла запросить дополнительные данные у сервера без полной перезагрузки, но вскоре был найден способ использовать ее для организации атак межсайтового скриптинга с целью пересылки пользовательских данных с одного сервера на другой. Производители браузеров противопоставили этому *правило одного домена* (same-origin policy): если страница поступила от сервера `http://dartexpense.com`, то отправлять с нее `HttpRequest`-запросы можно только этому серверу и никакому другому.

И тут возникает проблема, потому что на данный момент приложение `DartExpense` вообще не размещено ни на каком сервере, оно работает целиком в браузере (мы еще вернемся к использованию `HttpRequest` в следующей главе, когда добавим серверную часть, тоже написанную на Dart). По счастью, существует несколько способов обойти это ограничение, в том числе новый стандарт междоменового разделения ресурсов (cross-origin resource sharing – CORS), который начинают поддерживать серверы. Если сервер поддерживает CORS, а точнее добавляет в ответ специальные заголовки, то браузер не будет препятствовать использованию `HttpRequest` для получения данных от стороннего сервера. Ниже показано, как с помощью `HttpRequest` можно получить данные как от собственного сервера, так и от стороннего сервера, поддерживающего стандарт CORS:

```
var url = "http://openexchangerates.org/api/latest.json";
new HttpRequest.get(url, (response) {
    print(response.responseText);
});
```

← XHR-запрос к URL
← Сделать что-то с полученными данными

Но этот стандарт еще только входит в обиход. Надежнее воспользоваться более сложным способом запроса данных у внешнего сервера с помощью технологии JSON с набивкой (JSONP).

12.2.2. Использование JSONP для запроса данных у внешнего сервера

Технология JSONP основана на том, что скрипты в теге `<script>` можно запрашивать у любого сервера; именно так мы поступили в предыдущем разделе со скриптом `jsapi`, который скачивали с сервера Google. Вместо того чтобы запрашивать обычные данные в формате JSON, мы запрашиваем данные, обернутые JavaScript-функцией, которая называется *обратным вызовом*. Загрузив скрипт с внешнего сервера, страница выполняет его, при этом вызывается указанная функция обратного вызова, которой передаются отправленные сервером данные. Имя этой функции задается в строке запроса в виде параметра `callback=myCallbackFunctionName`, например:

`http://openexchangerates.org/api/latest.json?callback=onDataLoaded`

Этот URL указывается в свойстве `src` первого тега `<script>`, тогда как во втором определена JavaScript-функция `onDataLoaded(data)`, которая принимает данные. На рис. 12.9 показано, как это выглядит на практике.

Динамическое добавление элемента `<script>`

Первый тег `<script>`, нужный для запроса данных у сервера, можно динамически добавить в Dart, создав объект `ScriptElement` и записав в его свойство `src` нужный URL. Если добавить этот элемент в DOM, то браузер тут же запросит данные. Чтобы не допустить утечек памяти, мы добавляем, а затем удаляем тег `<script>`, как показано в следующем фрагменте:

```
ScriptElement scriptElement = new Element.tag("script"); ← Создаем элемент <script>
scriptElement.src =
↳ "http://openexchangerates.org/api/latest.json?callback=onDataLoaded";
scriptElement.type = "text/javascript";
document.head.children.add(scriptElement);
scriptElement.remove(); ← Удаляем его из DOM
```

← Добавление его в DOM приводит к отправке запроса

Получив от сервера JavaScript, содержащий обменные курсы, браузер пытается выполнить JavaScript-функцию `onDataLoaded()`. В ней мы с помощью функции `sendToDart()` отправляем данные приложению `DartExpense`, как показано ниже:

1. На страницу помещаются два скрипта: один запрашивает данные у внешнего сервера, второй содержит функцию обратного вызова, которой передаются полученные данные

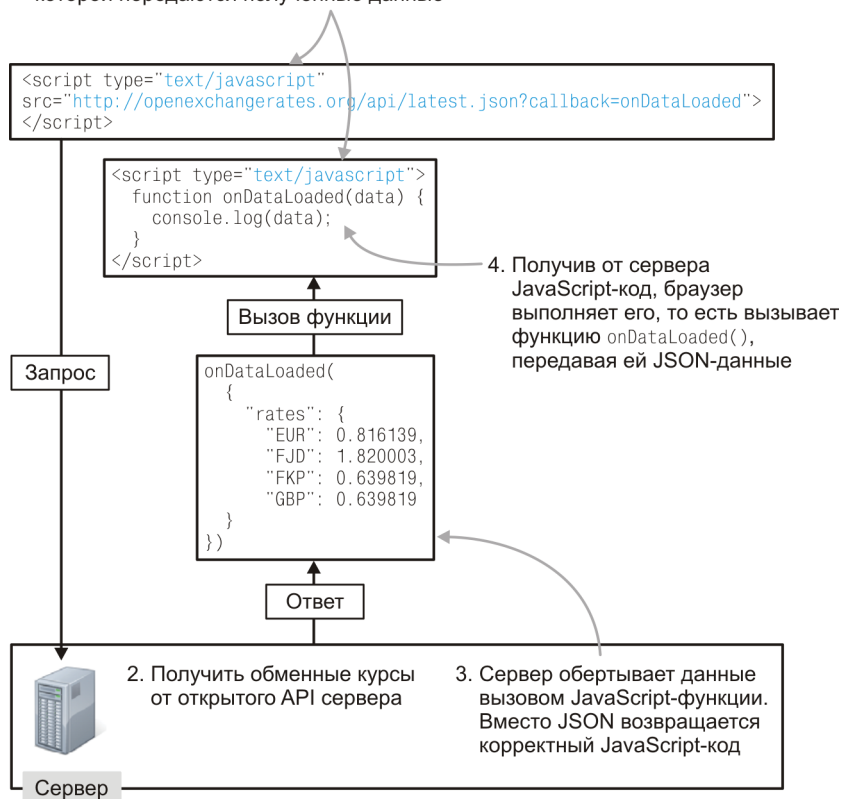


Рис. 12.9. Получение данных с помощью JSONP

```

<script type="text/javascript">
function onDataLoaded(data) {
  sendToDart("exchangeRates", data);
}
</script>

```

Последний шаг в этой цепочке – подготовить в Dart-коде функцию, которая будет обрабатывать сообщение с действием `exchangeRates`, отправленное из JavaScript-функции `onDataLoaded()`. Точно так же мы

поступали в предыдущей главе, когда писали прослушиватель сообщения с данными для диаграммы. Весь необходимый Dart-код собран в обработчике нажатия кнопки **Convert to GBP** (листинг 12.7).

Листинг 12.7. ui_edit.dart: конвертация из долларов США в английские фунты

```
class EditView implements View {
  DivElement rootElement;
  DivElement actions;

  // остальные методы и функции опущены
  _buildActions() {
    actions = new Element.tag("div");
    actions.children.add(_getSaveButton());
    actions.children.add(_getCancelButton());
    actions.children.add(_getConvertToGBPButton());
  }

  _getConvertToGBPButton() {
    ButtonElement convertButton = new Element.tag("button");
    convertButton.text = "Convert to GBP";
    convertButton.onClick.add( (event) {
      ScriptElement scriptElement = new Element.tag("script");
      scriptElement.src =
        "http://openexchangerates.org/api/latest.json?callback=onDataLoaded";
      scriptElement.type = "text/javascript";
      document.head.children.add(scriptElement);
      scriptElement.remove();
    });
  }

  onRateListener(event) {
    var data = JSON.parse(event.data);
    if (data["type"] == "js2dart" &&
        data["action"] == "exchangeRates") {
      window.on.message.remove(onRateListener);
    }
  }

  InputElement amountEl = query("#expenseAmount");
  var dollarValue = double.parse(amountEl.value);

  var payload = data["payload"];
  var gbpRate = payload["rates"]["GBP"];
  var gbpValue = dollarValue * gbpRate;
  amountEl.value = gbpValue.toStringAsFixed(2);
}
```

На панель действий добавляем
← кнопку конвертации

Создаем кнопку
конвертации

← В обработчике нажатия кнопки...

...удаляем элемент <script>,
с помощью которого получали курсы

Прочитав
данные о курсах...

...удаляем прослушиватель
← события message

Находим поле, содержа-
щее сумму в долларах

Извлекаем обменный курс
из полученных данных
и вычисляем сумму в фунтах
← Записываем сумму
в фунтах в текстовое поле

```
    window.on.message.add(onRateListener);    ← Добавляем прослушатель
  });                                         события message

  return convertButton;
}
}
```

При использовании технологии JSONP для получения данных с внешнего сервера действительно требуется написать больше стереотипного кода для реализации цепочки Dart $\Rightarrow$ `ter script` $\Rightarrow$ обратный вызов на JavaScript $\Rightarrow$ Dart. Но пока серверы не начнут в массовом порядке поддерживать стандарт CORS, правило одного домена по-прежнему будет мешать отправке AJAX-запросов сторонним серверам. Применяя описанную выше технику, вы сможете отправлять запросы сразу нескольким серверам для получения необходимых приложению данных.

Итак, мы видели, как интегрировать Dart с JavaScript и как воспользоваться этим механизмом интеграции для запроса данных у внешних серверов с помощью технологии JSONP. В следующем разделе мы рассмотрим интеграцию Dart еще с одним внешним компонентом – веб-браузером. Мы воспользуемся технологией HTML5 AppCache для запуска приложения, работающего вообще без сервера, и установим его в браузер Chrome в виде автономного пакета, который может быть опубликован в магазине Chrome Web Store.

Памятка

- Запросы `HttpRequest` можно посылать только серверам, которым поддерживают стандарт CORS, или серверу, на котором размещено приложение.
 - Технология JSONP заключается в добавлении тега `<script>`, чтобы получить запрашиваемые данные, обернутые вызовом JavaScript-функции.
 - JavaScript-функция, которой переданы полученные от сервера данные, может передать их Dart-функции с помощью рассмотренной выше функции `postMessage()`, вызываемой из функции `sendToDart()`.
-

12.3. Построение допускающих установку браузерных приложений, не требующих сервера

Среди прочих в спецификации HTML5 описана технология автономных приложений, работающих без соединения с сервером. Первоначально приложение должно быть загружено с сервера, на-

пример <http://dartexpense.com>, но после сохранения его файлов браузером пользователь может перейти по адресу <http://dartexpense.com>, и приложение загрузится, даже если соединение с Интернетом отсутствует. Сочетая технологию автономной работы с возможностью устанавливать приложения в браузер Google Chrome, мы можем создавать приложения, которые работают на мобильных или планшетных устройствах, подключающихся к Интернету лишь эпизодически, с тем же успехом, что и на настольных ПК. Это третий вариант интеграции Dart с внешними системами (см. рис. 12.10).

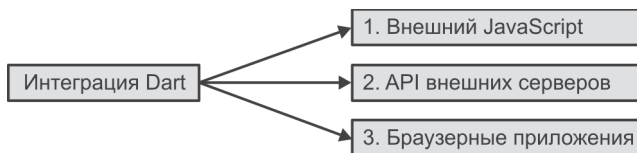


Рис. 12.10. Интеграция Dart с браузером

Чтобы приложение можно было назвать истинно автономным, оно прежде всего должно быть работоспособным в отсутствие подключения к Интернету. Для этого можно воспользоваться технологией AppCache.

12.3.1. Применение технологии AppCache для запуска приложений в автономном режиме

В технологии AppCache используется специальный *файл манифеста кэша*, который содержит список файлов – HTML, Dart, JavaScript, изображений и CSS-стилей, – которые составляют приложение. Чтобы указать, что приложение будет использовать AppCache, необходимо добавить атрибут `manifest` в тег `html`, указав в нем имя файла манифеста кэша, как показано ниже на примере файла `DartExpense.html`:

```
<!DOCTYPE html> ← DOCTYPE, соответствующий версии HTML5
```

```
<html manifest="dartexpense.appcache"> ← В атрибуте manifest находится  
// содержимое страницы опущено ссылка на файл dartexpense.appcache  
</html>
```

Во время загрузки приложения браузер пытается подключиться к серверу и прочитать файл манифеста `dartexpense.appcache`. Если это получается, то перечисленные в манифесте файлы сохраняются локально. Если подключиться не удастся, то браузер использует локально сохраненные файлы вместо реальных файлов с сервера.

Примечание. Несмотря на отсутствие реального сервера, при использовании редактора Dart приложение обслуживается встроенным отладочным сервером. Чтобы убедиться, что приложение может работать автономно, необходимо закрыть редактор. Можно вместо этого разместить приложение, состоящее только из статических файлов, на любом веб-сервере, например Apache или IIS.

Именно файл манифеста несет ответственность за все это волшебство. Он состоит из трех секций, показанных на рис. 12.11. В них перечислены файлы, которые следует кэшировать локально, файлы, которые доступны только при наличии подключения к сети, и локальные альтернативы, которые можно использовать, когда сеть недоступна. На рис. 12.11 показаны файлы, которые нужно использовать при работе с Google Chrome, а не Dartium, в частности версия приложения, преобразованная из Dart в JavaScript.

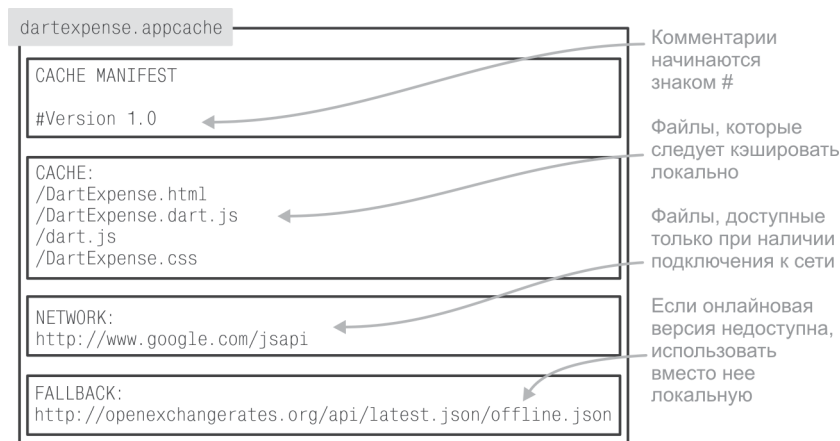


Рис. 12.11. В файле `dartexpense.appcache` указано, какие части приложения допускают кэширование, а какие – нет

Рассмотрим эти секции по очереди.

Файлы, подлежащие локальному кэшированию

В секции `CACHE` перечислены файлы, которые браузер должен кэшировать локально. Один раз загрузив их при первом обращении к приложению, браузер затем обслуживает эти файлы из кэша, не запрашивая у сервера. Этим `AppCache` отличается от обычного кэша, который требует, чтобы сервер был доступен. Файлы в `AppCache` полностью автономны – если вы попытаетесь перейти по адресу `http://127.0.0.1:8080/DartExpense.html`, даже в отсутствие сервера, файл все равно загрузится. Это очень удобно при работе с устройствами, которые подключаются к сети лишь эпизодически, например телефонами, планшетами и даже ноутбуками, поскольку часть функциональности приложения доступна пользователю даже в автономном режиме.

Предупреждение. `AppCache` работает, только если удастся закэшировать все файлы, перечисленные в манифесте. Если при первом обращении к приложению сервер не вернул хотя бы один файл, то никакие файлы не сохраняются.

Но раз браузер никогда не обращается к серверу, то что произойдет, если файл `DartExpense.html` или какая-то часть кода приложения изменится? Браузер пытается загрузить файл манифеста и проверить, изменился ли он. Если файл манифеста не удалось загрузить или он не изменился, то браузер использует локальную копию. Если приложение изменилось, то уведомить об этом браузер проще всего, изменив в файле манифеста строку комментария, например написав `#Version 1.1` вместо `#Version 1.0`. Одно лишь это действие заставит браузер заново поместить в кэш все файлы в следующий раз, как сервер станет доступным.

Благодаря чтению файлов из локального кэша вместо сервера время запуска приложения уменьшается даже в случае, когда оно содержит много кода и графики. Коль скоро эти ресурсы находятся на локальной машине, при повторном обращении к приложению не требуется загружать их с сервера.

Совет. В Google Chrome посмотреть, какие сайты хранят файлы в кэше `AppCache`, можно, перейдя на страницу <chrome://appcache-internals>. Вы удивитесь, узнав, сколько сайтов пользуются этой возможностью. А что, если какой-то части приложения для работы нужны динамические данные или внешние API? Как раз для этого предназначена секция `NETWORK`.

Файлы, доступные только при наличии подключения к сети

Иногда приложение в целом отлично работает автономно, но каким-то его частям нужны динамические данные с вашего собственного или какого-то внешнего сервера. Обращения к URL-адресам, перечисленным в секции NETWORK (они могут содержать и метасимволы, например `http://www.google.com/*`), производятся в обход кэша, даже если браузер работает в автономном режиме. Чтобы при таких условиях приложение оставалось работоспособным, необходимо уметь узнавать о наличии подключения, иначе попытка доступа к URL завершится ошибкой. Мы можем изменить приложение DartExpense, сделав кнопку **View Chart** доступной, только если есть подключение. Для этого нужно проверить булевское свойство `window.navigator.onLine`, как показано в следующем листинге.

Листинг 12.8. ui_list.dart: кнопка View Chart доступна, только если есть подключение к сети

```
class ListView implements View {  
    // остальные свойства и методы опущены  
  
    _getChartButton() {  
        ButtonElement chartButton = new Element.tag("button");  
        chartButton.text = "Chart";  
        chartButton.on.click.add( (e) {  
            navigate(ViewType.CHART, null);  
        });  
        if (window.navigator.onLine == false) {  
            chartButton.disabled = true;  
        }  
  
        return chartButton;  
    }  
}
```

← Отключаем часть пользовательского интерфейса, если браузер работает в автономном режиме

Включение или отключение части функций в зависимости от того, есть подключение к сети или нет, в сочетании с использованием секции NETWORK в файле манифеста, позволяет постепенно снижать функциональность при переходе к автономному режиму. Этому способствует также третья секция манифеста — FALLBACK.

Переход к локальному файлу, когда сетевой недоступен

Для файлов и URL-адресов, перечисленных в секции NETWORK, необходимо явно проверять, работает ли приложение в режиме подключения к сети. Что же касается секции FALLBACK, то она дает элегантное решение, позволяющее вернуть локальную версию файла только в том случае, когда сетевая недоступна. Локальная версия может содержать фиктивные данные или состояние приложения на случай недоступности сети. В приложении DartExpense для конвертации из долларов в фунты мы обращаемся к внешнему API за обменными курсами. Можно было бы подготовиться к отсутствию сети, указав в секции FALLBACK сначала реальный URL, а за ним локальный файл, который будет использоваться, когда подключения нет:

FALLBACK:

<http://openexchangerates.org/api/latest.json> /offline.json

В локальном файле будут храниться не самые актуальные обменные курсы. Можно изменить приложение DartExpense, так чтобы поставить пользователя в известность об этом. В листинге 12.9 мы модифицировали обработчик кнопки `convertButton` — теперь он не только запрашивает курс, но и предупреждает пользователя, что данные могли устареть и для получения действующего курса нужно подключиться к сети. Чтобы это было возможным, при недоступности внешнего сервера технология AppCache прозрачно для программиста возвращает не ошибку, а локальный файл `offline.json` с данными об обменных курсах.

Листинг 12.9. `ui_edit.dart`

// остальная часть класса EditView опущена

```
convertButton.onClick.add( (e) {  
  InputElement amountEl = document.query("#expenseAmount");  
  var dollarValue = double.parse(amountEl.value);  
  
  ScriptElement scriptElement = new Element.tag("script");  
  scriptElement.src =  
    ↪ "http://openexchangerates.org/api/latest.json?callback=onDataLoaded";  
  scriptElement.type = "text/javascript";  
  document.head.children.add(scriptElement);  
  scriptElement.remove();  
  ↪
```

Не заглядываем в кэш, если подключение к сети есть, в противном случае возвращаем локальный файл `offline.json`


```
onRateListener(event) {  
    var data = JSON.parse(event.data);  
    if (data["type"] == "js2dart" && data["action"] == "exchangeRates")  
    {  
        window.on.message.remove(onRateListener);  
  
        var payload = data["payload"];  
        var gbpRate = payload["rates"]["GBP"];  
        var gbpValue = dollarValue * gbpRate;  
        amountEl.value = gbpValue.toStringAsFixed(2);  
        if (window.navigator.onLine == false) {  
            window.alert("Для получения обменных курсов подключитесь к сети");  
        }  
    }  
};  
  
window.on.message.add(onRateListener);  
});
```

Эту проверку следует делать,
потому что обращение за данными
в сети работает даже в автономном режиме

В совокупности эти три секции описывают ресурсы, которые всегда берутся из локального кэша, никогда не берутся из кэша и берутся из кэша только в том случае, когда подключения к сети нет. В результате приложение остается полезным как при наличии подключения к сети, так и в автономном режиме.

И последний шаг – превратить приложение из URL-адреса, возвращенного поисковой системой, в нечто, что пользователь сможет установить в Google Chrome.

12.3.2. Создание пакета приложения, допускающего установку в Chrome

Создание пакета приложения для браузера Chrome дает ряд преимуществ разработчику, в частности снимает ограничение в 5 Мб на размер локального хранилища и позволяет решить вечную проблему: как привлечь пользователей. Написав и опубликовав приложение в сети, вы должны еще приложить немало усилий к тому, чтобы пользователи вообще нашли его, а потом еще и вернулись. Зачастую пользователь поработает с приложением раз-другой, а потом забывает про него или не может вспомнить его URL-адрес. Если предложить пользователю приложение, допускающее установку в Chrome, а именно это позволяет сделать магазин Chrome Web

Store, то его название и значок будут видны пользователю всякий раз, как он открывает в Chrome новую вкладку (см. рис. 12.12).

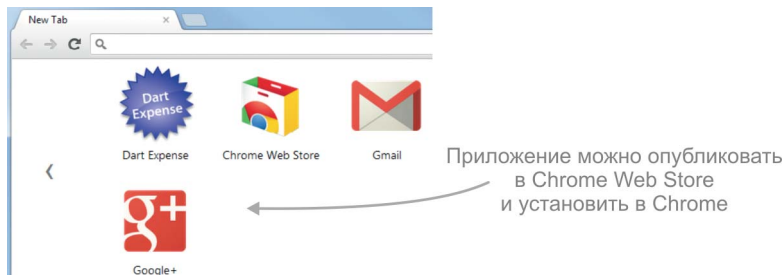


Рис. 12.12. Если приложение установлено в Chrome, то его значок регулярно появляется перед пользователем

Чтобы установить приложение в Chrome, понадобится преобразованная в JavaScript версия Dart-приложения (до тех пор, пока основная версия Chrome не начнет поддерживать Dart). Чтобы превратить свое приложение в пакет для Chrome, вы должны включить дополнительный файл – манифест Chrome (это не то же самое, что манифест AppCache, о котором мы говорили выше, но оба файла могут существовать совместно) и значок размером 128×128 пикселей.

Манифест Chrome – это файл `manifest.json` в формате JSON, в котором хранятся конфигурационные параметры приложения. Этот файл содержит три секции: метаданные приложения (имя, описание и файл значка), URL-адреса, к которым приложение будет обращаться и откуда будет запускаться, и список необходимых приложению разрешений. В предыдущей главе мы видели, что размер локального хранилища ограничен 5 Мб. В приложении Chrome можно запросить разрешение на создание хранилища неограниченного размера. Это позволит пользователю локально хранить гораздо больше данных. Упомянем некоторые другие разрешения: доступ к истории навигации в браузере, доступ к буферу обмена, доступ к закладкам. Полный перечень можно получить, поискав в сети «Chrome Manifest Permissions». В листинге ниже приведен пример файла `manifest.json` для приложения DartExpense (работающего в локальном режиме).

Листинг 12.10. manifest.json: параметры устанавливаемого в Chrome приложения

```
{
  "name": "Dart Expense",
  "description": "Track Expenses",
  "version": "1",
  "icons": {
    "128": "icon_128.png"
  },

  "app": {
    "urls": [
      "*/127.0.0.1/",
      "*/www.google.com",
      "*/openexchangerates.org"
    ],
    "launch": {
      "web_url": "http://127.0.0.1:3030/DartExpense.html"
    }
  },

  "permissions": [
    "unlimitedStorage"
  ]
}
```

Метаданные приложения

URL-адреса, к которым будет обращаться приложение

← URL, откуда запускается приложение

Запрошенные разрешения

Подготовив файл `manifest.json` и значок, поместите их в папку со всеми прочими файлами приложения. Список файлов может выглядеть так:

```
c:\Projects\DartExpense\DartExpense.html
c:\Projects\DartExpense\DartExpense.dart.js
c:\Projects\DartExpense\DartExpense.css
c:\Projects\DartExpense\dart.js
c:\Projects\DartExpense\dartexpense.appcache
c:\Projects\DartExpense\manifest.json
c:\Projects\DartExpense\icon_128.png
```

Файлы, входящие в состав приложения

Необязательный файл манифеста

← AppCache

← Манифест приложения Chrome

← Значок приложения Chrome

Далее есть два пути. Будучи разработчиком, вы, наверное, захотите проверить, как приложение будет работать после установки. К счастью, в Chrome есть режим разработчика для установки приложений. Но сначала нужно открыть окно **Chrome Extensions** (Расширения) – из меню или перейдя по адресу `chrome://chrome/`

extensions, – где вы увидите флажок **Developer Mode** (Режим разработчика) (см. рис. 12.13). Если отметить этот флажок, появятся две дополнительные кнопки: **Load Unpacked Extension** (Загрузить распакованное расширение) и **Pack Extension** (Упаковка расширений).

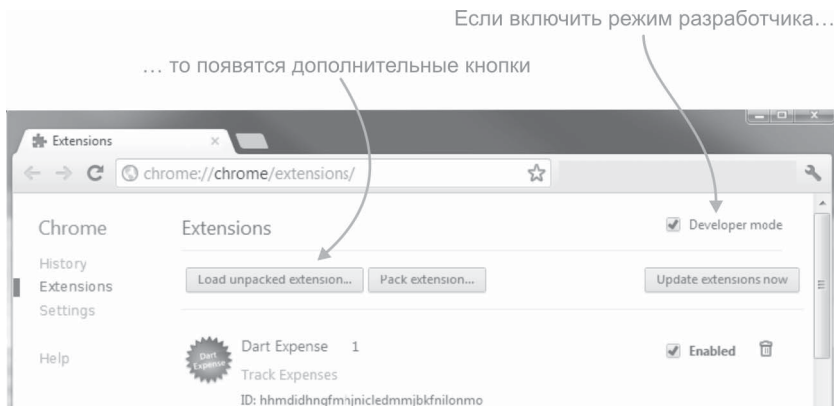


Рис. 12.13. В режиме разработчика появляются дополнительные средства для работы с устанавливаемыми приложениями

Для локального тестирования приложения нажмите кнопку **Load Unpacked Extension**. Она добавляет приложение непосредственно в локальную папку, например `c:\Projects\DartExpense`. Убедившись, что приложение работает, нажмите кнопку **Pack Extension**, чтобы преобразовать содержимое папки приложения в файл с расширением `srx` (расширение Chrome). Этот файл следует загрузить в магазин Chrome Web Store или разместить на собственных серверах. При развертывании каждой новой версии увеличивайте номер версии в файле манифеста – это даст установленному приложению возможность узнать о том, что хорошо бы обновиться, например потому, что вы добавили новые значки или требуете дополнительных разрешений.

Для приложений, размещенных в Web Store, Google предоставляет средства его обнаружения и продвижения, в частности Google Analytics, Google Plus, а также возможность добавить снимки экрана и видео, демонстрирующее приложение в действии. Все для того, чтобы помочь вам донести до мира весть о своем замечательном приложении.

Памятка

- Технология AppCache позволяет запускать приложение в автономном режиме, когда сервер недоступен.
 - AppCache также дает возможность использовать локальные файлы, если их сетевые версии недоступны.
 - Устанавливаемые в Chrome приложения позволяют снять некоторые ограничения и помогают пользователям найти ваше приложение и воспользоваться им.
-

12.4. Резюме

В этой главе мы видели, как Dart интегрируется с внешними системами благодаря возможности взаимодействия с JavaScript-кодом, работающим в отдельной ВМ. Взаимодействие возможно в обе стороны и осуществляется путем отправки сообщений функцией `postMessage()`, встроенной в браузер.

Далее мы видели, как с помощью механизма передачи сообщений между JavaScript и Dart можно запрашивать данные у внешних серверов. Для этого нужно динамически добавить тег `<script>`, который получает от стороннего API данные, обернутые вызовом JavaScript-функции. Эта функция передаст полученные данные Dart-приложению. Таким образом удастся обойти ограничения безопасности, действующие для серверов, которые пока не поддерживают стандарта CORS.

Наконец, мы узнали, что экосистема Dart позволяет выйти за пределы традиционных клиентов и серверов и создать допускающее установку приложение, способное работать, даже когда сервер недоступен, — идеальное решение для мобильных и планшетных устройств. Предлагая пользователю пакет, который тот сможет установить в свой браузер, вы можете снять некоторые ограничения, например на размер локального хранилища (5 Мб), и сделать так, что приложение постоянно будет маячить перед пользователем, когда он работает с браузером.

Это была последняя глава в третьей части книги. Мы рассмотрели обширный материал, однако спецификация HTML5 предлагает еще много API для создания потрясающих приложений для работы и досуга. В сети немало книг и примеров применения HTML5. Многие примеры написаны на JavaScript, а не Dart, но мы надеемся, что изложенные в предыдущих главах сведения помогут вам воспользоваться этими API в Dart-приложениях. К тому же большинство примеров легко переносятся с JavaScript на Dart.



В следующей части мы начнем работать с Dart на стороне сервера, рассмотрим доступ к файлам и папкам, работу по протоколу HTTP и веб-сокеты и напоследок подключим приложение DartExpense к базе данных через написанный на Dart сервер, который обслуживает запросы на статические файлы и предоставляет REST API для передачи данных.



ЧАСТЬ IV. DART НА СТОРОНЕ СЕРВЕРА

В последней части книги мы узнаем о работе Dart на сервере. Возможность использовать один и тот же код в браузере и на сервере может резко повысить продуктивность, так как нужно писать и тестировать меньше кода.

В главе 13 мы покажем, как писать простые серверные скрипты, запускаемые из командной строки. На их примере вы научитесь работать с серверными API для доступа к файлам и папкам и увидите, как передавать запрашиваемые клиентами данные по протоколу HTTP.

В главе 14 мы разовьем тему передачи данных по HTTP и наладим взаимодействие с клиентским приложением из третьей части книги. Вы научитесь использовать веб-сокеты для организации двусторонней связи между клиентом и сервером, с одним и тем же Dart-кодом на обеих сторонах. В главе 14 также показано, как хранить посылаемые клиентом данные в серверной базе.

В главе 15 мы познакомимся с технологиями изоляторов, лежащей в основе параллелизма в Dart. Изоляторы позволяют запускать в виртуальной машине Dart несколько процессов, как в многопоточных серверных языках, используя при этом более простой механизм передачи сообщений. Вы также узнаете, как архитектура изоляторов позволяет динамически загружать исполняемый код в работающее Dart-приложение, что полезно для реализации подключаемых модулей.



Глава 13. Работа с файлами и протоколом HTTP на сервере

В этой главе

- ❑ Запуск Dart-скриптов из командной строки.
- ❑ Взаимодействие с файловой системой.
- ❑ Передача данных по протоколу HTTP.

Виртуальная машина Dart может быть размещена в двух разных средах. Ранее в этой книге мы занимались использованием Dart в браузере. В этом случае владельцем ВМ был браузер Dartium, а доступ к модели DOM предоставлялся с помощью библиотеки `dart:html`. В этой главе мы приступим к изучению серверной ВМ Dart, которая не имеет доступа к модели DOM, зато может обращаться к средствам ввода-вывода операционной системы, в частности файлам и сетевым сокетами, посредством библиотеки `dart:io`.

Пример «Обозреватель файлов Dart» в этой главе – это клиент-серверное приложение, которое предоставляет работающий внутри браузера текстовый редактор Dart-файлов. Мы разработаем его в три этапа и начнем с серверной части. Серверная ВМ Dart реализована в виде исполняемого командного файла на платформах Windows, Mac и Linux. Для начала мы напишем простенький Dart-скрипт, который будет распечатывать содержимое каталога с соответствиями с аргументами, заданными в командной строке.

Затем мы рассмотрим обслуживание HTTP-запросов, поступающих от браузера. Мы покажем, как сопоставлять URL-адресам пути к файлам и как возвращать статические файлы из серверного Dart-приложения. Это позволит разработать полное клиент-серверное решение целиком на Dart.

Наконец, мы объединим скрипты распечатки содержимого каталога и работы в качестве HTTP-сервера в единое клиент-серверное приложение, которое будет возвращать список файлов и каталогов браузеру в формате JSON и даст клиентскому веб-приложению воз-

возможность перемещаться по файловой системе и читать .dart-файлы. В конце главы мы будем располагать клиент-серверным приложением, которое способно просматривать файловую систему сервера и загружать .dart-файлы (или другие текстовые файлы) в браузер для просмотра. На рис. 13.1 показана клиентская часть приложения, которая работает в браузере и взаимодействует с серверным Dart-скриптом.

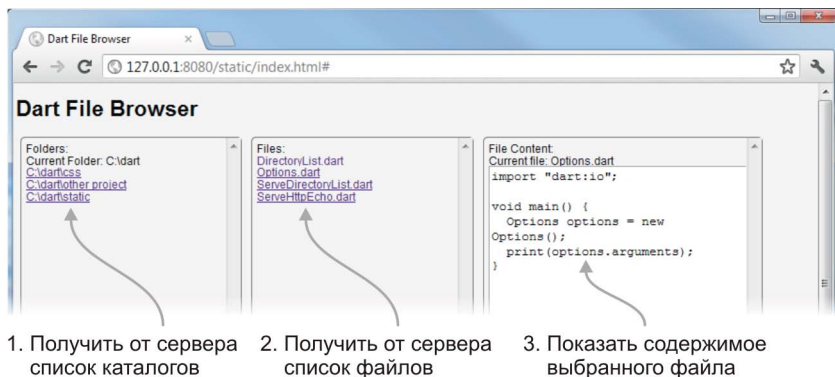


Рис. 13.1. Клиентская часть приложения «Обозреватель файлов Dart»

Но, прежде чем мы займемся пользовательским интерфейсом, необходимо познакомиться с серверной частью приложения и понять, как взаимодействовать с операционной системой для чтения файлов и каталогов.

13.1. Запуск серверного Dart-скрипта

Обозреватель файлов Dart – это серверное приложение, имеющее вид скрипта, который запускается либо из командной строки, либо из редактора Dart, и может выводить на консоль текст с помощью предложения `print()`. В этом разделе мы напишем скрипт для вывода списка подкаталогов и файлов в каталоге, путь к которому является параметром скрипта. Например, команда

```
dart.exe DirectoryList.dart C:\Dart\
```

ВЫВОДИТ

```
<DIR> c:\dart\css
<DIR> c:\dart\other_project
<DIR> c:\dart\static
<FILE> c:\dart\DirectoryList.dart
<FILE> c:\dart\Options.dart
... и так далее ...
```

Создавая новый проект в редакторе, вы можете выбрать либо веб-приложение, либо командное приложение. До сих пор мы выбирали веб-приложение, в результате чего редактор создавал два файла: с расширениями `.html` и `.dart`. У командного приложения никакой HTML-части нет, ему нужен только `.dart`-файл.

Dart-скрипт, запускаемый из командной строки, называется *серверным скриптом*. Как и в клиентском скрипте, в нем есть функция `main()`, с которой начинается выполнение. Кроме того, серверный скрипт, как и клиентский, может представлять собой библиотеку, импортировать библиотеки и даже содержать цикл обработки событий (см. рис. 13.2).

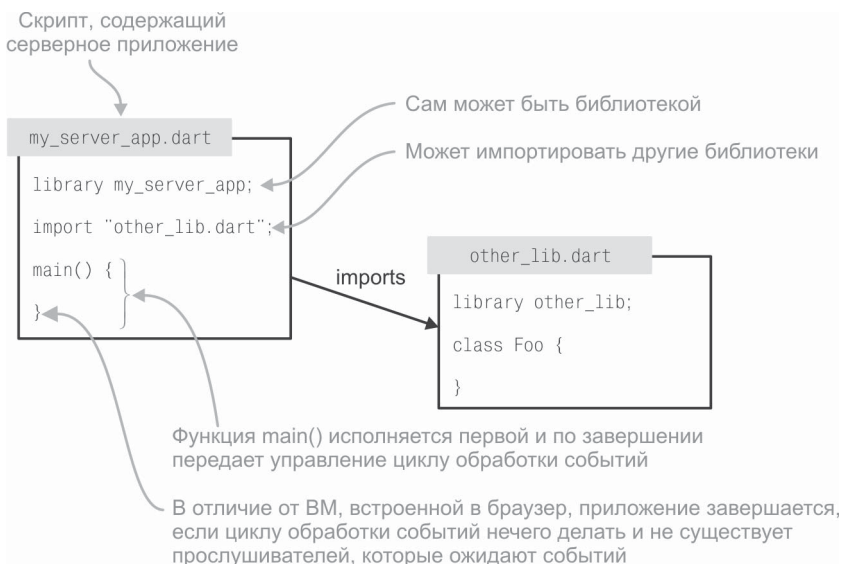


Рис. 13.2. Dart-скрипты, запускаемые из командной строки, обладают теми же возможностями и структурой, что и клиентские скрипты

На рис. 13.3 показан простой серверный скрипт, запущенный из редактора Dart и из командной строки. При запуске из редактора вывод появляется на консоли редактора, а при запуске из командной строки – в окне терминала.

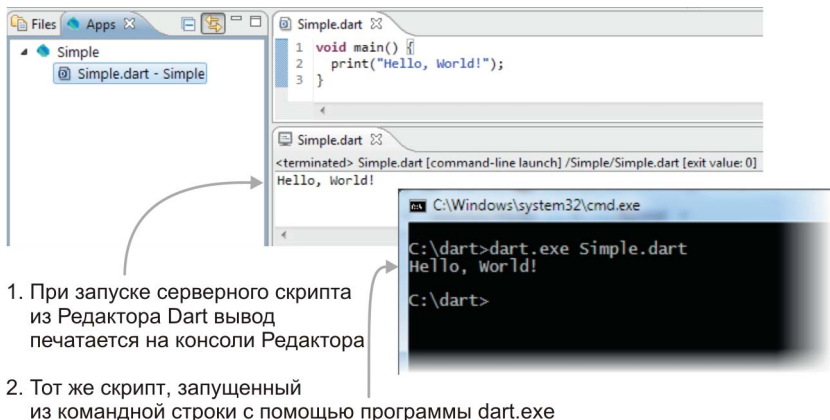


Рис. 13.3. Запуск простого серверного скрипта

Совет. Редактор Dart, который можно бесплатно скачать с сайта dartlang.org, предоставляет все инструменты, необходимые для запуска и отладки серверных скриптов. В этой главе предполагается, что вы будете пользоваться редактором Dart Editor, но с равным успехом можно запускать скрипты с помощью исполняемой программы `dart.exe`, которая находится в папке `bin/` установленного дистрибутива Dart SDK.

Существенная разница между серверными и клиентскими скриптами заключается в окружении. Клиентские скрипты работают в виртуальной машине, размещенной в браузере, и имеют доступ к браузерной модели DOM. Это дает возможность импортировать библиотеку `dart:html`. Серверные скрипты работают вне браузера, поэтому при попытке воспользоваться библиотекой `dart:html` возникнет ошибка. С другой стороны, серверные скрипты имеют доступ к библиотеке `dart:io`, в которой находятся классы и функции для доступа к файловой системе, для работы с протоколом HTTP и веб-сокетами, а также для взаимодействия по сети с помощью сетевых сокетов. Различия перечислены в табл. 13.1.

Таблица 13.1. Сводка различий между библиотеками `dart:html` и `dart:io`

<code>dart:html</code> : только в браузере	<code>dart:io</code> : только на сервере
Манипуляция элементами модели DOM	Доступ к файлам и папкам ОС
Присоединение обработчиков событий браузера	Запуск других исполняемых программ в ОС
Доступ к API браузерного хранилища	Установление соединений типа <code>HttpClient</code> с другими веб-серверами
Выполнение Ajax-запросов с помощью <code>HttpRequest</code>	Обслуживание HTTP-запросов и веб-сокетов
Отрисовка пользовательского интерфейса в браузере	Обмен данными через сокет

Из других отличий отметим, что серверный скрипт может получать аргументы, переданные ему в командной строке. Эта возможность понадобится нам для получения содержимого указанного каталога, поэтому рассмотрим ее подробнее.

Различия между циклами обработки событий на клиенте и на сервере

В предшествующих главах мы рассматривали цикл обработки событий в Dart. При запуске скрипта в браузере в функции `main()` конструируется пользовательский интерфейс и присоединяются обработчики событий, после чего управление передается циклу обработки событий. Когда из внешнего источника приходит событие, например нажатие кнопки, цикл обращается к функциям приложения, отвечающим за обработку этого события. Так продолжается до тех пор, пока приложение не будет остановлено в результате закрытия вкладки или ухода на другую страницу. На стороне сервера все выглядит несколько иначе. Цикл обработки событий по-прежнему существует, и, как мы увидим ниже, остается возможность присоединять обработчики событий и передавать управление циклу. Разница же в том, что Dart-скрипт завершается, когда не осталось обработчиков, ожидающих завершения какого-либо события. Это означает, что простой скрипт «Hello World» на рис. 13.3 завершается сразу же после печати сообщения. Эквивалентный код в браузере вернул бы управление циклу обработки событий, работающему внутри браузерной виртуальной машины Dart.

13.1.1. Доступ к аргументам командной строки

В Dart имеется тип `Options` для доступа к аргументам командной строки. Создав экземпляр `Options` с помощью ключевого слова `new`, мы получим заполненный объект, содержащий путь к вызванному

двоичному интерпретатору Dart-скриптов, путь к файлу скрипта, содержащего функцию `main()`, и список аргументов, переданных в командной строке. В листинге 13.1 приведен простой Dart-скрипт, который распечатывает все аргументы командной строки и сведения о себе самом:

```
>dart.exe TryOptions.dart hello world
C:\dart-sdk\bin\dart.exe
C:\dart\TryOptions.dart
2
[hello, world]
```

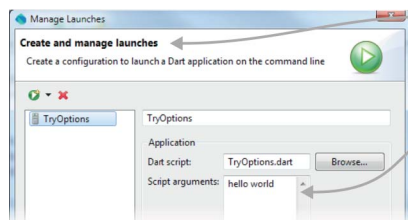
Листинг 13.1. Доступ к аргументам командной строки

```
void main() {
    Options options = new Options();    ← Создаем новый экземпляр Options

    print(options.executable);          | Получаем путь к двоичному интерпретатору Dart
    print(options.script);              | и путь к самому скрипту

    List<String> args = options.arguments; | Получаем
    print(args.length);                  | аргументы
    print(args);                         | командной строки
}
```

В редакторе Dart аргументы командной строки, передаваемые скрипту, задаются в диалоговом окне **Run** ⇒ **Manage Launches** (Запуск ⇒ Управление запуском), которое показано на рис. 13.4. В данном случае скрипту передаются два аргумента: `hello` и `world`.



В Редакторе Dart для настройки запуска скриптов из командной строки применяется диалоговое окно **Manage Launches**

Переданные скрипту аргументы доступны через объект `Options`

Рис. 13.4. Передача аргументов командной строки в редакторе Dart

Если командный скрипт запущен из редактора Dart, то результаты предложений `print()` печатаются на консоли редактора.

Теперь, зная, как добраться до аргументов командной строки, научимся использовать их в приложении «Обозреватель файлов Dart», которое будет выводить список файлов и папок. Для решения этой задачи нам понадобятся еще некоторые типы из библиотеки `dart:io`.

13.1.2. Доступ к файлам и папкам с помощью `dart:io`

У обозревателя файлов Dart есть два режима, задаваемых в командной строке. Если указать аргумент `--list` и после него путь к каталогу, то будет выведен список файлов и папок в этом каталоге.

Если же указать аргумент `--out` и после него имя файла, то на консоль будет выведено содержимое этого файла. В листинге 13.2 показано начало программы, где мы выводим сведения о возможных аргументах, если ни один аргумент не задан. Здесь же импортируется библиотека `dart:io`, где находятся типы `File` и `Directory`, необходимые для реализации функций `listDir()` и `outputFile()`, которые будут приведены ниже.

Листинг 13.2. Печать сведений об аргументах командной строки, если ни один аргумент не задан

```
import "dart:io";      ← Импортируется серверная библиотека dart:io

void main() {          Создаем объект options
  Options options = new Options(); ← для доступа к аргументам командной строки

  if (options.arguments.length != 2) { | Если число переданных аргументов
    printHelp();                       | не равно двум, печатаем справку
  }
  else if (options.arguments[0] == '--list') { | Если первый аргумент --list, передаем
    listDir(options.arguments[1]);           | второй аргумент функции listDir()
  }
  else if (options.arguments[0] == '--out') { | Если первый аргумент --out, передаем
    outputFile(options.arguments[1]);        | второй аргумент функции outputFile()
  }
}

printHelp() {
  print("""
Dart Directory Lister. Usage:
  List files and directories: --list DIR
  Output file to console : --out FILE""");
}
```

Выводим на консоль сведения о порядке вызова, применяя синтаксис многострочных строк

```
listDir(String folderPath) {  
}  
  
outputFile(String filePath) {  
}
```

Эти функции
будут реализованы
в следующих разделах

В файле `DirectoryList.dart` мы должны реализовать две функции: `listDir(folderPath)` выводит список файлов и папок в каталоге, заданном параметром `folderPath`, а `outputFile(filePath)` читает файл, заданный параметром `filePath`. В обоих случаях нам понадобятся типы `File` и `Directory`.

Перечисление файлов и папок с помощью класса `DirectoryLister`

Сначала разберем случай, когда скрипту `DirectoryList.dart` переданы аргументы вида `--list c:\dart`. Первый аргумент устанавливает режим вывода списка, а второй содержит путь к каталогу, содержимое которого нас интересует.

В классе `Directory` имеются как методы, общие с классом `File`, например `exists()`, `create()` и `delete()`, так и особый метод `list()`, возвращающий объект типа `DirectoryLister`, которым мы и воспользуемся для перечисления файлов и папок в указанном каталоге.

Типы для файлового ввода-вывода предоставляют как синхронный, так и асинхронный доступ к файловой системе, причем предпочтение отдается асинхронным операциям. Выражается это в том, что имена методов, осуществляющих синхронные операции, содержат суффикс `sync`, тогда как имена без суффикса подразумевают асинхронный доступ, например: `Directory.exists()` и `Directory.existsSync()`.

Синхронные и асинхронные методы

В таком скрипте, как `DirectoryList.dart`, между синхронным и асинхронным доступом к файлам и каталогам есть тонкое различие. Если используется синхронный метод, например `existsSync()`, то приложение будет ждать ответа от файловой системы, ничего в это время не делая. Когда файловая система ответит, метод `existsSync()` вернет `true` или `false`, после чего приложение продолжит работу.

При использовании асинхронной версии возвращается объект типа `Future<bool>`, и приложение продолжает работу, не дожидаясь ответа от файловой системы. Как только ответ придет, будущее значение будет заполнено.

Это различие несущественно в скрипте `DirectoryList.dart`, потому что пока не станет известно, существует объект в файловой системе или нет, в приложении все равно ничего не происходит. Тем не менее в примерах

ниже мы пользуемся асинхронными версиями, так как это различие начнет играть важную роль при обслуживании HTTP-запросов на отправку содержимого каталога от браузера. Если бы мы применяли блокирующие синхронные методы, то второй запрос от браузера был бы заблокирован на время, пока первый ждет ответа от файловой системы. Когда требуется, чтобы приложение масштабировалось на сотни и даже тысячи одновременных запросов, необходимо использовать неблокирующий асинхронный ввод-вывод. Синхронные методы годятся, когда нужно быстро написать простой скрипт, но для обеспечения масштабируемости рекомендуется пользоваться асинхронными.

Если вы подзабыли, в чем различие между блокирующими и неблокирующими вызовами, перечитайте обсуждение в главе 9.

Асинхронные методы, ожидающие возврата одного значения, например `exists()`, возвращают объект `Future`, который получает реальное значение в функции, переданной его методу `then()`. Методы, возвращающие несколько значений, например список файлов и папок в некотором каталоге, возвращают объект, обладающий методом, который принимает функцию, вызываемую для каждого элемента возвращенного списка по мере их получения. Это позволяет начать вывод на консоль, хотя файловая система еще не закончила возвращать данные приложению.

Применение метода `exists()` и типа `DirectoryLister` демонстрируется на рис. 13.5, где показано, как открыть каталог и получить список находящихся в нем файлов и папок. Это дает реализацию функции `listDir(folderPath)` в скрипте `DirectoryList.dart`.

Если запустить приложение с аргументами `--list c:\dart`, то оно выведет список файлов и папок в каталоге `c:\dart`:

```
<DIR> c:\dart\css
<DIR> c:\dart\other_project
<DIR> c:\dart\static
<FILE> c:\dart\DirectoryList.dart
<FILE> c:\dart\Options.dart
```

Примечание. В примерах приведены пути в ОС Windows, но программа с тем же успехом работает в Mac и Linux. Попробуйте заменить `c:\dart` путем `~/dart`, чтобы получить содержимое папки `dart` в своем домашнем каталоге.

Добившись того, что приложение возвращает список файлов в каталоге, посмотрим, как с помощью типа `File` асинхронно прочитать содержимое файла.

Чтение файлов с помощью класса *File*

Скрипт `DirectoryList.dart` можно использовать также для вывода содержимого файла, имя которого задано в командной строке после аргумента `--out`, например: `--out c:\dart\TryOptions.dart`. Эта команда должна прочитать файл и вывести его содержимое на консоль. Для этого мы воспользуемся типом `File`.



Рис. 13.5. Функция `listDir()` пользуется асинхронным методом `exists()` и классом `DirectoryLisr` для возврата списка файлов и папок в каталоге

Как и `Directory`, тип `File` предоставляет синхронные и асинхронные версии методов, например `readAsString()` и `readAsStringSync()`. Асинхронная версия возвращает объект `Future<String>`, который дает доступ к содержимому файла и пользуется неблокирующим вводом-выводом, тогда как синхронная блокирует выполнение, но возвращает значение `String`, в котором находится все содержимое файла. Асинхронный метод `readAsString()` используется следующим образом:

```

outputFile(String filePath) {
    File file = new File(filePath);
    file.readAsString().then((content) {
        print(content);
    });
}

```

Создается объект,
 ← представляющий файл с указанным путем
 ← Читается содержимое файла
 ← Когда будущее значение заполнится, печатается все содержимое файла

Здесь мы асинхронно читаем файл `c:\dart\TryOptions.dart`. После того как все данные будут прочитаны из файловой системы, значение `Future` оказывается заполненным, и все содержимое файла передается функции, заданной при вызове метода `then()` объекта `Future`.

Внутри метода `readAsString()` используется класс `InputStream`, который возвращает буферизованные последовательные данные без блокировки. Метод `read()` этого класса возвращает данные в виде объекта `List<int>`, но на самом деле это массив байтов, каждый элемент которого представляет один байт из файла `TryOptions.dart`.

Примечание. В Dart нет различных числовых типов, таких как `byte`, `word` и `long`. Вместо этого имеются родительский тип `num` и два дочерних: `int` и `double`.

В типе `String` определен конструктор `String.fromCharCode()` для создания объекта `String` из списка `List<int>`.

В листинге 13.3 класс `InputStream` используется для реализации функции `outputFile()`. Метод `onData()` вызывается всякий раз, как становится доступна очередная порция данных; чтобы получить доступ к этим данным, вызывается метод `read()` объекта `InputStream`, который может возвращать данные или `null`, если данных больше нет (рис. 13.6).

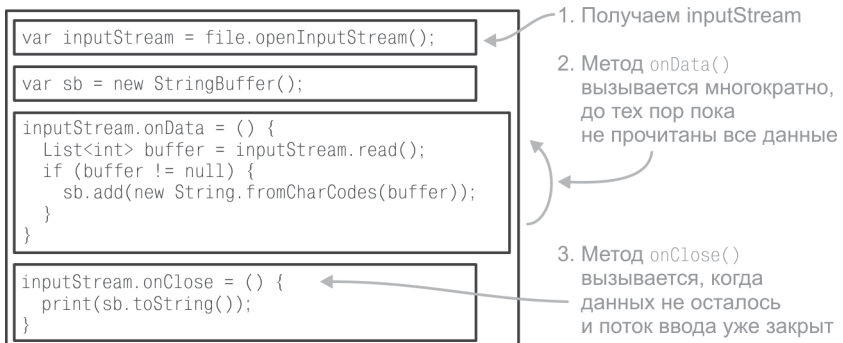


Рис. 13.6. Чтение данных из потока `InputStream`, связанного с файлом

После того как все данные прочитаны, вызывается метод `onClosed()`, с помощью которого мы выводим все полученные данные на консоль. Для хранения строк, создаваемых при каждом вызове `onData()`, используется объект типа `StringBuffer`. Внутренний список строк, хранящийся в объекте `StringBuffer`, эффективно преобразуется в одну строку при обращении к его методу `toString()`. В листинге ниже приведен код функции `outputFile()`, в котором используется объект `InputStream`, связанный с файлом, а также – для полноты картины – написанный ранее код функции `listDir()`.

Листинг 13.3. Реализация `outputFile()` с применением объекта `InputStream`

//...остальные методы опущены...

```
outputFile(String filePath) {
    File file = new File(filePath); ← Создаем объект типа File
    file.exists().then((exists) {
        if (exists) {
            Получаем InputStream
            InputStream inputStream = file.openInputStream(); ← для файла
            StringBuffer sb = new StringBuffer(); ← Создаем объект StringBuffer для хранения
                                                    прочитанных из файла данных

            inputStream.onData = () {
                List<int> data = inputStream.read(); ← В обработчике onData читаем данные
                                                    из InputStream

                if (data != null) {
                    sb.add(new String.fromCharCodes(data)); ← Создаем строку из List<int>
                }
            };

            inputStream.onClosed = () {
                print(sb.toString()); ← После закрытия InputStream печатаем строку из буфера
            };
        }
    });
}

listDir(String folderPath) {
    var directory = new Directory(folderPath);
    directory.exists().then((bool exists) {
        DirectoryLister lister = directory.list();

        lister.onDir = (dirPath) {
            print("<DIR> $dirPath");
        };
    });
}
```

```

};

lister.onFile = (filePath) {
  print("<FILE> $filePath");
};

lister.onDone = (completed) {
  print("Finished");
};
});
}

```

Использование класса *OutputStream* для вывода данных

Для записи данных в файл можно использовать тип *OutputStream*. Он получает данные из объекта *File* и предоставляет методы *write(List<int> buffer)* и *writeString(String string)* для неблокирующей записи в файловую систему. После того как все данные записаны (посредством одного или нескольких вызовов *write()*), следует вызвать метод *close()*, чтобы показать, что данные отправлены по месту назначения.

В следующем фрагменте показано, как создать новый файл и записать в него данные, хотя в обозревателе файлов Dart это и не нужно:

```

new File("example.txt").create().then((File newFile) { ← Создаем файл example.txt
  OutputStream stream = newFile.openOutputStream(); ← Получаем объект OutputStream
  stream.writeString("test"); ← Записываем текст          Получаем еще текст
  List<int> buffer = "some more data".charCodes(); ← в виде списка List<int>
  stream.write(buffer); ← Записываем текст из буфера
  str.close(); ← Закрываем входной поток
});

```

Хотя операционная система, стоящая за потоком вывода, принимает данные так быстро, как может, при необходимости осуществляется внутренняя буферизация. Методы *write()* возвращают булевское значение, показывающее, имела место буферизация или нет.

Типы *File* и *Directory* предоставляют все стандартные механизмы для эффективного неблокирующего взаимодействия с файловой системой с помощью потоков ввода и вывода. В этом разделе мы рассмотрели довольно много вопросов: от получения аргументов

командной строки до доступа к файлам и папкам с помощью неблокирующего асинхронного ввода-вывода.

В следующем разделе мы начнем взаимодействовать с браузером по протоколу HTTP – принимать и отправлять данные. Для этого тоже потребуются потоки ввода и вывода. Мы покажем также, как отправлять данные непосредственно из `InputStream` в `OutputStream`; эта техника называется конвейером.

Памятка

- Dart-скрипт может получить аргументы командной строки с помощью объекта `Options`.
 - Библиотеку `dart:io` можно использовать только в серверных приложениях, а библиотека `dart:html` доступна лишь в браузере.
 - Методы с именами, оканчивающимися суффиксом `sync`, блокируют выполнение программы, тогда как асинхронные методы не блокируют выполнение и возвращают будущие значения.
 - Потоки ввода и вывода являются неблокирующими и предоставляют механизм буферизации для чтения и записи данных.
-

13.2. Обслуживание HTTP-запросов от браузера

Обозреватель файлов Dart – замечательное командное приложение, которое умеет выводить на консоль информацию о файлах и папках, но еще лучше он смотрелся бы в виде веб-приложения, которое отправляет ту же информацию веб-браузерам и прочим HTTP-клиентам. В следующем разделе мы напишем на Dart REST-совместимый API для отправки информации о файлах и папках браузеру.

REST-совместимый API

Акроним REST означает Representational State Transfer (передача представимых состояний). Это один из подходов к построению веб-служб, в котором HTTP-глаголы, например GET и POST, описывают метод запроса, а URL-адреса представляют сами передаваемые ресурсы.

Например, GET-запрос по адресу `http://localhost:8080/folderList/c:/dart` вполне может рассматриваться как API для получения списка папок. Список папок – это в данном случае передаваемый ресурс. На стороне сервера в ответ на этот запрос могла бы быть вызвана функция

```
getFolderListAsJson("c:/dart")
```

которая возвращает список папок в формате JSON.

Архитектурный стиль REST описан на многих сайтах, рекомендуем обратиться к ним, если вы незнакомы даже с основами.

Нам предстоит изменить Dart-скрипт, так чтобы он не завершался после выполнения запроса, а продолжал работать до тех пор, пока его не снимут принудительно. Связано это с тем, что тип `HttpServer` из библиотеки `dart:io` снова привносит в программу цикл обработки событий, когда начинает прослушивать HTTP-запросы.

Мы составим новое клиент-серверное приложение «Обозреватель файлов Dart» из двух. Во-первых, мы напишем серверное приложение, которое решает двоякую задачу: возвращает файлы, составляющие клиентскую часть, и отвечает на запросы клиента. Во-вторых, мы напишем клиентское приложение, которое, будучи загружено, посылает серверу запросы на получение списков файлов и папок. Поток запросов и ответов показан на рис. 13.7.

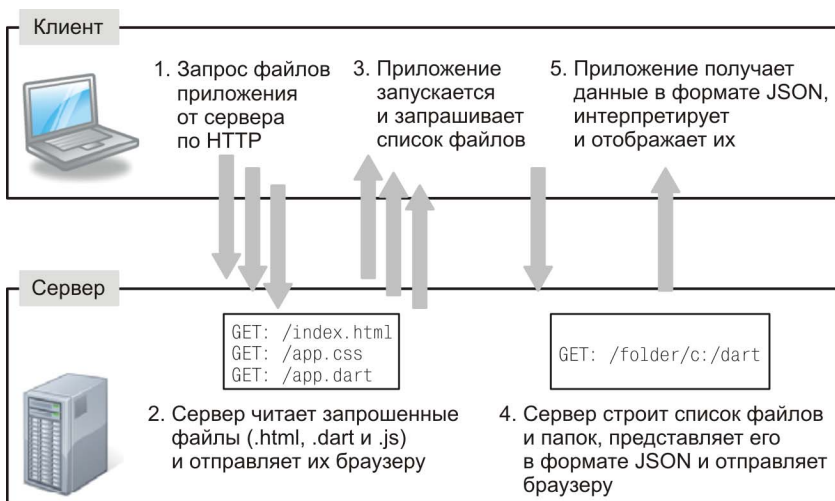


Рис. 13.7. Поток запросов между клиентской и серверной частями приложения

Именно здесь неблокирующий ввод-вывод оказывается по-настоящему полезным. Благодаря асинхронным API серверная часть приложения не блокируется, и управление возвращается в цикл обработки событий так быстро, как это возможно, в результате чего сервер способен обслуживать новые запросы, хотя первый еще не завершился.

13.2.1. Класс *HttpServer*

В библиотеке `dart:io` есть три основных класса, которые необходимы для обслуживания HTTP-запросов: `HttpServer` предоставляет обработчики, срабатывающие, когда приходит запрос; `HttpRequest` представляет входящие данные, а `HttpResponse` — исходящие. В табл. 13.2 перечислены наиболее употребительные свойства и методы этих классов.

Таблица 13.2. Наиболее употребительные свойства и методы основных классов, обслуживающих HTTP-запросы

Класс	Метод или свойство	Описание
HttpServer		Используется для обслуживания HTTP-запросов
	<code>addRequest-Handler (matcher, handler)</code>	Принимает две функции: сопоставитель с сигнатурой <code>bool matcher(HttpRequest req)</code> и обработчик с сигнатурой <code>void handler(HttpRequest req, HttpResponse res)</code> Если сопоставитель возвращает <code>true</code> , то для обработки запроса вызывается обработчик
	<code>defaultRequest-Handler (handler)</code>	Принимает функцию-обработчик, которая вызывается, если никакой другой обработчик не найден. Если обработчик по умолчанию не задан, то сервер возвращает ответ с кодом состояния 404 (Страница не найдена)
	<code>listen(String host, int port)</code>	Начинает прослушивание указанного порта для указанного узла
HttpRequest		Содержит данные, пришедшие от браузера
	<code>String method</code>	Метод запроса, например: GET, POST, PUT
	<code>String path</code>	Путевая часть URI-адреса, например <code>/file/C:/dart</code> в запросе <code>http://localhost/file/C:/dart</code>
	<code>String queryString</code>	Строка запроса в URI-адресе, например <code>key=value</code> в запросе <code>http://localhost/file?key=value</code>
	<code>Map query-Parameters</code>	Словарь, содержащий пары ключ–значение из строки запроса
	<code>InputStream inputStream</code>	Входящие данные, отправленные в составе запроса
HttpResponse		Содержит данные, отправленные браузеру
	<code>HttpHeaders headers</code>	Позволяет задать заголовки ответа, например тип содержимого
	<code>OutputStream outputStream</code>	Используется для отправки данных браузеру

Класс `HttpServer` получает запрос и пытается сопоставить его с обработчиком, вызывая по очереди все сконфигурированные сопоставители, пока какой-то из них не вернет обработчик. Если этого не произойдет, то вызывается обработчик по умолчанию. Можете проверить этот механизм на небольшом примере в листинге 13.4, который обрабатывает запрос `GET: /echo/` и возвращает обратно метод и путь. В ответ на любой другой запрос возвращается строка «Hello World». На рис. 13.8 показаны ответы на запросы к двум разным URL.

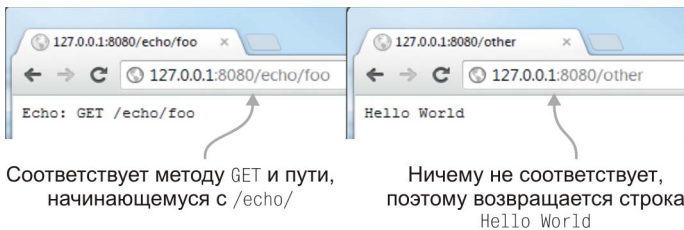


Рис. 13.8. Ожидаемый ответ сервера

В листинге 13.4 приведен полный код сервера, возвращающего такие данные. После вызова метода `listen()` объекта `HttpServer` управление получает цикл обработки событий, который вызывает сопоставители и обработчики при получении запроса на обслуживаемый узел и порт. В примерах ниже используется порт 8080, но вы можете заменить его любым другим, если на вашей машине этот порт занят.

Листинг 13.4. Обслуживание простых HTTP-запросов Dart-сервером

```
import "dart:io";

main() {
  HttpServer server = new HttpServer(); ← Создаем экземпляр сервера

  server.addHandler(
    (HttpRequest req) => req.path.startsWith("/echo/"), ← Этот сопоставитель
    (HttpRequest req, HttpResponse res) { ← возвращает true, если путь
      var method = req.method; ← в запросе начинается с /echo/
      var path = req.path;
      res.outputStream.writeString("Echo: $method $path"); ← Обработчик: возвращает простую строку
      res.outputStream.close(); ← Закрываем поток вывода, закончив посылать данные
    }
  );
}
```



```
server.defaultRequestHandler = (HttpRequest req, HttpResponse res) {  
  res.outputStream.writeString("Hello World");  
  res.outputStream.close();  
};
```

Обработчик по умолчанию:
вызывается, если никаких
других обработчиков не найдено

```
server.listen("127.0.0.1", 8080);  
print("Listening...");  
}
```

← Сервер начинает прослушивать указанный порт
на указанном IP-адресе

Поняв, как отвечать на HTTP-запросы браузера, посмотрим, как воспользоваться этой техникой для отправки браузеру статических файлов.

13.2.2. Передача статических файлов по HTTP

Первое, что должна сделать серверная часть приложения «Обозреватель файлов Dart», – отправить браузеру с помощью объекта `HttpServer` файлы, составляющие клиентскую часть. Сюда входят файлы с расширениями `.html`, `.css` и `.dart` или `.js`. В предыдущем разделе мы научились читать файлы из потока ввода, а объекту `HttpResponse` необходим поток вывода для отправки данных браузеру. По счастью, Dart предоставляет функцию `pipe()`, которая отправляет данные напрямую из `InputStream` в `OutputStream` и закрывает поток вывода, когда все данные переданы. Этим механизмом мы и воспользуемся для отправки файлов браузеру.

Клиентскую часть приложения составляют следующие файлы, хранящиеся на сервере в подпапке `client`:

```
c:\DirectoryServer  
  \ServerApp.dart  
  \client  
    \index.html  
    \ClientApp.css  
    \ClientApp.dart  
    \ClientApp.dart.js
```

Чтобы отправить именно те файлы, которые запросил браузер, нам понадобится свойство `HttpRequest.path`. Мы напишем класс, содержащий сопоставитель и обработчик, которые будут обслуживать запросы на статические файлы, то есть файлы с ожидаемыми расширениями имени: `.html`, `.dart`, `.css` и `.js`. В листинге 13.5 приведен код

класса `StaticFileHandler` и показано, как он используется совместно с `HttpServer`. Сопоставитель возвращает `true`, если запрашивается файл одного из вышеперечисленных типов. Обработчик получает запрошенное имя файла, дописывает в его начало имя подпапки `./client` и соединяет конвейером поток ввода из файла `InputStream` с потоком вывода в HTTP-ответ `OutputStream`. Обратите внимание, что поток вывода не закрывается методом `close` — это автоматически делает метод `pipe()`.

Листинг 13.5. Обслуживание запросов на статические файлы

```
import "dart:io";

class StaticFileHandler {
  bool matcher(HttpRequest req) {
    return req.path.endsWith(".html") ||
           req.path.endsWith(".dart") ||
           req.path.endsWith(".css") ||
           req.path.endsWith(".js");
  }

  void handler(HttpRequest req, HttpResponse res) {
    var requestedFile = "./client${req.path}";
    File file = new File(requestedFile);
    file.openInputStream().pipe(res.getOutputStream());
  }
}

main() {
  var server = new HttpServer();

  var staticFiles = new StaticFileHandler();

  server.addRequestHandler(staticFiles.matcher, staticFiles.handler);

  server.listen("127.0.0.1", 8080);
  print("Listening...");
}
```

Сопоставитель: возвращает `true`, если путь завершается одним из ожидаемых расширений имени файла

Обработчик вызывается, ← если его нашел сопоставитель

← Получаем из запроса имя файла относительно подпапки `./client`

Соединяем конвейером поток ввода из файла с потоком вывода в HTTP-ответ

Создаем экземпляр ← обработчика статических файлов

← Передаем сопоставитель и обработчик в качестве аргументов

Это тривиальный пример, любой реальный сервер должен еще обеспечивать как минимум обработку ошибок и безопасность. Тот сервер, что мы написали, аварийно завершится, если запрошен несуществующий файл (необходимо добавить проверку `file.exists()`).

А если задать запрос вида `http://localhost:8080/../../someOtherFolder/someFile.txt`, то можно будет получить доступ к любому файлу на серверной машине.

Ну а теперь, научившись отправлять серверу файлы, составляющие клиентскую часть приложения, построим REST API для отправки списка папок и файлов в формате JSON.

Памятка

- Класс `HttpServer` прослушивает HTTP-запросы к указанному узлу и порту.
 - Для обработки HTTP-запроса необходимы две функции: сопоставитель и обработчик. Они добавляются в объект `HttpServer` методом `addRequestHandler()`.
 - Если никаких других обработчиков запроса не найдено, вызывается обработчик по умолчанию `defaultRequestHandler()`.
-

13.3. REST API для клиентов

Теперь мы знаем достаточно, чтобы объединить код HTTP-сервера с кодом получения содержимого каталога, который рассматривался выше в этой главе. Мы хотим принять GET-запрос с указанием каталога на нашем диске и в ответ отправить список файлов и папок в этом каталоге. Для построения двух списков мы воспользуемся методами `onDir()` и `onFile()` объекта `DirectoryLister`, а затем объединим их с помощью библиотеки `dart:json`. Эта библиотека – прекрасный пример использования одного и того же кода на стороне клиента и сервера.

Мы также хотим посылать браузеру содержимое указанного файла, но на этот раз мы отправим его не «как есть», а обернем форматом JSON. Таким образом, мы получаем API из двух вызовов, который должно поддерживать приложение (рис. 13.9).

Чтобы решить поставленную задачу, нам понадобятся три обработчика запросов: один будет обслуживать запросы на статические файлы, составляющие клиентскую часть приложения, второй – возвращать список файлов и папок, а третий – содержимое файла. Класс `StaticFileHandler` мы уже написали, а два других класса – `FolderListHandler` и `FileContentHandler` – приведены на рис. 13.10 вместе с пояснениями.

Функция `main()`, в которой все три класса используются, приведена в листинге 13.6. Там же показана вспомогательная функция `addHeaders()`. Она добавляет в ответ два важных заголовка, позво-

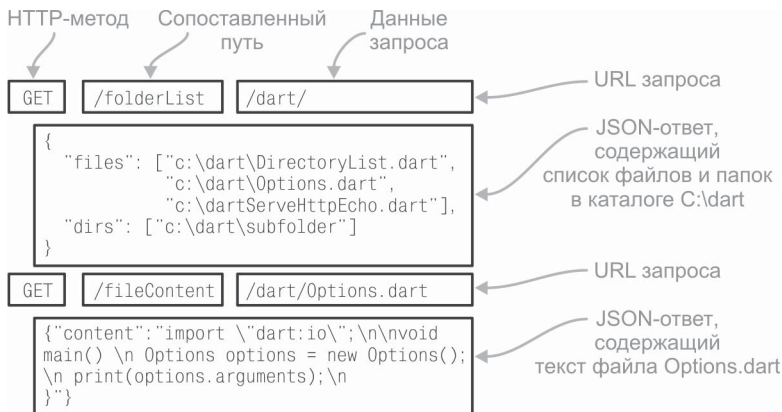


Рис. 13.9. API из двух вызовов для получения содержимого каталога и содержимого файла в формате JSON

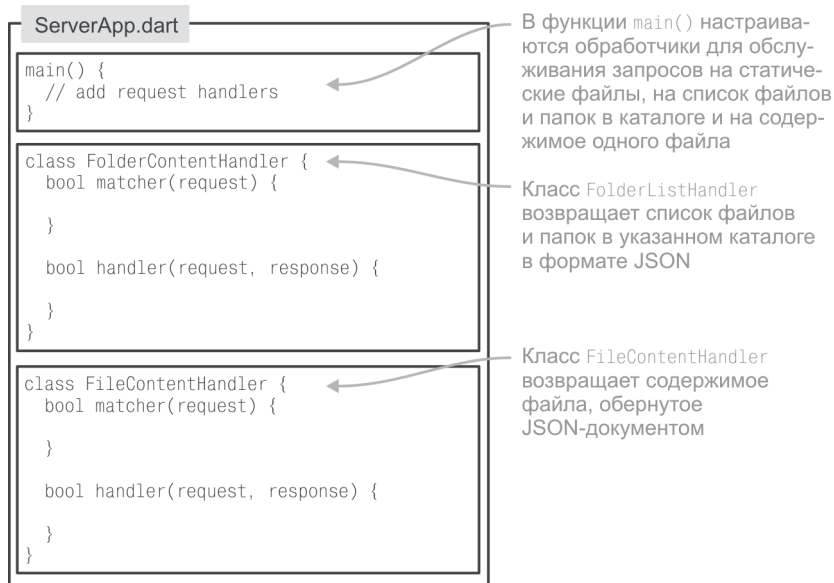


Рис. 13.10. Общая структура серверной части приложения

ляющих использовать встроенный в редактор Dart отладчик при конструировании пользовательского интерфейса. Браузеры обычно запрещают отправлять AJAX-запросы серверу, отличному от того, который прислал страницу. Эти два заголовка говорят браузеру, что в данном случае допустимо принять скрипт с другого URL-адреса. Дополнительные сведения о междоменном разделении ресурсов (cross-origin resource sharing – CORS) ищите в сети.

Листинг 13.6. Модифицированная функция `main()`, содержащая все три класса обработчиков

```
import "dart:io";
import "dart:json"; ← Импортируется библиотека dart:json на стороне сервера

main() {
  HttpServer server = new HttpServer();

  var staticFile = new StaticFileHandler();
  var folderList = new FolderListHandler();
  var fileContent = new FileContentHandler();

  server.addRequestHandler(staticFile.matcher, staticFile.handler);
  server.addRequestHandler(folderList.matcher, folderList.handler);
  server.addRequestHandler(fileContent.matcher, fileContent.handler);

  server.listen("127.0.0.1", 8080);
  print("Listening... on 127.0.0.1:8080");
}

addHeaders(res) {
  res.headers.add("Access-Control-Allow-Origin", "http://localhost/*");
  res.headers.add("Access-Control-Allow-Credentials", true);
}
```

Создаем экземпляры обработчиков

Начинаем прослушивать запросы

Присоединяем обработчики к серверу

Вспомогательная функция: отменяет блокировку AJAX-запросов браузером

Функция `main()` не представляет собой ничего сложного. Дополнительные обработчики – значит, дополнительные классы. Пока что все наши классы находятся в одном и том же файле, но в большом проекте их надо было бы разнести по нескольким библиотекам.

Далее мы рассмотрим класс `FolderListHandler`, отвечающий за отправку списка файлов и папок в формате JSON.

13.3.1. Отправка содержимого каталога в формате JSON

Класс `FolderListHandler` включает два метода: `matcher()` и `handler()`. Метод `matcher()` возвращает `true`, если получен запрос типа `GET`, в котором путь начинается строкой `/folderList`. Если сопоставитель `matcher()` возвращает `true`, то вызывается ассоциированный с ним метод `handler()`, в котором можно извлечь из пути запроса данные. Данные идут после префикса `/folderList`, так что мы воспользуемся библиотечной функцией `subString()`, чтобы удалить часть `/folderList`, оставив только имя интересующего нас каталога.

Получив имя каталога, мы можем прибегнуть к рассмотренным выше классам `Directory` и `DirectoryLister`. Имена файлов и каталогов распределяются по двум спискам, а в методе `onDone()` объекта `DirectoryLister` мы поместим эти списки в словарь, а словарь преобразуем в JSON-строку. Нас интересуют только файлы с расширением `.dart`, поэтому все остальные мы пропускаем.

Наконец, сформированная JSON-строка передается в поток вывода объекта `HttpResponse`, после чего этот поток закрывается. Весь код приведен в листинге 13.7.

Листинг 13.7. Реализация класса `FolderListHandler`

... `main()` и другие функции опущены ...

```
class FolderListHandler {
  bool matcher(HttpRequest req) {
    return req.path.startsWith("/folderList")
      && req.method == "GET";
  }

  void handler(HttpRequest req, HttpResponse res) {
    addHeaders(res);

    var folder = req.path.substring('/folderList'.length);
    DirectoryLister lister = new Directory(folder).list();

    List<String> dirList = new List<String>();
    List<String> fileList = new List<String>();

    lister.onDir = (dirName) {
      dirList.add(dirName);
    };
  }
}
```

Возвращает `true`, если метод и путь подходят

← Обработчик: добавляются заголовки CORS

Получаем объект `DirectoryLister` ← для запрошенной папки

Создаем пустые списки, в которые позже поместим имена файлов и папок

Обрабатываются имена папок

```

lister.onFile = (fileName) {
    if (file.endsWith(".dart")) {
        fileList.add(fileName);
    }
};

lister.onDone = (done) { ← После того как имена всех файлов и папок получены...
    var resultMap = new Map<String,List>();
    resultMap["files"] = fileList;           ...преобразуем словарь,
    resultMap["dirs"] = dirList;             содержащий два списка,
    var jsonString = JSON.stringify(resultMap); ← в JSON-строку

    res.outputStream.writeString(jsonString);
    res.outputStream.close();               Отправляем строку браузеру
                                           и закрываем поток
};
}
}

```

Если сейчас запустить сервер и перейти в браузере по адресу <http://localhost:8080/folderList/dart>, то сервер вернет имена файлов и папок в каталоге `c:\dart`, представленные в формате JSON (рис. 13.11).

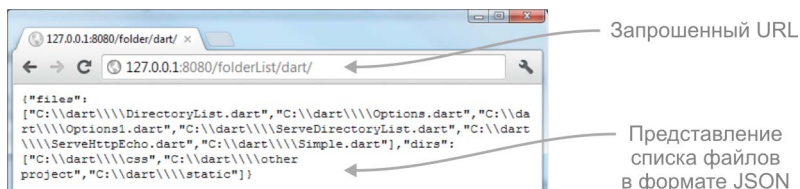


Рис. 13.11. Возврат сервером данных в формате JSON

13.3.2. Отправка содержимого файла в формате JSON

Для отправки содержимого файлы мы используем класс `FileContentHandler`, который умеет обрабатывать запросы, начинающиеся с `/fileContent`, извлекаем следующее за префиксом имя файла, читаем его содержимое и возвращаем прочитанный текст, обернув его предварительно JSON-строкой.

Запрос к URL <http://localhost:8080/fileContent/dart/Options.dart> вернет содержимое файла `Options.dart` в формате JSON. Код класса `FileContentHandler` приведен ниже.

Листинг 13.8. Реализация класса FileContentHandler

... main() и другие функции опущены...

```
class FileContentHandler {
  bool matcher(HttpRequest req) {
    return req.path.startsWith("/fileContent") | Возвращаем true, если метод
      && req.method == "GET"; | и путь подходят
  }

  void handler(HttpRequest req, HttpResponse res) {
    addHeaders(res);

    var filename = req.path.substring("/file".length); ← Извлекаем
                                                         имя файла из пути

    File file = new File(filename); ← Создаем объект, представляющий файл

    file.readAsString().then((String fileContent) {
      var result = new Map<String,String>();
      result["content"] = fileContent; | Читаем из файла текст
                                       и записываем его в словарь
                                       с ключом content

      res.outputStream.writeString(JSON.stringify(result)); | Преобразуем словарь
      res.outputStream.close(); | в формат JSON
    }); | и отправляем браузеру
  }
}
```

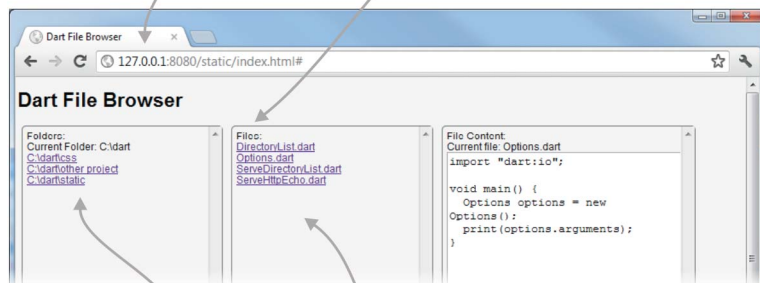
Вот мы и закончили серверную часть приложения. Сервер умеет отвечать на запросы о содержимом каталога и возвращать текст файла. Теперь добавим на стороне клиента простой пользовательский интерфейс, с помощью которого будем взаимодействовать с сервером.

13.3.3. Добавление пользовательского интерфейса на стороне клиента

Клиентская часть обозревателя файлов Dart загружается в браузер и далее отправляет асинхронные AJAX-запросы к REST API сервера. Получив от сервера данные, браузер обновляет пользовательский интерфейс. Основные действия показаны на рис. 13.12.

Поскольку клиентская часть — одностраничное веб-приложение, в котором нет переходов между физическими страницами, то управ-

1. GET-запрос для получения файлов с расширениями .html, .dart и .css
2. Запрос GET /folderList для получения списка файлов и папок



3. Щелчок по имени папки приводит к отправке нового запроса GET /folderList серверу
4. Щелчок по имени .dart-файла приводит к отправке запроса GET /fileContent с последующим отображением текста файла

Рис. 13.12. Основные действия клиентской части приложения: запрос данных у сервера и отрисовка их в пользовательском интерфейсе

лять историей навигации должны мы сами с помощью функции `pushState()` и события `popState` (см. главу 11). В листинге 13.8 приведен полный код клиентской части приложения. Функция `navigate()` отвечает за добавление записи в историю навигации путем обращения к функции `pushState()`, а обработчик события `popState`, объявленный в `main()`, — за реакцию на нажатие встроенной в браузер кнопки **Назад**. Все это мы уже видели ранее в примере `DartExpense`, но теперь статические файлы возвращает серверная часть, написанная тоже на Dart, и мы обращаемся к REST API сервера, отправляя ему GET-запросы. Общая структура клиентской части приложения изображена на рис. 13.13.

В клиентском коде (см. листинг 13.9) имеются также функции `loadFolderList()` и `loadFileContent()`, которые отправляют GET-запросы серверу и обновляют пользовательский интерфейс по получении ответа. Функция `loadFolderList()` вызывается для перехода в другую папку, а функция `loadFileContent()` — для загрузки содержимого файла, после того как пользователь щелкнет по его имени.

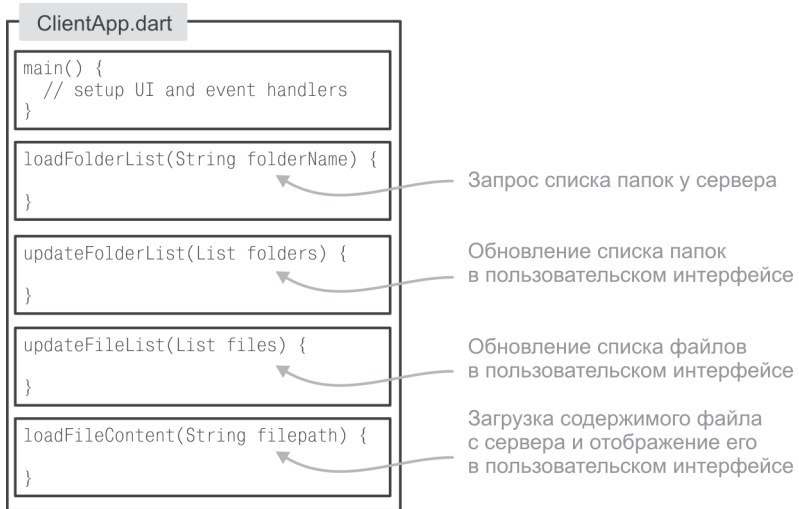


Рис. 13.13. Общая структура клиентской части приложения

Листинг 13.9. Клиентская часть приложения «Обозреватель файлов Dart»

```
import "dart:html";
import "dart:json";

main() {
  window.on.popState.add((data) => loadFolderList(data.state));
  navigate("/");
}

navigate(String folderName) {
  loadFolderList(folderName);
  window.history.pushState(folderName, folderName, "#$folderName");
}

loadFolderList(String folderName) {
  document.querySelector("#currentFolder").innerHTML = "Current: $folderName";
  var url = "http://127.0.0.1:8080/folder$folderName";

  new HttpRequest.get(url, (response) {
    var jsonData = response.responseText;
    Map result = JSON.parse(jsonData);
    updateFolderList(result["dirs"]);
    updateFileList(result["files"]);
  })
}
```

Присоединяем обработчик события `popState` для навигации по кнопке **Назад**

← Переходим в корневую папку

← Загружаем запрошенную папку

Добавляем запрос папки в историю навигации

Обновляем ГИП и строим URL запроса

Загружаем с сервера список файлов и папок

Обновляем ГИП на основе нового списка файлов и папок

```

});
}

updateFolderList(List folders) {
    var content = document.query("#folderList");
    content.children.clear();

    for(String dirName in folders) {
        var link = new Element.html("<div><a href='#'>$dirName</a></div>");
        link.onClick.add((e) => navigate(dirName));
        content.children.add(link);
    }
}

updateFileList(List files) {
    var content = document.query("#fileList");
    content.children.clear();

    for(String filepath in files) {
        var filename = filepath.substring(filepath.lastIndexOf('\\')+1);
        var link = new Element.html("<div><a href='#'>$filename</a></div>");
        link.onClick.add((e) => loadFileContent(filepath));
        content.children.add(link);
    }
}

loadFileContent(String filepath) {
    var filename = filepath.substring(filepath.lastIndexOf('\\')+1);
    document.query("#filename").innerHTML = "Current: $filename";
    var url = "http://127.0.0.1:8080/file$filepath";

    new HttpRequest.get(url, (response) {
        var contentText = JSON.parse(response.responseText)["content"];
        var content = document.query("#fileContent");
        content.innerHTML = contentText;
    });
}

```

Находим <div> folderList
и помещаем в него папки

При щелчке по имени папки
вызываем navigate()

Находим <div> fileList
и помещаем в него файлы

При щелчке по имени
файла вызываем
loadFileContent()

Извлекаем имя файла
и обновляем ГИП

Получаем содержимое
запрошенного файла...

...и отображаем его в текстовой
области fileContent

Для полноты в листинге 13.10 приведена HTML-разметка клиентской части. Макет можете оформить с помощью собственных CSS-стилей или взять CSS-файл из исходного кода, прилагаемого к книге. Разметка содержит элементы <div> с идентификаторами folderList, fileList и fileContent. HTML-файл загружается с сервера наряду с файлом ClientApp.dart.

Листинг 13.10. HTML-файл, в котором размещена клиентская часть приложения

```
<!DOCTYPE html>

<html>
  <head>
    <meta charset="utf-8">
    <title>Static</title>
    <link rel="stylesheet" href="directoryList.css">
  </head>
  <body>
    <h2>Dart File Browser</h2>

    <div id="application">
      <div class="content">Folders: <br/>
        <span id="currentFolder">Current Folder:</span>
        <div id="folderList" ></div>
      </div>

      <div class="content">Files: <br/>
        <div id="fileList"></div>
      </div>

      <div class="content">File Content: <br/>
        <span id="filename"></span><br/>
        <textarea id="fileContent" rows="20"></textarea>
      </div>
    </div>

    <script type="application/dart" src="ClientApp.dart"></script>
  </body>
</html>
```

Dart-файл с клиентской частью приложения

<div> для списка папок

<div> для списка файлов

<div> для содержимого файла

Теперь у нас есть вполне работоспособное клиент-серверное приложение, написанное целиком на Dart. Серверная часть возвращает статические файлы и динамические данные, получаемые из файловой системы, а клиентская запрашивает данные, пользуясь REST-интерфейсом сервера, и отображает их в браузере. Сервер может обслуживать одновременно несколько запросов без блокировки.

В качестве упражнения добавьте в клиентское приложение кнопку **Save** и реализуйте отправку содержимого файла назад на сервер с помощью AJAX-запроса методом POST. В ответ сервер может от-

крыть поток вывода в файл `OutputStream` и записать обновленное содержимое на диск. Тем самым мы превратим обозреватель файлов Dart в редактор файлов.

Памятка

- Одностраничное веб-приложение загружается в браузер, а затем запрашивает данные с помощью объекта `HttpRequest`.
 - Класс `HttpServer` предоставляет асинхронные API для эффективного обслуживания запросов от нескольких клиентов.
 - Некоторые библиотеки, в частности `dart:json`, можно использовать как на клиенте, так и на сервере.
-

13.4. Резюме

В этой главе мы разработали полное решение на Dart, имеющее серверную и клиентскую часть. В серверной части используется класс `HttpServer` из библиотеки `dart:io`, а также классы `File` и `Directory`, с помощью которых программа выполняет неблокирующие асинхронные операции доступа к файловой системе.

Мы видели, как серверный скрипт получает доступ к аргументам командной строки и как он читает и записывает файлы с помощью типов `InputStream` и `OutputStream`. Мы также рассмотрели некоторые вспомогательные функции-обертки, например `readAsString()`, которые упрощают работу с потоками.

Мы воспользовались классом `HttpServer` для отправки как статических файлов, так и динамических данных клиенту и научились соединять конвейером поток ввода из файла (`InputStream`) с потоком вывода в `HttpResponse` (`OutputStream`) и таким образом отправлять содержимое файла браузеру. Мы также использовали библиотеку `dart:json` на стороне клиента и сервера.

В следующей главе мы вернемся к приложению `DartExpense` из глав 10 и 11 и дополним его серверной базой данных. Мы будем обслуживать запросы от `DartExpense` с помощью серверного приложения – так же, как в этой главе, – только вместо получения папок и файлов из файловой системы будем взаимодействовать с базой данных. Класс `HttpServer` будет использоваться для передачи JSON-данных между клиентским приложением и базой данных класса `NoSQL` по протоколу HTTP.



Глава 14. Отправка, синхронизация и сохранение данных

В этой главе

- ❑ Организация двусторонней связи между клиентом и сервером с помощью веб-сокетов.
- ❑ Совместное использование кода клиентом и сервером.
- ❑ Использование класса `HttpClient` для связи между серверами.
- ❑ Хранение данных на стороне сервера.

Хотя веб-приложения следует проектировать с учетом работы в автономном режиме, но большинство из них все же должны хранить какие-то данные на сервере – в файловой системе, в базе данных на основе SQL, например MySQL, или в базе данных NoSQL типа CouchDB. В силу самой природы веб пользователи будут подключаться к приложению с разных машин, а это означает, что даже если данные на сервере не хранятся, все равно необходим какой-то серверный компонент, отвечающий за их синхронизацию между машинами.

В этом разделе мы вернемся к приложению DartExpense из третьей части книги и добавим в него синхронизацию через сервер с помощью веб-сокетов – появившейся в HTML5 технологии, которая позволяет организовать двустороннюю связь между клиентом и сервером путем создания постоянного соединения между ними. Добившись того, что клиенты DartExpense, работающие в разных браузерах, могут общаться друг с другом через сервер, мы рассмотрим вопрос о сохранении данных на сервере в масштабируемой базе данных Apache CouchDB, относящейся к классу NoSQL. Мы выбрали именно эту базу, потому что она не нуждается в каких-либо сторонних драйверах или библиотеках, а предоставляет REST API, к которому сервер может обращаться с помощью класса `HttpClient` из серверной библиотеки Dart. Этот класс позволяет обращаться

к любому серверу, раскрывающему API на основе HTTP, хранение данных – далеко не единственное его применение.

Но чтобы приступить к работе, нам понадобится сервер статических файлов, который мы написали в предыдущей главе; он будет передавать клиенту файлы, составляющие приложение DartExpense. На этом фундаменте мы построим поддержку веб-сокетов и хранения данных.

14.1. Передача приложения DartExpense с сервера

В предыдущей главе мы показали, как написать сервер статических файлов, который будет возвращать браузеру файлы, составляющие клиентскую часть приложения. Этим сервером мы воспользуемся и для передачи приложения DartExpense. Для тех, кто забыл, на рис. 14.1 показан поток запросов и данных между браузером и сервером.

Клиентская часть приложения DartExpense находится в подпапке `./client` на сервере, составляющие ее файлы сервер и возвращает по протоколу HTTP. Приложение работает точно так же, как если бы оно было загружено непосредственно из файловой системы; это по-прежнему клиентское приложение, только составляющие его файлы получены от сервера. Имея механизм загрузки приложения, мы можем приступить к организации взаимодействия с сервером посредством веб-сокетов.

14.2. Использование веб-сокетов для организации двусторонней связи

В предыдущей главе мы обсуждали, как использовать REST API для получения от сервера данных о содержимом каталога. У сервера *запрашивали* список файлов и папок в каталоге `c:/dart`, а сервер *отвечал*, отправляя данные в формате JSON. Это стандартный способ взаимодействия через веб: клиент отправляет запросы, а сервер на них отвечает. Соединение между клиентом и сервером существует только в течение времени, необходимого для одной пары запрос–ответ.

Веб-сокеты – это новая модель обмена данными, которая позволяет серверу поддерживать постоянное соединение с клиентом. В этом случае сервер может посылать клиенту данные, не дожидая-

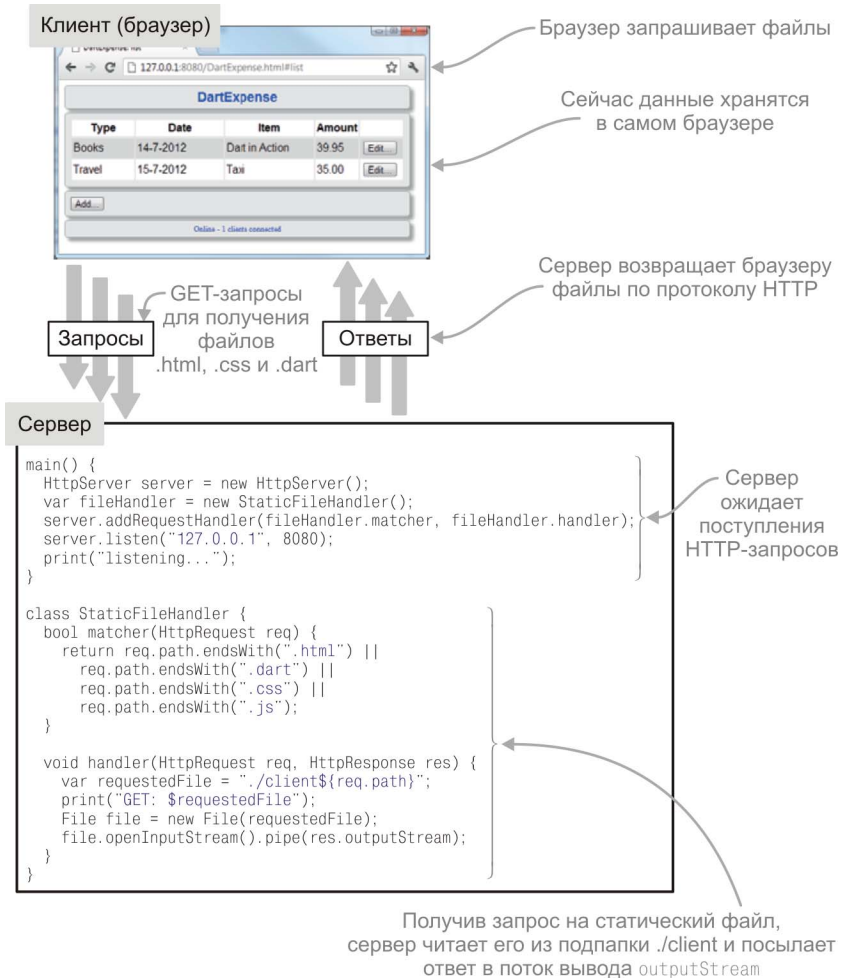


Рис. 14.1. Сервер на базе класса `HttpServer` посылает файлы приложения браузеру

ясь запроса от него. Ниже мы воспользуемся этой возможностью для синхронизации нескольких клиентов `DartExpense`, это позволит изменить запись о расходе в одном браузере и увидеть результат в другом. Но сначала рассмотрим более простой пример: покажем

количество подключенных к серверу клиентов DartExpense. Всякий раз, как подключается новый клиент, сервер отправляет количество подключенных клиентов каждому подключенному браузеру, как показано на рис. 14.2.

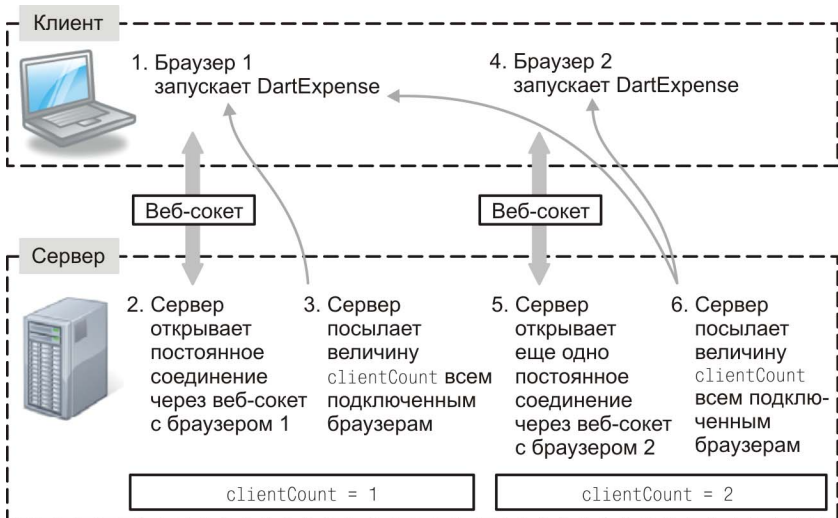


Рис. 14.2. Веб-сокеты позволяют организовать двустороннее соединение между клиентом и сервером, что позволяет серверу отправлять данные, не дожидаясь запроса от клиента

Для использования веб-сокетов необходимо модифицировать обе части приложения: клиентскую и серверную. Клиент должен подключаться к серверу и отвечать на сообщения, отправляемые сервером, с помощью функции обратного вызова. Сервер должен прослушивать запросы на подключение от клиентов, хранить список подключенных клиентов и обрабатывать посылаемые клиентами данные.

Примечание. Веб-сокеты и традиционные GET- и POST-запросы прекрасно уживаются друг с другом. общепринято использовать методы GET и POST для обращения к REST API сервера, а дополнительные возможности обеспечивать за счет веб-сокетов.

14.2.1. Соединение через веб-сокеты на стороне клиента

Клиентское приложение, работающее в браузере, должно сначала открыть соединение через веб-сокеты с сервером. Для этого мы добавим метод `connectToWebsocket()` в класс `AppController`. Этот класс, который был разработан в главе 10, отвечает за конструирование пользовательского интерфейса и подключение обработчиков событий, так что является самым подходящим местом для установления соединения с сервером и организации механизма прослушивания сообщений, поступающих через веб-сокеты. В листинге 14.1 приведена функция `main()` приложения `DartExpense`, которая вызывает `connectToWebsocket()`. Здесь же определена константа `WEBSOCKET_URL`, содержащая URL-адрес для подключения к веб-сокету. В URL указан протокол `ws://`, а не `http://`, чтобы браузер знал, что речь идет о соединении через веб-сокеты.

Листинг 14.1. `DartExpense.dart`: функция `main()` создает соединение через веб-сокеты

```
...предложения import опущены...

final WEBSOCKET_URL = "ws://127.0.0.1:8080/websocket"; ← URL, по которому браузер
                                                         подключается к серверному
                                                         веб-сокету

void main() {
  var uiContainer = document.query("#dartexpense");
  var dataSource = new LocalStorage();

  _app = new AppController(uiContainer, dataSource);

  app.buildUI();
  app.loadFirstView();

  app.connectToWebsocket(); ← Вызов новой функции для подключения к веб-сокету

  window.on.popState.add(onPopState);
}

AppController _app;
AppController get app => _app;
```

Функция `connectToWebsocket()` решает две задачи: создает объект, представляющий соединение с веб-сокетом по адресу `WEBSOCKET_URL`,

и настраивает прослушиватель событий `on.message`, поступающих от сервера. Этот прослушиватель работает, как любой асинхронный прослушиватель событий, например нажатий кнопки, — как пользователь может нажать кнопку в любой момент, так и сервер может в любое время послать данные клиенту. Функция-прослушиватель извлекает из пришедшего от сервера сообщения количество подключенных клиентов и показывает их в нижнем колонтитуле приложения `DartExpense`, обращаясь к функции `refreshFooterStatus()`.

В листинге 14.2 приведен код новых методов `connectToWebsocket()` и `refreshFooterStatus()` класса `AppController`. Сервер отправляет количество подключенных клиентов в JSON-сообщении вида: `{"action": "CLIENT_COUNT_REFRESH", "connectedClients": 3}`. Если поле `action` равно `CLIENT_COUNT_REFRESH`, то необходимо извлечь значение поля `connectedClients` и показать его в нижнем колонтитуле.

Листинг 14.2. Добавление метода `connectToWebsocket()` в класс `AppController`

```
class AppController {
  // ...остальные свойства и методы опущены...

  WebSocket _websocket;      ← Новое свойство для хранения соединения с веб-сокетом
  int _connectedClients = 0; ← Новое свойство для хранения количества подключенных клиентов

  connectToWebsocket() {
    _websocket = new WebSocket(WEB_SOCKET_URL); ← Создаем соединение с веб-сокетом
                                                    по указанному URL
    _websocket.on.message.add( (MessageEvent message) {
      Map data = JSON.parse(message.data);
      if (data["action"] == "CLIENT_COUNT_REFRESH" {
        _connectedClients = data["connectedClients"];
        refreshFooterStatus();
      }
    });
  }

  refreshFooterStatus() {
    var statusText = "$_connectedClients clients connected";
    document.query("#footer").innerHTML = statusText;
  }
}
```

Добавляем функцию обработки события, которая будет реагировать на пришедшие от сервера данные

Обновляем текст в нижнем колонтитуле, содержащий количество подключенных клиентов

В результате создания объекта `WebSocket` открывается двустороннее соединение с сервером. Теперь нужно сделать так, чтобы сервер

реагировал на открытие соединения и посылал новое значение счетчика всем подключенным браузерам.

14.2.2. Обработка подключения через веб-сокеты на сервере

Серверная часть приложения уже умеет обрабатывать HTTP-запросы с помощью класса `HttpServer` из библиотеки `dart:io`. У нас есть обработчик для обслуживания запросов на статические файлы, а теперь нужно добавить обработчик `WebSocketHandler` запросов на соединение через веб-сокеты. Класс `WebSocketHandler` также находится в серверной библиотеке `dart:io` и отвечает за создание объекта `WebSocketConnection`, представляющего соединение с одним браузером. У этого объекта имеется метод `send()`, с помощью которого сервер может отправлять данные клиенту, и метод `onMessage()`, который вызывается, когда приходят данные от клиента. На рис. 14.3 показаны

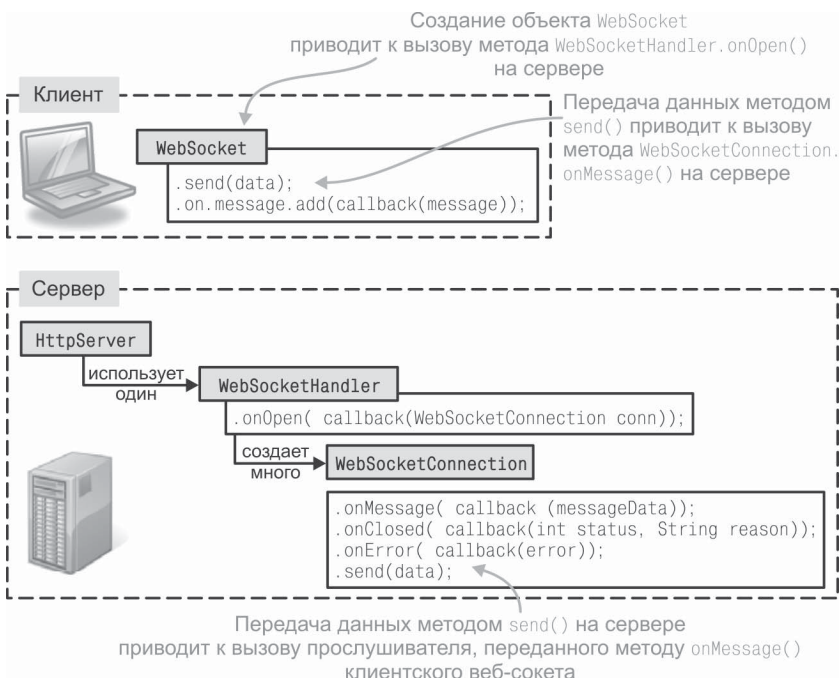


Рис. 14.3. Встроенные классы, применяемые для управления соединениями через веб-сокеты на стороне клиента и сервера

классы на стороне клиента и сервера, участвующие в обмене данными через веб-сокеты.

Обработка запроса через веб-сокеты

Чтобы сервер мог обрабатывать запросы через веб-сокеты, он должен вызвать метод `addRequestHandler()` объекта `HttpServer`, как было показано в предыдущей главе. Мы передаем ему встроенную в класс `WebSocketHandler` функцию `onRequest()`, как показано ниже:

<pre>HttpServer server = new HttpServer(); var websocketHandler = new WebSocketHandler(); var matcher = (req) => req.path == "/websocket"; server.addRequestHandler(matcher, websocketHandler.onRequest); server.listen("127.0.0.1", 8080);</pre>	Добавляем объект <code>WebSocketHandler</code> для обработки запросов через веб-сокеты

Чтобы выполнить какое-то действие в ответ на подключение браузера, мы должны также предоставить реализацию метода `onOpen()` класса `WebSocketHandler`. Этот метод обратного вызова получает новый объект соединения, позволяющий обмениваться данными между сервером и конкретным клиентом.

Учет подключенных браузеров

На сервере создается объект `WebSocketConnection` для каждого подключившегося браузера, и эти объекты нужно где-то хранить. У каждого соединения через веб-сокеты есть уникальный идентификатор (возвращаемый свойством `hashCode`), который позволяет хранить соединения в словаре `Map`, чтобы можно было найти и удалить объект после закрытия соединения. Это понадобится нам позже, когда встанет задача отправить данные для синхронизации всем браузерам, кроме того, который прислал новые данные.

Получив новый объект `WebSocketConnection` от метода `WebSocketHandler.onOpen()`, мы можем послать новое значение счетчика подключений браузеру. Для этого нужно передать параметр типа `String` или `List<int>` методу `send()` каждого соединения. Данные посылаются в виде JSON-строки; мы обходим список соединений и для каждого вызываем метод `send()`, как показано в листинге ниже.

Листинг 14.3. Отправка счетчика клиентов при подключении нового браузера

```
import "dart:io";  
import "dart:json";
```

```

main() {
  HttpServer server = new HttpServer();
  var fileHandler = new StaticFileHandler();
  server.addRequestHandler(fileHandler.matcher, fileHandler.handler);

  var wsHandler = new WebSocketHandler(); ← Создаем новый экземпляр WebSocketHandler

  var matcher = (req) => req.path == "/websocket";
  server.addRequestHandler(matcher, websocketHandler.onRequest); | Добавляем
                                                                    сопоставитель
                                                                    и обработчик
                                                                    в объект HttpServer

  var connections = new Map(); ← Создаем пустой словарь
                               для хранения соединений с клиентами

  var sendUpdatedConnectionCount = () {
    var data = new Map();
    data["action"] = "CLIENT_COUNT_REFRESH"
    data["connectedClients"] = connections.length;
    var message = JSON.stringify(data);

    for(var clientConnection in connections.values) {
      clientConnection.send(message);
    }
  };

  wsHandler.onOpen = (WebSocketConnection conn) { | Сохраняем
                                                    новое соединение
                                                    в словаре
    var key = conn.hashCode();
    connections[key] = conn;

    sendUpdatedConnectionCount(); ← Отправляем обновленный счетчик всем клиентам
  };

  server.listen("127.0.0.1", 8080);
  print("listening...");
}

```

Отслеживание отключений браузеров и ошибок

Помимо добавления в словарь новых соединений и отправки обновленного счетчика всем клиентам, мы должны реализовать еще два метода обратного вызова в классе `WebSocketConnection`: `onMessage()` и `onClosed()`. Пока что реализация `onMessage()` пуста, потому что клиенты еще не посылают серверу никаких данных. Однако обработчик `onClosed()` необходим, так как он удаляет закрытые соединения из словаря и отправляет новое значение счетчика подключений оставшимся клиентам. Метод `onClosed()` вызывается, когда

браузер явно закрывает соединение и в случае ошибки. Параметр `status`, значения которого определены в классе `WebSocketStatus`, показывает, была ли ошибка. Эти обработчики необходимо присоединить в методе `wsHandler.onOpen`, когда еще имеется доступ к объекту `WebSocketConnection` (листинг 14.4).

Листинг 14.4. Добавление обработчика `onClosed()` в объект соединения

```
main() {
  // ...остальной код из предыдущих листингов опущен...

  wsHandler.onOpen = (WebSocketConnection conn) {
    var key = conn.hashCode();
    connections[key] = conn;
    conn.onClosed = (int status, String reason) {
      connections.remove(conn.hashCode()); ← Удаляем это соединение из списка
      sendUpdatedConnectionCount(); ← Посылаем обновленный счетчик всем клиентам
    };

    conn.onMessage = (message) {
      // ничего не делать
    };

    sendUpdatedConnectionCount();
  };

  server.listen("127.0.0.1", 8080);
  print("listening...");
}
```

Теперь сервер может посылать произвольные данные всем подключенным клиентам. Если запустить сервер и постепенно увеличивать число клиентов, то количество подключенных клиентов в нижнем колонтитуле страницы DartExpense будет изменяться (рис. 14.4).

Решив простую задачу, рассмотрим более сложную – синхронизацию данных о расходах между разными браузерами.

14.2.3. Использование веб-сокетов для межбраузерной синхронизации

Теперь у нас есть способ *проталкивания* данных нескольким клиентам, и мы можем применить его для синхронизации клиентов. В примере со счетчиком подключений клиенты не посылали серверу

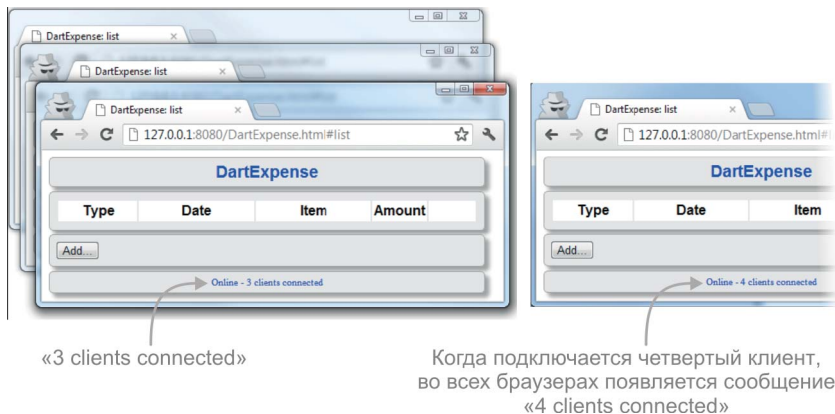


Рис. 14.4. После подключения нового клиента все существующие уведомляются о том, что число клиентов изменилось

данных методом `send()`. Достаточно было просто установить соединение с сервером. Однако никто не мешает воспользоваться методом `send()` для отправки серверу JSON-представления объекта `Expense` через веб-сокеты. Получив эти данные, сервер может разослать их всем остальным браузерам, хотя они ничего и не запрашивали. Это приведет к обновлению данных на экране, как показано на рис. 14.5.

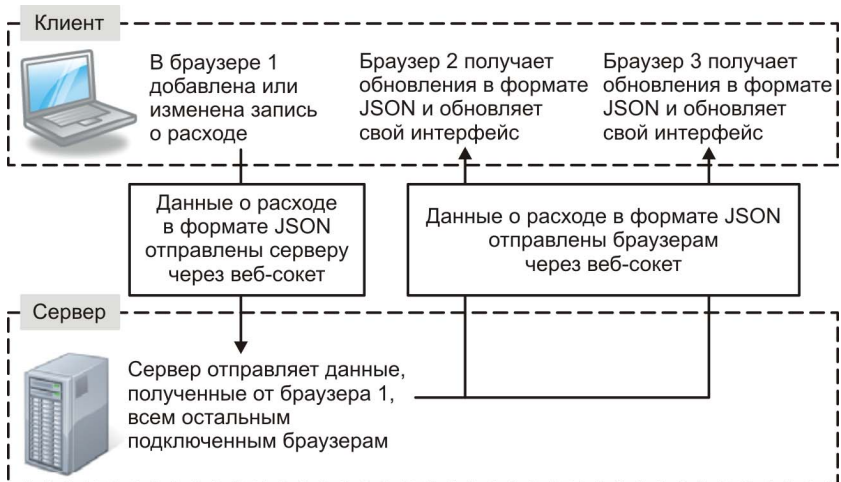


Рис. 14.5. Синхронизация данных во всех браузерах через веб-сокеты

В этом примере механизм синхронизации несложен (см. врезку «Синхронизация в реальных программах»). В текущей версии нажатие кнопки **Save** на странице редактирования записи приводит к ее сохранению в формате JSON в локальном хранилище браузера методом `addOrUpdate(Expense expense)`. Мы модифицируем клиентскую часть `DartExpense`, так чтобы измененный объект `Expense` передавался также функции `sync(Expense expense)`, которая отправляет JSON-данные серверу.

Синхронизация в реальных программах

В реальном мире синхронизация представляет собой более сложную проблему. Например, как быть, если два пользователя редактируют данные одновременно? Типичные решения таковы.

- Последний выигрывает – каждое изменение применяется по мере поступления, поэтому второй пользователь затирает изменения, внесенные первым. Это самый простой для реализации подход, но для пользователя (второго) он не очень привлекателен.
- Первый выигрывает – когда второй пользователь пытается сохранить запись, он получает уведомление о том, что она уже изменена. В результате изменения, внесенные первым пользователем, сохраняются, а второй может принять осмысленное решение.

В случае веб-сокетов тоже есть возможность более разумного поведения, поскольку мы можем отправить клиенту данные, если кто-то другой произвел изменение. Вот примеры возможных стратегий.

- Совместное редактирование – когда один пользователь вводит новые данные, изменения одновременно появляются на экране другого пользователя, и наоборот.
- Запрет редактирования – когда первый пользователь начинает вводить данные, редактируемая запись становится недоступна в интерфейсе второго пользователя. Когда первый пользователь нажимает кнопку **Save**, данные на экране второго пользователя обновляются, и запись становится доступна для редактирования. Это развитие стратегии «первый выигрывает» с улучшенной обратной связью для второго пользователя.

Эти решения сложнее реализовать, поэтому дизайн следует тщательно продумать, зато благодаря удобству работы хорошее приложение становится отличным.

Отправка данных из браузера через веб-сокет

Для отправки данных серверу достаточно просто вызвать метод веб-сокета `send()`, передав ему строку или список `List<int>`. На самом деле это зеркальное отражение серверного метода `send()`, который мы видели чуть выше. Сначала реализуем именно эту часть (рис. 14.6).

В листинге 14.5 показано, что нужно изменить в классе `AppController` для отправки серверу JSON-данных через веб-сокет. Чтобы

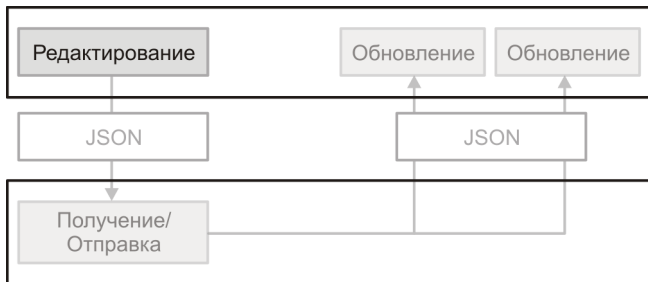


Рис. 14.6. Отправка измененных данных в формате JSON

синхронизировать идентификаторы записей о расходах, мы посылаем также текущее значение статического поля `nextId` класса `Expense`. Чтобы отличить эти данные от сведений о количестве подключенных клиентов, мы ввели новый тип действия `SYNC` в дополнение к `CLIENT_COUNT_REFRESH`.

Листинг 14.5. Отправка серверу данных в формате JSON через веб-сокеты

```

class AppController {
  // ...остальные свойства и методы опущены...

  WebSocket _websocket; ← Существующее соединение через веб-сокеты

  addOrUpdate(Expense expense) {
    _appData.addOrUpdate(expense);
    sync(expense); ← Синхронизируем данные о расходах и сохраняем в локальном хранилище
  }

  sync(Expense expense) {
    Map data = new Map();
    data["action"] = "SYNC";
    data["expense"] = expense.toJson();
    data["nextId"] = Expense.currentNextIdValue;
    var jsonData = JSON.stringify(data); ← Преобразуем словарь в формат JSON

    _websocket.send(jsonData); ← Отправляем данные серверу
  }
}

```

Добавляем запись о расходе и значение следующего идентификатора в словарь

Получение данных от браузера через веб-сокет

Данные отправились в путешествие к другим браузерам. Теперь мы должны реализовать серверную часть (рис. 14.7).

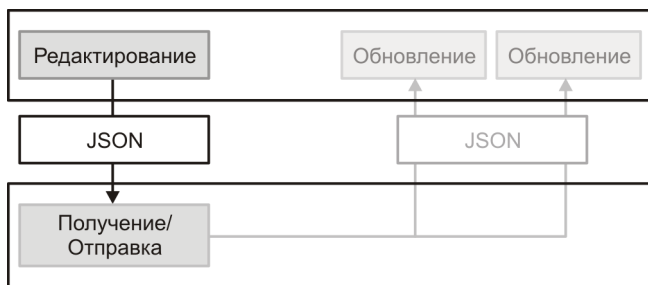


Рис. 14.7. Получение данных о расходе от подключенного клиента

Ранее мы не стали реализовывать метод `onMessage()` класса `WebSocketConnection`. Теперь пора это исправить. Если тип действия равен `SYNC`, то сервер должен переслать полученные данные о расходе всем прочим подключенным браузерам. Текущее соединение (по которому поступили данные) можно идентифицировать по хеш-коду и не отправлять данные тому браузеру, который их прислал. Изменения, которые следует внести в файл `DartExpenseServer.dart`, показаны в следующем листинге.

Листинг 14.6. Отправка серверу данных в формате JSON через веб-сокет

```

main() {
  // ...прочий код из предыдущих листингов опущен...

  wsHandler.onOpen = (WebSocketConnection conn) {
    var key = conn.hashCode();
    connections[key] = conn;

    conn.onClosed = (int status, String reason) {
      //... показанная ранее реализация опущена
    }

    conn.onMessage = (message) {
      var data = JSON.parse(message);
      if (data["action"] == "SYNC") {
        for (var clientConnection in connections.values) {
          // Для каждого соединения
          // ← с клиентом...
        }
      }
    }
  }
}
  
```

```

...если это не текущее
if (clientConnection.hashCode() != conn.hashCode()) { ← соединение...
  clientConnection.send(message); ← ...переслать полученные данные
}
}
}
};

sendUpdatedConnectionCount();
};

server.listen("127.0.0.1", 8080);
print("listening...");
}

```

Получение данных синхронизации в других браузерах

И в заключение снова вернемся к клиентскому приложению и реализуем обновление данных на экране после получения новых данных от сервера (рис. 14.8).

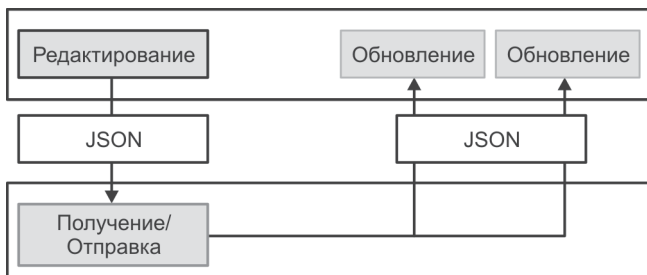


Рис. 14.8. Другие клиенты получают JSON-данные и обновляют свой интерфейс

У нас уже есть обработчик события `on.message`, который реагирует на действие `CLIENT_COUNT_REFRESH`. Нужно добавить еще одну ветвь `if`, в которой обрабатывать действие `SYNC`. Получив такое сообщение, браузер преобразует данные о расходе из формата `JSON` в объект `Expense` и передает его методу `addOrUpdate(Expense expense)`, как если бы этот объект был изменен в самом браузере. Наконец, мы переходим к списковому представлению, заставляя браузер перерисовать список расходов. Код приведен в следующем листинге.

Листинг 14.7. Получение данных синхронизации в браузере

```

class AppController {
    // ...остальные свойства и методы опущены...

    WebSocket _websocket;
    int _connectedClients = 0;

    connectToWebsocket() {
        _websocket = new WebSocket(WEBSOCKET_URL);

        _websocket.on.message.add( (MessageEvent message) {
            Map data = JSON.parse(message.data);
            if (data["action"] == "CLIENT_COUNT_REFRESH" {
                _connectedClients = data["connectedClients"];
                refreshFooterStatus();
            }
            else if (data["action"] == "SYNC") {
                var expenseJSON = data["expense"];
                var expense = new Expense.fromJson(expenseJSON);
                Expense.currentNextIdValue = data["nextId"];
                _appData.addOrUpdate(expense);
                navigate(ViewType.LIST, null);
            }
        });
    }
}

```

Преобразуем данные в формате JSON в объект Expense

← Обновляем значение следующего ID

← Добавляем или обновляем запись о расходе, как при редактировании в интерфейсе

← Переходим в списковое представление, чтобы увидеть результат редактирования

Это был длинный раздел, зато теперь приложение может связать несколько браузеров, используя сервер в качестве посредника. Это полезно для рассылки push-уведомлений типа «поступила новая почта», для синхронизации состояния в многопользовательских браузерных играх и для повышения удобства работы с производственными приложениями.

В следующем разделе мы научимся сохранять полученные от браузера данные синхронизации в NoSQL-базе данных CouchDB.

Памятка

- Сервер может обрабатывать подключения через веб-сокеты, присоединив `WebSocketHandler` к объекту `HttpServer`.
- Чтобы открыть соединение с веб-сокетом сервера, клиент должен создать объект класса `WebSocket`, указав нужный URL-адрес сервера.

- Когда клиент подключается к серверу, объект `WebSocketHandler` создает экземпляр `WebServerConnection`, который позволяет серверу отправлять данные клиенту.
 - Метод `send()` клиентского и серверного объекта соединения служит для отправки данных, которые на клиентской стороне передаются обработчику `on.message`, а на серверной – методу обратного вызова `onMessage()`.
-

14.3. Сохранение данных в базе CouchDB с помощью класса `HttpClient`

В этом разделе мы воспользуемся серверным классом `HttpClient` для сохранения данных в базе Apache CouchDB. CouchDB – база данных класса NoSQL, в которой данные хранятся в формате JSON; в этом смысле она похожа на локальное браузерное хранилище, которое мы рассматривали в третьей части книги.

У CouchDB имеется REST API, к которому клиенты могут обращаться по протоколу HTTP; в роли клиента может, в частности, выступать сервер DartExpense. Даже если вы не планируете использовать CouchDB в собственных программах, примеры ниже познакомят вас с классом `HttpClient`, позволяющим обращаться к различным REST API. Кроме того, у нас будет случай еще раз поработать с типами `InputStream` и `OutputStream`, которые рассматривались в предыдущей главе. `HttpClient` – это серверная версия класса `HttpRequest`, с помощью которого мы запрашивали данные из браузера.

Взаимодействие с другими базами данных

Классы клиента и сервера HTTP в Dart построены на основе низкоуровневого класса `Socket`, который применяется для передачи двоичных данных по протоколу TCP. Тип `Socket` позволяет разработчикам драйверов баз данных общаться непосредственно с сервером базы данных по специальному коммуникационному протоколу прикладного уровня. На момент написания этой книги в разных стадиях готовности находились драйверы с открытым исходным кодом для MySQL, MongoDB, Redis и Riak.

Есть также возможность связать серверный код с машинным кодом с помощью ключевого слова `external`; это позволяет обращаться из Dart к драйверам, написанным на C/C++. В качестве примера можно привести открытый интерфейс с SQLite, но это сложная тема, которая выходит за рамки данной книги.

На данный момент сервер DartExpense получает данные от приложения, работающего в браузере, и синхронизирует данные в раз-

ных браузерах, пользуясь веб-сокетами. Кроме того, каждый клиент хранит данные в локальном хранилище. Проблема в том, что если пользователь запустит приложение в своем браузере и введет новую запись о расходе, то эти данные не будут добавлены в приложение, запущенное в другом браузере позже; синхронизация производится только между клиентами, которые были подключены в момент, когда сохранено изменение. Чтобы исправить ситуацию, приложение должно запрашивать данные у сервера при каждом запуске. Чтобы поддержать хранение данных на сервере, мы должны дополнить приложение следующими возможностями:

- ☐ загрузка всех существующих записей с сервера;
- ☐ сохранение новых и измененных записей на сервере.

Сейчас клиент отправляет данные в формате JSON через веб-сокет, задавая действие SYNC или CLIENT_COUNT_REFRESH. Мы добавим еще действие LOAD, в котором будем запрашивать начальные данные с сервера, и повторно используем действие SYNC, чтобы отправлять данные не только другим браузерам, но и в базу данных. Поток данных после этих изменений показан на рис. 14.9.

Добавление еще одного слоя в архитектуру приложения сопряжено с некоторыми трудностями. По существу, мы загружаем (LOAD) данные о расходах из базы и синхронизируем (SYNC) данные с другими браузерами и с базой. Но прежде чем приступить к реализации, дадим краткий обзор механизмов взаимодействия с CouchDB.

14.3.1. Краткое введение в CouchDB

База данных Apache CouchDB относится к классу NoSQL. Данные в ней хранятся в формате JSON, а доступ осуществляется через REST API. На сайте CouchDB имеются программы установки, никакой настройки, кроме выполнения установщика и последующего запуска сервера, не требуется (в системах Windows в меню Пуск добавляется пакетный файл для запуска сервера).

Если сервер запущен, то для доступа к данным – чтения и сохранения – можно использовать HTTP-методы GET, PUT и POST. Имеется также веб-приложение Futon, встроенное в сервер, которое позволяет манипулировать данными визуально. На рис. 14.10 показано, как выглядит запись о расходе с ID=1 в Futon и в браузере после GET-запроса к серверу CouchDB.

В серверном коде мы будем использовать HTTP API CouchDB, в котором доступ к записям в базе осуществляется с помощью раз-

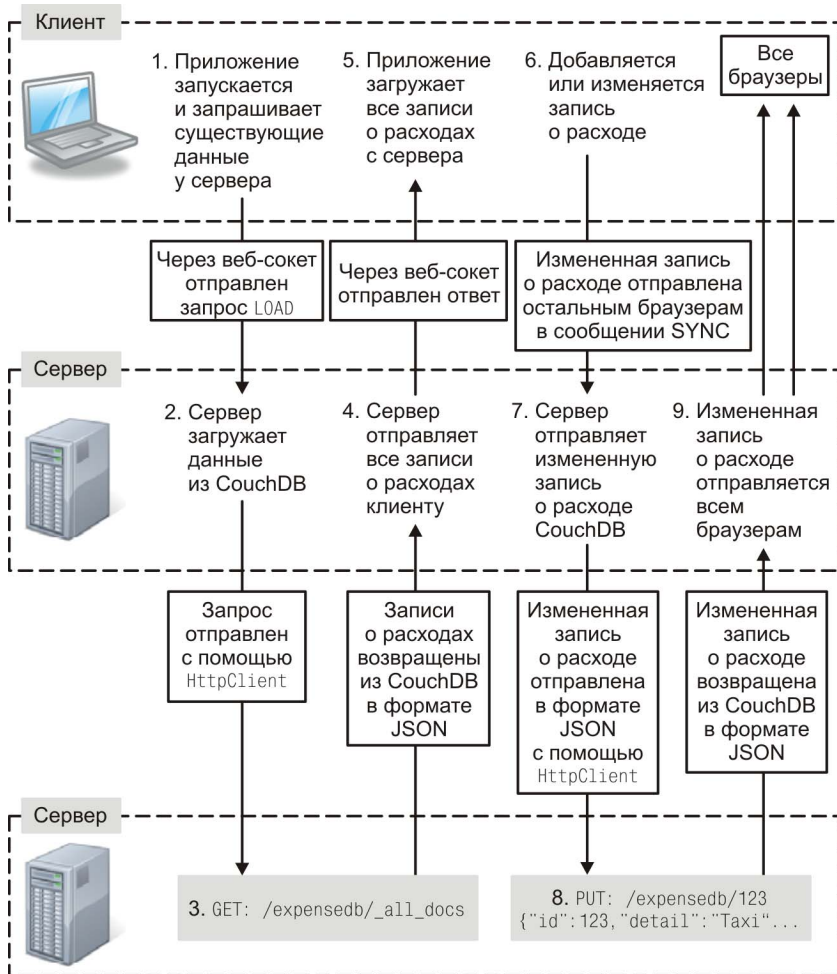


Рис. 14.9. Сохранение данных на стороне сервера в базе CouchDB

личных HTTP-методов и задания путей в запросе. В табл. 14.1 показано, к каким частям API мы будем обращаться. Если указан конкретный ID, его следует заменить идентификатором, который был присвоен записи о расходе браузерным веб-приложением.

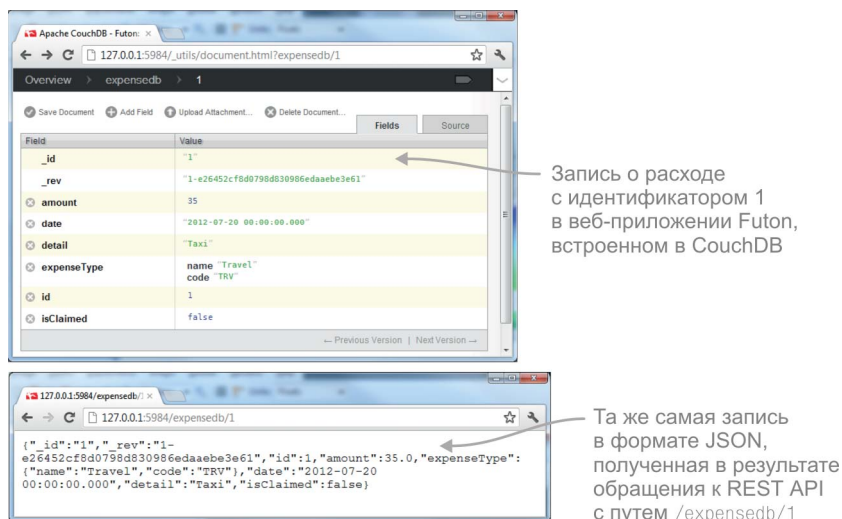


Рис. 14.10. Пользовательский веб-интерфейс к CouchDB и HTTP API

Таблица 14.1. Вызовы API, используемые в программе

HTTP-метод	Путь	Данные в запросе	Результат
PUT	/expensedb/	Нет	Создается новая запись в базе данных expensedb
GET	/expensedb/_all_docs	Нет	Из базы данных читаются все записи
PUT	/expensedb/1	Новая или измененная запись о расходе в формате JSON	Создается или обновляется запись о расходе с идентификатором ID = 1
GET	/expensedb/nextId	Нет	Читается запись о nextId

Каждый JSON-документ представлен идентификатором ID, который может быть произвольной строкой. В документе со специальным идентификатором nextId хранится идентификатор следующей записи о расходе, о котором должны знать все работающие экземпляры клиентского веб-приложения, чтобы случайно не присвоить двум разным расходам один и тот же идентификатор. В текущей версии программы это значение отправляется в сообщении типа SYNC

вместе с записью о расходе и сохраняется одновременно с обновлением записи.

Еще одна важная особенность CouchDB заключается в том, что вместе с записью о расходе необходимо сохранять уникальный номер ее версии. Это позволяет CouchDB узнать, является ли запись новой или существующей, и в случае существующей записи обновлять самую последнюю ее версию.

Предупреждение. Добавление в приложение нового слоя – базы данных – увеличивает его сложность. Листинги, приведенные в этой главе, основаны на предыдущих, но включены только фрагменты, в которых демонстрируется использование классов `HttpClient`, `InputStream` и `OutputStream`. Полный код приложения можно найти на сопроводительном сайте книги по адресу www.manning.com/DartInAction.

14.3.2. Совместное использование модельного класса *Expense* в коде клиента и сервера

Классом `Expense`, находящимся в файле `models.dart`, мы теперь воспользуемся не только в браузере, но и на сервере. Зачем писать еще раз почти идентичный код, если можно взять уже имеющийся файл, преобразовав его в библиотеку? Для этого нужно только добавить предложение `library` в начало файла, а затем импортировать его в клиентскую и серверную часть приложения.

Для сохранения JSON-данных в базе CouchDB нам необходимо свойство `_rev`, содержащее уникальный номер версии записи, который автоматически формируется CouchDB. Поэтому добавим такое свойство в класс `Expense`. В листинге 14.8 показано, как выглядит файл `models.dart` после преобразования в библиотеку и добавления свойства `_rev` в класс `Expense`. Когда к объекту класса `Expense` применяется функция `JSON.stringify()`, она использует реализованный в этом классе интерфейс `Map`, при этом в JSON-представление автоматически включается свойство `_rev`.

Листинг 14.8. `models.dart`: класс `Expense` после включения свойства `_rev`

```
library expense_models; ← Файл models.dart теперь является библиотекой
import "dart:json";

class Expense implements Hashable, Map {
  // ...остальные свойства опущены
```

```
String _rev = null;
String get rev => _rev;
set rev(value) => _rev = value;

// ...остальные методы опущены
operator [](key) {
    if (key == "id") {
        return this.id;
    } else if (key == "_rev") {
        return this.rev;
    } else if //...остальные свойства, включаемые в JSON, опущены
    }
}
```

14.3.3. Добавление поддержки сохранения данных на сервере

Все взаимодействие с базой данных CouchDB производится с помощью класса `HttpClient` из библиотеки `dart:io`. Как и другие серверные API, этот класс асинхронный и неблокирующий. Мы делаем запрос и получаем будущее значение, которое заполнится, когда сервер CouchDB вернет данные. Чтобы воплотить эту идею в жизнь, мы должны обернуть все обращения к CouchDB классом `CouchDbHandler`.

При использовании класса `HttpClient` мы обычно производим над данными операции, отправляя запросы методом `GET` или `PUT` и указывая в `URL` путь определенного вида, например `http://localhost:5984/expensedb/1`. В случае метода `PUT` запрос может содержать дополнительные данные, например измененную запись о расходе. Ответ, возвращаемый `CouchDB`, всегда имеет тип `String` и доставляется в виде потока `InputStream`. Поскольку это типичные способы взаимодействия с `CouchDB`, то мы включим в класс `CouchDbHandler` два закрытых служебных метода. Метод `_getStringFromInputStream()` будет преобразовывать `InputStream` в `Future<String>`, а метод `_getData()` — возвращать результат обращения к `CouchDB` в виде `Future<String>`.

Вспомогательные функции для взаимодействия с помощью HttpClient

Именно в этих служебных функциях и происходит вся интересная работа, относящаяся к взаимодействию с CouchDB. Код метода `getStringFromInputStream()` похож на тот, что мы видели в главе 13 при чтении данных из потока ввода `InputStream` из файла. Но на этот

раз данные читаются не из файла, а из потока `InputStream`, предоставленного объектом `HttpClientResponse`, который создается, когда сервер отвечает на запрос. Хотя данные доступны непосредственно, мы читаем их в объект `StringBuffer`, а когда поток `InputStream` закрывается, заполняем значение `Future<String>`, копируя в него содержимое буфера `StringBuffer`. В листинге ниже приведена реализация метода `_getStringFromInputStream()`.

Листинг 14.9. Служебный метод `_getStringFromInputStream()` из класса `CouchDbHandler`

```
class CouchDbHandler {

  Future<String> _getStringFromInputStream(var inputStream) {

    var completer = new Completer<String>();
    StringBuffer sb = new StringBuffer();    ← Создаем строковый буфер
                                              для хранения текста ответа

    inputStream.onData = () {
      var buffer = inputStream.read();
      if (buffer != null) {
        sb.add(new String.fromCharCode(buffer));
      }
    };

    inputStream.onClosed = () => completer.complete(sb.toString());    ←
                                                                           Содержимое строкового буфера
                                                                           копируется в будущее значение
                                                                           типа Future<String>

    return completer.future;    ← Возвращается
                                будущее значение

  }

  // остальные методы опущены
}
```

Этот метод используется во втором служебном методе `_getData()`, применяемом для отправки и получения данных из CouchDB. В нем мы вызываем метод `HttpClient.open()`, который возвращает новый объект соединения для работы с CouchDB API. Этот объект важен, потому что мы должны передать функции-обработчики его методам `onRequest()`, `onResponse()` и `onError()`. На рис. 14.11 показано, как эти методы используются в сочетании с объектом `HttpClient`.

В класс `CouchDbHandler` мы можем включить конкретную реализацию метода `_getData()`, следующую описанному принципу. Как видно из листинга 14.10, у метода `_getData()` есть два необязатель-

Метод `open()` возвращает новый объект соединения, к которому можно присоединить обработчики

В этом обращении к `open()` мы запрашиваем JSON-данные о расходе с идентификатором 1

```
var conn = httpClient.open("GET", "127.0.0.1", 5984, "/expensedb/1");

conn.onRequest = (HttpClientRequest req) {
    var outputStream = req.outputStream();
    outputStream.writeText("foo");
    outputStream.close();
}

conn.onResponse = (HttpClientResponse res) {
    var inputStream = res.inputStream();
    Future<String> str = _getStringFromInputStream(inputStream);
    str.then( (stringValue) {
        // do something with returned string
    });
}

conn.onError = (error) {
}
```

Если требуется отправить дополнительные данные (обычно при использовании метода POST или PUT), то их нужно записать в поток `outputStream` объекта `HttpClientRequest`

Данные из ответа, полученного объектом соединения, можно прочитать из потока `inputStream` объекта `HttpClientResponse`

В случае ошибки вызывается обработчик `onError()`

Рис. 14.11. Использование `HttpClient` для чтения данных из `HttpServer`

ных параметра, `method` и `data`, которые позволяют посылать запрос методом PUT или POST вместо подразумеваемого по умолчанию GET и включать в запрос данные, например измененную запись о расходе. Объект `Future<String>`, возвращаемый методом `_getData()`, представляет JSON-данные, прочитанные из базы CouchDB.

Листинг 14.10. Служебный метод `_getData()` из класса `CouchDbHandler`

```
class CouchDbHandler {
    Future<String> _getData(String path, {method: 'GET', data: null}) {

        var conn = client.open(method, host, port, path); ← Вызываем сервер и получаем
                                                         в ответ новый объект
                                                         соединения

        if (data != null) {
            conn.onRequest = (HttpClientRequest req) {
                var outputStream = req.outputStream();
                outputStream.writeString(data);
                outputStream.close();
            };
        }
    }
}
```

Если нужно отправить данные, устанавливаем обработчик `onRequest`

```

var completer = new Completer<String>();
conn.onResponse = (HttpClientResponse response) {
  _getStringFromInputStream(response.inputStream)
    .then((responseText) {
      completer.complete(responseText);
    });
};

conn.onError = (e) {
  print(e);
};

return completer.future;
}
}

```

Копируем ответ `responseText` от CouchDB в объект `Future<String>`

Сообщение
об ошибке выводим
на консоль

Модификация *DartExpense*

Теперь, имея в классе `CouchDbHandler` служебные методы, мы можем добавить оставшуюся функциональность: конструктор и три открытых метода. Конструктор создаст новую базу `expensedb`, если она еще не существует, а каждый из методов `loadData()`, `addOrUpdate()` и `getNextId()` будет возвращать будущее значение. Общая структура методов изображена на рис. 14.12.

Все три метода обращаются к `_getData()`, а возвращенное им значение `Future<String>` мы используем для преобразования полученной

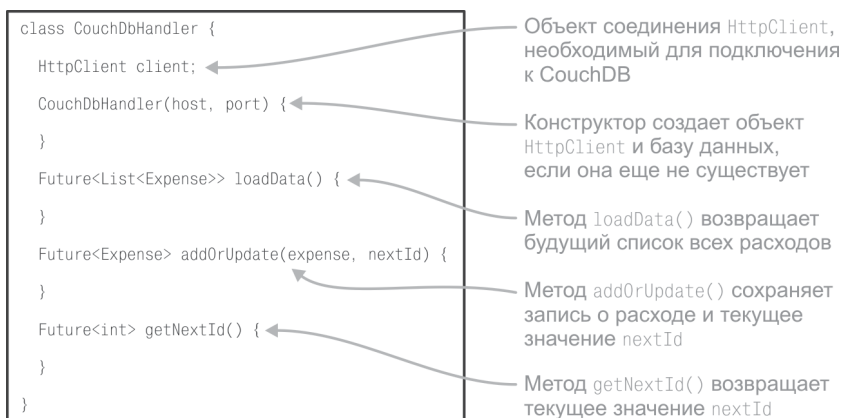


Рис. 14.12. Структура серверного класса `CouchDbHandler`

JSON-строки в объект `Expense`. В табл. 14.2 показано, как именно программа обращается к методу `_getData()`.

В листинге ниже приведен код конструктора и трех методов класса `CouchDbHandler`. Вызовы, перечисленные в табл. 14.2, выделены полужирным шрифтом; остальной код прямолинеен и служит для преобразования объекта `Expense` в формат JSON, ожидаемый CouchDB, и наоборот.

Таблица 14.2. Обращения к служебному методу `_getData()` из программы

Вызов метода	Действие
<code>_getData("/expensedb/", method: "PUT");</code>	Создает базу данных
<code>_getData("/expensedb/\$id?include_docs=true");</code>	Извлекает все записи о расходах
<code>_getData("/expensedb/\$id", method: "PUT", data: expense.toJson());</code>	Создает или обновляет запись о расходе в базе данных
<code>_getData("/expense/nextId");</code>	Извлекает значение <code>nextId</code>

Листинг 14.1. CouchDbHandler: добавление новой функциональности в DartExpense

```
class CouchDbHandler {

  //..._getStringFromInputStream() опущен
  //..._getData() опущен

  HttpClient client;
  var host;
  var port;

  CouchDbHandler(this.host, this.port) {
    this.client = new HttpClient();
    _get("/expensedb/", method: "PUT"); ← Создаем базу данных, если она еще не существует
  }

  Future<List<Expense>> loadData() {
    var completer = new Completer<List<Expense>>();

    var expenseList = new List<Expense>();

    _getData("/expensedb/_all_docs?include_docs=true") ← Получаем все хранящиеся
    .then( (responseText) {                                     в базе данных записи
      Map data = JSON.parse(responseText);
    });
  }
}
```

```

    for (var rowData in data["rows"]) {
        if (rowData["id"] != "nextId") {
            var expenseJSON = rowData["doc"];
            var expense = new Expense.fromMap(expenseJSON);
            expenseList.add(expense);
        }
    }

    completer.complete(expenseList); ← Заполняем будущий список расходов
});
return completer.future;
}

Future<Expense> addOrUpdate(Expense expense, int nextId) {
    var completer = new Completer<Expense>();

    _getData("/expensedb/${expense.id}", ← Методом PUT отправляем
    ↪      method: 'PUT', data: expense.toJson() ) данные о расходе в CouchDB
    ↪ .then( (responseText) {
        Map data = JSON.parse(responseText);
        expense.rev = data["rev"];
        completer.complete(expense); ← Сохраняем возвращенный номер версии "rev"
        и заполняем будущее значение расхода
    });

    var nextIdMap = new Map();
    nextIdMap["nextId"] = nextId;
    _getData("/expensedb/nextId", ← Сохраняем текущее значение nextId в базе данных
    ↪      method: "PUT", data: JSON.stringify(nextIdMap));

    return completer.future;
}

Future<int> getNextId() {
    var completer = new Completer<int>();

    _getData("/$dbName/nextId").then( (responseText) { ← Получаем текущее значение
    ↪      var data = JSON.parse(responseText);
    ↪      var nextId = 1;
    ↪      if (data.containsKey("nextId")) {
    ↪          nextId = data["nextId"];
    ↪      }

    ↪      completer.complete(nextId); ← Заполняем будущее значение nextId
    ↪    });

    return completer.future;
}
}

```


Листинг довольно длинный, зато содержит весь код, необходимый для сохранения и загрузки данных на сервере. Чтобы включить этот класс в существующее приложение, достаточно обратиться к методам `loadData()` и `addOrUpdate()` из рассмотренного выше кода работы с соединением через веб-сокеты. Когда браузер присылает запрос с действием `LOAD`, мы возвращаем список расходов в формате JSON, а в ответ на запрос с действием `SYNC` передаем запись о расходе методу `addOrUpdate()`. Полный исходный код приложения имеется на сайте издательства Manning.

Памятка

- Метод `open()` класса `HttpClient` возвращает объект соединения, к которому можно присоединить обработчики `onRequest()` и `onResponse()`.
 - Обработчику `onRequest()` передается параметр типа `HttpClientRequest`, запись в его свойство `OutputStream` приводит к отправке данных.
 - Обработчику `onResponse()` передается параметр типа `HttpClientResponse`, чтение из его свойства `InputStream` приводит к получению данных.
-

14.4. Резюме

В этой главе мы рассмотрели еще два серверных API, с помощью которых можно писать интересные и функционально насыщенные приложения на Dart. Технология веб-сокетов, появившаяся в HTML5, позволяет организовать постоянное двустороннее соединение между клиентом и сервером, по которому сервер может посылать клиентскому приложению в браузере данные даже без запроса с его стороны. Этой возможностью можно воспользоваться для совместного редактирования документов и синхронизации данных между разными браузерами.

Мы изучили также класс `HttpClient`, который наделяет серверное Dart-приложение способностью общаться с другими серверами (объектами `HttpServer`) через REST API. Это серверный эквивалент клиентского класса `HttpRequest`. Мы воспользовались им для сохранения в базе данных CouchDB получаемых от клиентской части приложения `DartExpense` объектов, представленных в формате JSON.

В следующей главе мы рассмотрим более сложный аспект Dart – изоляторы. С их помощью мы можем реализовать параллельную обработку, воспользовавшись возможностями современных многоядерных процессоров для организации нескольких циклов обработки событий, обменивающихся между собой сообщениями. Это может оказаться полезным, когда нужно выполнить длительный фоновый процесс, но при этом реагировать на запросы клиентов.



Глава 15. Организация параллелизма с помощью изоляторов

В этой главе

- ❑ Запуск функции в новом изоляторе.
- ❑ Передача сообщений между изоляторами.
- ❑ Динамическая загрузка Dart-кода.
- ❑ Использование изоляторов для пакетной обработки.

Оборудование современных компьютеров обеспечивает большую вычислительную мощность, чем когда-либо ранее, – будь то сервер, обслуживающий тысячи запросов в минуту, или мобильный телефон либо планшет у вас в кармане. Вместо того чтобы без конца наращивать тактовую частоту процессора, проектировщики микросхем достигают этого результата за счет увеличения числа процессорных ядер. Поэтому веб-приложение, желающее воспользоваться дополнительной мощностью, должно уметь задействовать все имеющиеся ядра.

Выше мы говорили, что Dart-код работает в одном потоке, но умеет выполнять асинхронные операции, например при доступе к файловой системе или к внешним серверам. В браузере приложение пользуется асинхронными API для отправки AJAX-запросов серверу. Но сама Dart-программа по-прежнему работает в одном потоке и задействует только одно процессорное ядро. Это проектное ограничение введено для того, чтобы написанный на Dart код можно было преобразовать в JavaScript – тоже однопоточный язык.

В этой главе мы будем рассматривать изоляторы – подход, принятый в Dart, для того чтобы дать однопоточной программе возможность полнее задействовать имеющееся оборудование. Изоляторами, как нетрудно догадаться, называются изолированные блоки исполняемого кода. Единственный способ обмена данными между ними – передача сообщений, примерно так же, как передаются со-

общения между клиентом и сервером или между Dart и JavaScript. Мы воспользуемся изоляторами на сервере и модифицируем приложение «Обозреватель файлов Dart» из главы 13, так чтобы оно анализировало результирующий список файлов.

Познакомившись с основными идеями изоляторов, мы покажем, как воспользоваться ими для динамической загрузки и исполнения кода. Это позволит создать механизм подключаемых модулей, то есть возможность выполнить сторонний код, не зная заранее о его существовании.

И напоследок мы посмотрим, как с помощью изоляторов создать несколько исполнителей для параллельной обработки списка. Это действенная техника разработки приложений, способных в полной мере задействовать имеющееся оборудование и достигать очень высокой производительности.

Начнем с описания общей концепции изоляторов и создания нового изолятора. Параллельное программирование – это сложная задача, и, хотя изоляторы Dart в какой-то мере упрощают ее решение, все равно необходимо познакомиться с двумя новыми классами: `ReceivePort` и `SendPort`.

15.1. Использование изоляторов как единиц работы

В главе 13 мы написали серверный скрипт, который строил список файлов и папок в текущем каталоге. Используя асинхронную функцию `DirectoryLister()` для чтения содержимого каталога, мы передаем управление операционной системе. Точно так же можно передать управление из одной части Dart-программы в другую, работающую асинхронно, часть, в которой проанализировать результаты – подсчитать количество и общий размер файлов разных типов. Сначала мы реализуем эту идею в виде простого серверного скрипта с одним изолятором, а затем на его основе напишем скрипт, анализирующий список файлов в нескольких изоляторах. На рис. 15.1 изображена структура приложения «Анализ каталога», которое получает список файлов, после чего подсчитывает количество и общий размер файлов каждого типа.

Две самые важные функции – это `getFileTypes()` и `getFileSizes()`. В предыдущей реализации они вызывались последовательно; теперь мы воспользуемся изоляторами и заставим их работать одновременно.

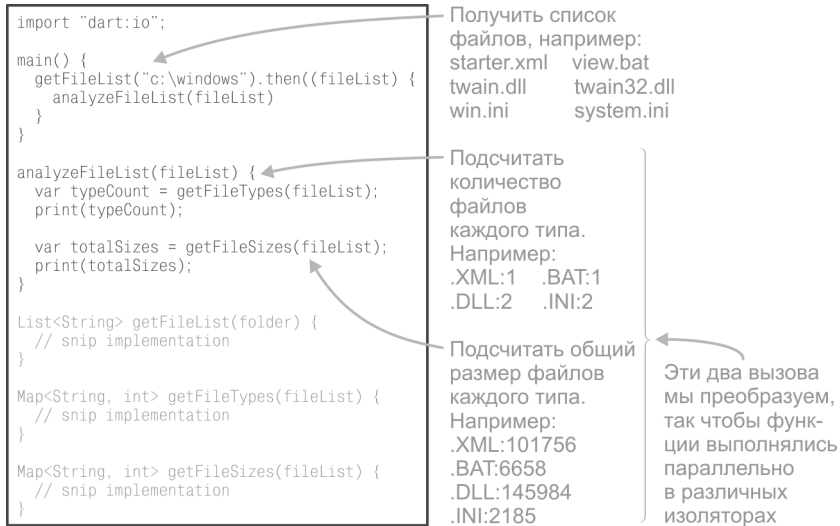


Рис. 15.1. Структура приложения «Анализ каталога»

15.1.1. Создание изолятора

На самом деле мы уже встречались с изоляторами. Любое написанное на Dart приложение выполняется в так называемом *изоляторе по умолчанию*. Запуская приложение, виртуальная машина Dart создает изолятор и передает ему функцию `main()` в качестве точки входа. Функция `main()` начинает исполнять однопоточный код в изоляторе. Это функция верхнего уровня, которая не принимает параметров и ничего не должна возвращать.

Ваша Dart-программа может точно так же, как ВМ, создать изолятор и передать ему функцию, являющуюся точкой входа. Как и `main()`, эта функция должна располагаться на верхнем уровне, не принимать параметров и ничего не возвращать. Имя функции может быть произвольным, но функция `main()` может исполняться только в изоляторе по умолчанию.

Для создания нового изолятора с указанной точкой входа необходимо импортировать библиотеку `dart:isolate` и вызвать функцию `spawnFunction()`, передав ей имя точки входа. В листинге 15.1 приведена модифицированная функция `analyzeFileList()`, которая исполняет функцию подсчета количества и общего размера файлов каждого типа в разных изоляторах (`FileTypes` и `FileSizes`). Пока что этот

скрипт не делает ничего полезного: он лишь создает два изолятора, которые печатают сообщение и тут же завершаются. Иными словами, это аналог программы «Здравствуй, мир», только с изоляторами.

Примечание. Запустив эту программу, вы с удивлением обнаружите, что сообщение в изоляторе `FileSizes` печатается не до конца. Изолятор по умолчанию (в котором выполняется функция `main()`) выходит первым, и программа прекращает работу, не дожидаясь завершения остальных изоляторов. В этом примере мы добавили в конец `main()` цикл ожидания; позже мы его удалим.

Листинг 15.1. `directoryAnalysis.dart`

```
import "dart:isolate";  ← Импортируется библиотека dart:isolate

void main() {
  // опущено ...
  analyzeFileList(fileList);
}

analyzeFileList(fileList) {
  spawnFunction(getFileTypesEntryPoint);
  spawnFunction(getFileSizesEntryPoint);
  print("Default isolate");

  for (int i = 0; i < 100000; i++) { }
}

void getFileTypesEntryPoint() {
  print("FileTypes isolate");
}

void getFileSizesEntryPoint() {
  print("FileSizes isolate");
}
```

Запускаем функции,
являющиеся точками входа,
в изоляторах

← Цикл ожидания: дает возможность
запущенным изоляторам завершиться
до завершения приложения

Точка входа:
не принимает параметров
и возвращает void

Функция `spawnFunction()` запускает новый изолятор. Виртуальная машина создает поток, который будет исполнять код, и, как мы увидим ниже, этот поток не разделяет никакое состояние с родительским потоком.

Это важный момент. Весь код находится в одном и том же физическом файле, но теперь существуют три контекста выполнения, каждый со своей памятью, объектами и состоянием. В этом легко

убедиться, добавив в библиотеку свойство верхнего уровня и изменив его значение в каждом из трех изоляторов, как показано на рис. 15.2.

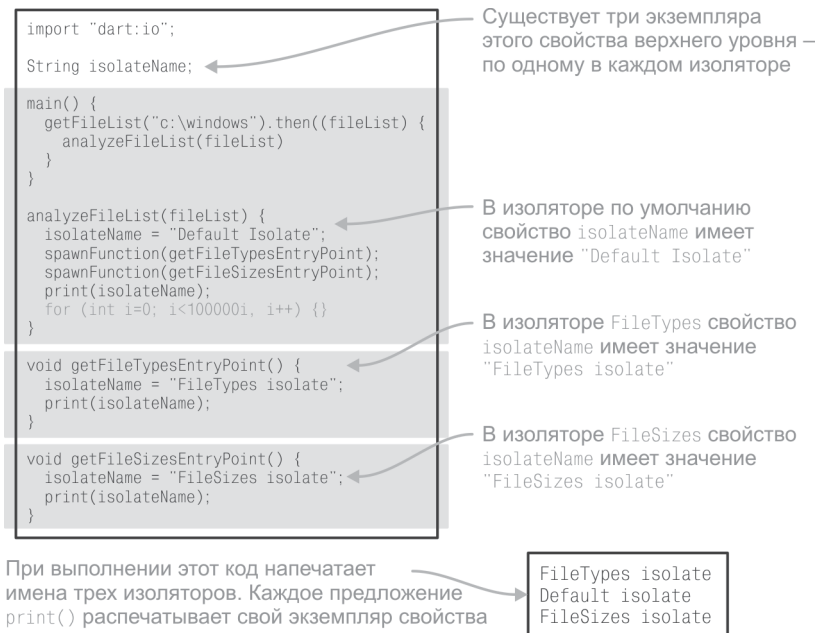


Рис. 15.2. Свойство верхнего уровня `isolateName` в каждом изоляторе уникально

Каждый изолятор получает новый контекст памяти, отделенный от всех остальных. В этом отношении имеется концептуальное сходство с исполнением написанного на Dart и JavaScript кода в одном браузере или с исполнением программы на клиенте и на сервере.

Научившись создавать изоляторы, мы можем поручить им какую-то работу. Для этого необходимо передать изолятору данные, например список файлов. Мы покажем также, как предотвратить завершение изолятора по умолчанию (не добавляя цикла ожидания `for`), чтобы другие изоляторы могли доработать до конца.

15.1.2. Односторонняя связь с изолятором

Каждый изолятор Dart выполняется в своем собственном контексте. Это означает, что у изоляторов нет общей памяти. В изоляторе по

умолчанию список файлов хранится в переменной `fileList`. Чтобы сделать эту переменную доступной другим изоляторам, изолятор по умолчанию должен передать ее в виде сообщения – точно так же, как передаются сообщения между клиентом и сервером или между Dart и JavaScript.

Однако изоляторы поддерживают более гибкую передачу сообщений. Сообщения необязательно предварительно преобразовывать в формат JSON, некоторые типы можно передавать непосредственно – их полный перечень приведен в табл. 15.1.

Таблица 15.1. Типы, которые можно передавать между изоляторами

Типы	Классы	Примечания
Примитивные типы	<code>null</code> , <code>num</code> , <code>bool</code> , <code>double</code> , <code>String</code>	Хотя это экземпляры классов, при передаче между изоляторами они трактуются как примитивные типы
Типы коллекций	<code>List</code> , <code>Map</code>	Могут содержать экземпляры допустимых примитивных типов и типов коллекций, в том числе списки и словари

Отправка сообщений другому изолятору

Функция `spawnFunction`, создающая изолятор, возвращает экземпляр класса `SendPort`, который предоставляет метод `send()` для отправки данных этому изолятору:

```
SendPort fileTypesSendPort = spawnFunction(getFileTypesEntryPoint);  
var message = "Привет изолятору FileTypes от изолятора по умолчанию";  
fileTypesSendPort.send(message);
```

Пока все просто, но этот код работать не будет. Принимающий изолятор `FileTypes` должен уметь получать данные, при этом отправлять данные могут различные изоляторы.

Прием данных от другого изолятора

Изолятор получает сообщения посредством функции обратного вызова, присоединенной к экземпляру класса `ReceivePort`. Любой изолятор, в том числе подразумеваемый по умолчанию, имеет порт приема `ReceivePort`, к которому можно обратиться через свойство верхнего уровня `port`. Свойство `port` определено в библиотеке `dart:isolate`, и, подобно созданному нами ранее свойству `IsolateName`, в каждом изоляторе свое.

Обращаясь к свойству `port`, мы обращаемся к экземпляру `ReceivePort` в *данном изоляторе*; в любом изоляторе есть только один порт приема. Это означает, что обработчику `port.receive(callback)` будут поступать все сообщения, отправленные данному изолятору из любого другого. Это справедливо как для изолятора по умолчанию, так и для дополнительных изоляторов типа `FileTypes` или `FileSizes` (рис. 15.3). Функция обратного вызова принимает два параметра: `data` одного из допустимых для передачи между изоляторами типов и `replyTo` типа `SendPort`. Параметр `replyTo` мы подробнее рассмотрим ниже.

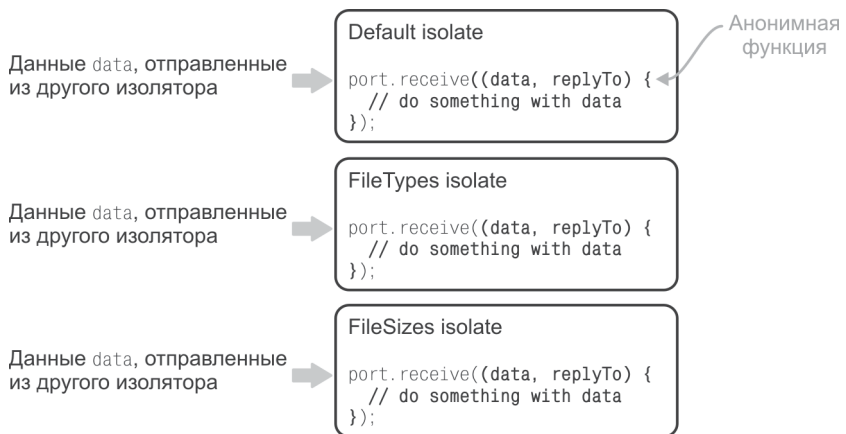


Рис. 15.3. Обработчик `port.receive()` вызывается для приема данных, адресованных текущему изолятору

Когда изолятор по умолчанию отправляет данные изолятору `FileTypes`, тот принимает их и может обработать. В листинге ниже приведена сокращенная версия текущего серверного скрипта, в которой изолятор `FileTypes` принимает от изолятора по умолчанию сообщение вида `"hello world"`.

Листинг 15.2. Изолятор `FileTypes` принимает данные от изолятора по умолчанию

```
import "dart:isolate";

void main() {
  // опущено ...
```



```

analyzeFileList(fileList);
}

analyzeFileList(fileList) {
    var fileTypesSendPort = spawnFunction(getFileTypesEntryPoint);
    fileTypesSendPort("Привет от изолятора по умолчанию");
    print("Это изолятор по умолчанию");
}

void getFileTypesEntryPoint() {
    print("Это изолятор FileTypes");
    port.receive( (data, replyTo) {
        print(data);
    });
}

```

Изолятор по умолчанию создает
изолятор FileTypes

← Изолятор по умолчанию
посылает сообщение
изолятору FileTypes

Изолятор FileTypes получает сообщение
← и передает его функции обратного вызова

← Печатается «Привет от изолятора по умолчанию»

Функция `sendPort.send()` позволяет отправлять данные от изолятора по умолчанию в изолятор `FileTypes`. Теперь мы готовы к тому, чтобы добавить возможность анализа списка файлов. Поток данных в новой версии приложения показан на рис. 15.4.

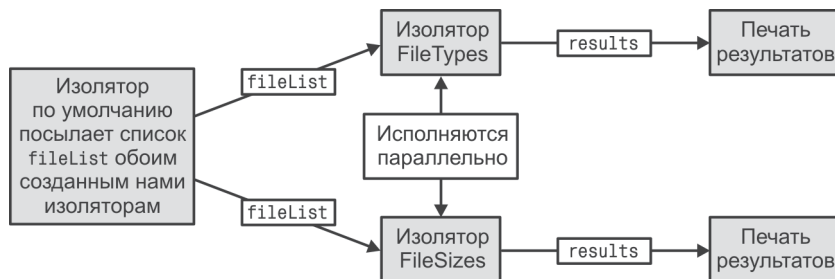


Рис. 15.4. Изолятор по умолчанию отправляет данные другим изоляторам

В листинге 15.3 приведен код анализа файлов в двух изоляторах, каждому из которых посылается один и тот же список файлов. Каждый изолятор выводит полученные результаты на консоль.

Листинг 15.3. Подсчет количества и общего размера файлов каждого типа в двух разных изоляторах

```

import "dart:io";
import "dart:isolate";

```

```

void main() {
  getFileList("c:\windows").then(List<String> fileList) {
    analyzeFileList(fileList);
  });
}

analyzeFileList(fileList) {
  var fileTypeSendPort = spawnFunction(getFileTypesEntryPoint);
  fileTypeSendPort.send(fileList);
  var fileSizeSendPort = spawnFunction(getFileSizesEntryPoint);
  fileSizeSendPort.send(fileList);
}

void getFileTypesEntryPoint() {
  var receivePort = port;
  receivePort.receive((data, replyTo) {
    Map<String,int> typeCount = getFileTypes(data);
    print(typeCount);
  }));
}

void getFileSizesEntryPoint() {
  var receivePort = port;
  receivePort.receive((data, replyTo) {
    Map<String,int> totalSizes = getFileSizes(data);
    print(totalSizes);
  }));
}

// getFileList(), getFileSizes(), getFileTypes() опущены

```

Создаем изолятор FileTypes, передавая ему функцию верхнего уровня

← Отправляем fileList изолятору FileTypes

Присоединяем обработчик принятых данных к изолятору FileTypes

Получив список файлов, изолятор анализирует его и выводит результаты на консоль

Изолятор FileSizes работает аналогично изолятору FileTypes

Но в этом коде осталась одна проблема: изолятор по умолчанию не ждет завершения двух остальных. В идеале мы должны были бы начать анализ файлов в изоляторах FileTypes и FileSizes, будучи уверены, что изолятор по умолчанию дождетсЯ завершения. Этим мы и займемся далее, а заодно рассмотрим параметр `replyTo`, обеспечивающий механизм двусторонней связи между изоляторами.

15.1.3. Двусторонняя связь с изолятором

Мы видели, как посылать изолятору данные, но односторонняя передача не так полезна, как двусторонний обмен данными между изоляторами. Следующий шаг – заставить изолятор по умолчанию ждать, пока изоляторы FileTypes и FileSizes вернут результат обработки (рис. 15.5).

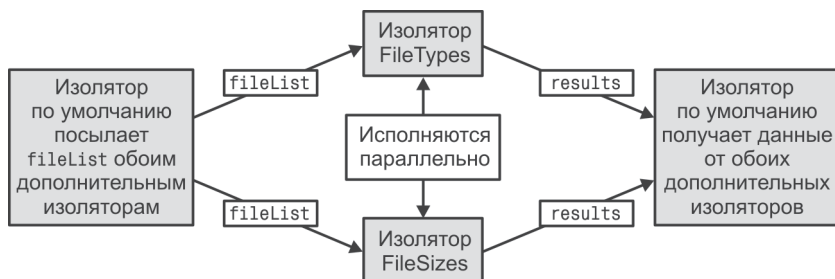


Рис. 15.5. Изолятор по умолчанию ждет, пока дополнительные изоляторы вернут ему результаты

В модифицированном скрипте изолятор по умолчанию получает результаты и печатает их на консоль. Коль скоро изолятор по умолчанию ждет результатов, он не завершается немедленно.

И снова об объекте SendPort

Чтобы изолятор FileTypes мог вернуть данные изолятору по умолчанию, ему необходим доступ к объекту SendPort последнего. Напомним, что объект SendPort предоставляет метод `send()` для отправки данных изолятору, в том числе подразумеваемому по умолчанию. Отношения между SendPort, ReceivePort и изоляторами показаны на рис. 15.6.

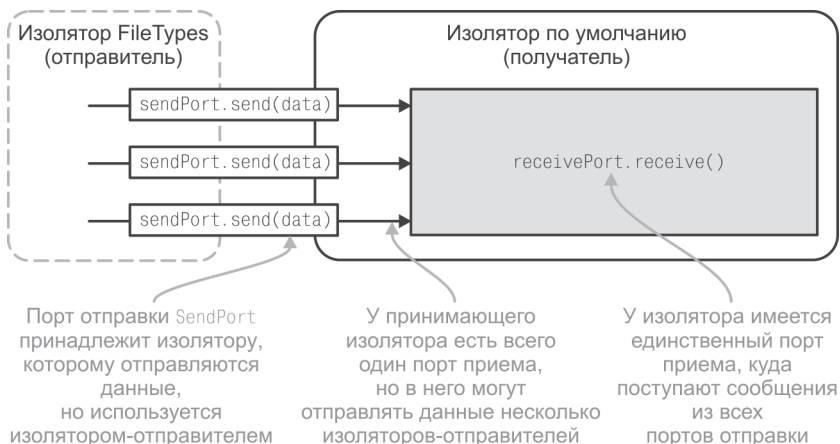


Рис. 15.6. Принимающий изолятор имеет один порт приема ReceivePort, но несколько портов отправки SendPort

Один изолятор может отправлять данные другому с помощью порта `SendPort`. Ранее мы видели, что функция `spawnFunction()` возвращает объект `SendPort` нового изолятора `FileTypes`. Но как изолятор `FileTypes` получает объект `SendPort` изолятора по умолчанию, чтобы отправить ему данные?

Использование `SendPort` в качестве аргумента `replyTo`

Именно здесь и вступает в игру аргумент `replyTo` функции обратного вызова при приеме. Необязательный аргумент `replyTo` метода `send()` позволяет изолятору-отправителю передать *свой* `SendPort` изолятору-получателю.

Отправитель получает собственный экземпляр `SendPort`, вызывая метод своего же объекта `ReceivePort`: `port.toSendPort()`. Полученный `SendPort` можно передать другому изолятору, как показано в примере ниже:

<pre>main() { var defaultIsolateSendPort = port.toSendPort(); }</pre>	Получаем <code>SendPort</code> изолятора ← по умолчанию
<pre>getFileTypesEntryPoint() { var fileTypesIsolateSendPort = port.toSendPort(); }</pre>	Получаем <code>SendPort</code> ← изолятора <code>FileTypes</code>

Получив собственный объект `SendPort`, изолятор может передать его методу `send(data, replyTo)` объекта `SendPort` получателя. На рис. 15.7 показана упрощенная версия скрипта «Анализ каталога», в которой изолятор по умолчанию отправляет сообщение «hello world» изолятору `FileTypes`, а тот отправляет сообщение «hello back» обратно изолятору по умолчанию.

Обмен данными между изоляторами

Получив способ отправить данные назад вызвавшему изолятору, мы можем модифицировать приложение «Анализ каталога», так чтобы изолятор по умолчанию ожидал результатов от изоляторов `FileTypes` и `FileSizes`. Ниже показано, как это выглядит.

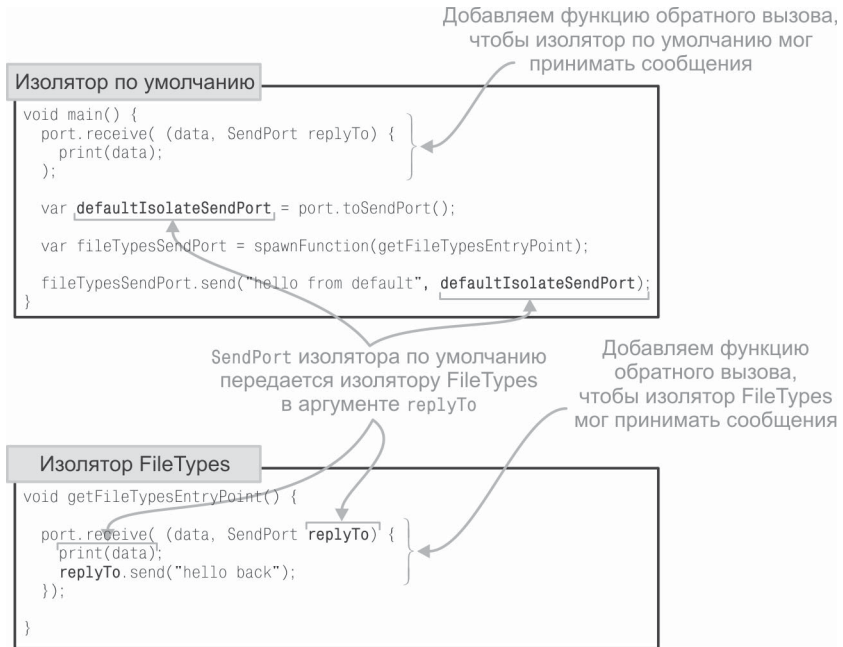


Рис. 15.7. Объект SendPort одного изолятора передается другому в аргументе replyTo

Листинг 15.4. Изоляторы FileTypes и FileSizes посылают результаты изолятору по умолчанию

// прочие функции и импортированные библиотеки опущены

```
analyzeFileList(fileList) {
    port.receive((data, replyTo) {
        print(data);
    });

    var defaultIsolateSendPort = port.toSendPort();
    var fileTypesSendPort = spawnFunction(getFileTypesEntryPoint);
    fileTypesSendPort.send(fileList, defaultIsolateSendPort);
}
```

Добавляем функцию обратного вызова при приеме в изолятор по умолчанию

← Получаем SendPort изолятора по умолчанию

← Посылаем SendPort изолятора по умолчанию изолятору FileTypes

```

var fileSizesSendPort = spawnFunction(getFileSizesEntryPoint);
fileSizesSendPort.send(fileList, defaultIsolateSendPort); ←
}
    Посылаем SendPort изолятора по умолчанию изолятору FileSizes

void getFileTypesEntryPoint() {
    var receivePort = port;
    receivePort.receive((data, replyTo) {
        Map<String,int> typeCount = getFileTypes(data);
        replyTo.send(typeCount); ← Передаем результаты обратно изолятору по умолчанию
    });
}

void getFileSizesEntryPoint() {
    var receivePort = port;
    receivePort.receive((data, replyTo) {
        Map<String,int> totalSizes = getFileSizes(data);
        replyTo.send(totalSizes); ← Передаем результаты обратно изолятору по умолчанию
    });
}

```

Теперь изолятор по умолчанию отвечает за печать результатов, полученных от изоляторов FileTypes и FileSizes; он не завершается немедленно, а ждет сообщений от других изоляторов, которые должны поступить его функции обратного вызова при приеме. Но это означает, что у нас появилась новая проблема. Если раньше приложение завершалось слишком рано, то теперь оно неопределенно долго ждет сообщений от других изоляторов. Нам необходимо как-то сообщить изолятору по умолчанию, что он уже получил все данные и больше ждать нечего.

Заккрытие порта приема

К счастью, мы можем воспользоваться методом `close()` объекта `ReceivePort`, чтобы закрыть коммуникационный порт изолятора. После вызова `close()` сообщения, находившиеся в процессе передачи, теряются, и порт `ReceivePort` больше не получает никаких сообщений. Если это был порт приема изолятора по умолчанию, то тот продолжает выполняться, не ожидая сообщений.

Поскольку наш скрипт «Анализ каталога» ждет возврата двух наборов результатов, то мы можем просто добавить в обработчик принимаемых сообщений счетчик. Получив оба набора, мы закрываем `ReceivePort`, как показано в листинге ниже.

Листинг 15.5. Заккрытие ReceivePort после получения всех данных

```
analyzeFileList(fileList) {  
    var replyCount = 0;  
  
    port.receive((data, replyTo) {  
        replyCount ++; ← Увеличиваем счетчик после получения каждого ответа  
        print(data);  
  
        if (replyCount == 2) {  
            port.close();  
        }  
    });  
  
    var defaultIsolateSendPort = port.toSendPort();  
  
    var fileTypeSendPort = spawnFunction(getFileTypesEntryPoint);  
    fileTypeSendPort.send(fileList, defaultIsolateSendPort);  
    var fileSizeSendPort = spawnFunction(getFileSizesEntryPoint);  
    fileSizeSendPort.send(fileList, defaultIsolateSendPort);  
}
```

Получив
два ответа,
закрываем ReceivePort

Теперь приложение «Анализ каталога» параллельно исполняет изолированные участки кода, которые обмениваются между собой сообщениями. Все точки входа в изоляторы имеют точно такую же сигнатуру, как функция `main()`, являющаяся точкой входа в изолятор по умолчанию (не принимают параметров и возвращают `void`). Мы воспользуемся этой возможностью в следующем разделе, где разобьем приложение «Анализ каталога» на отдельные файлы, которые будем загружать динамически из изолятора по умолчанию.

Памятка

- У каждого изолятора имеется доступ к свойству верхнего уровня `port`, которое представляет его порт ввода – экземпляр класса `ReceivePort`.
 - Функция `SendPort spawnFunction(entryPointFunction)` создает новый изолятор и возвращает объект `SendPort`, через который ему можно посылать данные.
 - Для отправки изолятору данных вызывается метод `send()` объекта `SendPort`, принадлежащего изолятору-приемнику.
 - Для получения собственного порта отправки `SendPort` изолятор может вызвать метод `port.toSendPort()`.
-

15.2. Динамическая загрузка кода

Сейчас в приложении «Анализ каталога» имеются две функции анализа: одна подсчитывает количество типов файлов, другая – суммарный размер файлов каждого типа. Если открыть приложение, то есть дать пользователям возможность расширять его возможности с помощью подключаемых модулей, то оно сможет решать новые задачи без изменения основного кода.

Мы изменим приложение, так что оно будет динамически загружать исходные файлы, указанные в командной строке. Каждый такой файл производит тот или иной вид анализа списка файлов и возвращает результаты главному приложению. На рис. 15.8 показано, как работает новое приложение.

Чтобы реализовать этот план, нам нужно будет изменить главный файл `DirAnalysis.dart`, так чтобы он читал аргументы из командной строки. Для этого мы воспользуемся классом `Options`, который рассматривался в главе 13. Первым аргументом является подлежащий анализу каталог, за ним перечисляются исходные файлы, которые загружаются и вызываются для анализа списка файлов. В листинге ниже показана функция `main()` в файле `DirAnalysis.dart`, которая заполняет списки `fileList` и `dynamicSourceFiles`.

Листинг 15.6. `DirAnalysis.dart`: чтение аргументов в командной строке

```
import "dart:io";
import "dart:isolate";

void main() {
  var options = new Options();
  var analysisFolder = null;
  var dynamicSourceFiles = new List<String>();
  for (String argument in options.arguments) {
    if (analysisFolder == null) {
      analysisFolder = argument;
    } else {
      dynamicSourceFiles.add(argument);
    }
  }

  getFileList(analysisFolder).then(List<String> fileList) {
```

← Создаем объект `Options` для доступа к аргументам в командной строке

Переменные, в которых будут храниться анализируемый каталог и список динамически загружаемых исходных файлов

← Для каждого аргумента в командной строке...

...первый считается путем к подлежащему анализу каталога

Остальные – имена исходных файлов-анализаторов

Получаем список файлов из анализируемого каталога

← мого каталога

1. Перечень Dart-программ, выполняющих анализ каталога c:\windows, передается в командной строке



2. Каждый файл динамически загружается в собственный изолятор и получает данные от изолятора по умолчанию



Рис. 15.8. Приложение «Анализ каталога» динамически загружает исходные файлы, которые производят анализ списка файлов

```
analyzeFileList(fileList, dynamicSourceFiles); ← Передаем список файлов и список
});                                           динамически загружаемых
}                                             анализаторов

analyzeFileList(List<String> fileList, List<String> dynamicSourceFiles) {
  // todo: динамически загрузить исходный файл и передать ему список файлов
}

// функция getFileList() опущена
```

15.2.1. Создание изолятора для загружаемого файла

Теперь у нас есть переменная `fileList`, в которой могут храниться, например, список файлов в каталоге `c:\windows` и список `dynamicSourceFiles`, содержащий имена программных модулей, например `fileSize.dart` и `fileTypes.dart`. Главное приложение должно загрузить в изолятор все динамически загружаемые модули, о которых заранее не знает, а затем передать каждому какие-то данные. Все, как в функции `spawnFunction()`, только код динамически загружается из внешнего файла, поэтому мы используем функцию `spawnUri()`, передавая ей URI-адрес или имя файла, например `spawnUri("fileTypes.dart");`.

Модифицируем функцию `analyzeFileList()`, так чтобы она обходила список программных модулей, запускала каждый из них в отдельном изоляторе и передавала ему переменную `fileList`. Изолятор по умолчанию (приложение `DirAnalysis.dart`) будет ждать, пока не получит столько ответов, сколько загрузил модулей, после чего закроет свой входной порт. В следующем листинге приведена описанная только что версия функции `analyzeFileList()`.

Листинг 15.7. `DirAnalysis.dart`: использование функции `spawnUri()` для динамической загрузки исходного файла

```
// функции main() и getFileList() опущены
analyzeFileList(List<String> fileList, List<String> dynamicSourceFiles) {
    var replyCount = 0;    ← Счетчик ответов от запущенного изолятора
                           Настраиваем обработчик
    port.receive( (data, replyTo) { ← принятых данных в изоляторе по умолчанию
        print(data);
        replyCount++;
        if (replyCount == dynamicSourceFiles.length) {
            port.close();
        }
    });

    var defaultIsolateSendPort = port.toSendPort();

    for (String sourceFileName in dynamicSourceFiles) {
        var dynamicSendPort = spawnUri(sourceFileName);
        dynamicSendPort.send(fileList, defaultIsolateSendPort);
    }
}
```

Закрываем порт,
получив ответы
от всех запущенных
изоляторов

Для каждого
динамически загружаемого
исходного файла...

...запускаем новый изолятор
с таким же именем, что у файла

Отправляем новому изолятору список файлов и SendPort изолятора
по умолчанию, в который он будет посылать результат

Итак, мы поняли, как динамически загружать программные модули, например `fileTypes.dart` и `fileSizes.dart`, и как отправлять им данные. Теперь посмотрим на сами эти модули.

15.2.2. Определение динамически загружаемого исходного файла

В любом Dart-приложении имеется функция `main()`, которая не принимает параметров и возвращает `void`. Это точка входа в главный изолятор. Если в изолятор динамически загружается исходный файл, то его функция `main()` становится точкой входа, с которой начинается исполнение кода в изоляторе. Поэтому в любом динамически загружаемом исходном файле должна быть определена функция `main()`. Это разумно, потому что любой Dart-скрипт работает в изоляторе, быть может, подразумеваемом по умолчанию; на внутреннем уровне изолятор по умолчанию и динамически созданный изолятор ничем не отличаются. На рис. 15.9 схематически представлены изолятор по умолчанию и динамически загружаемые исходные файлы.

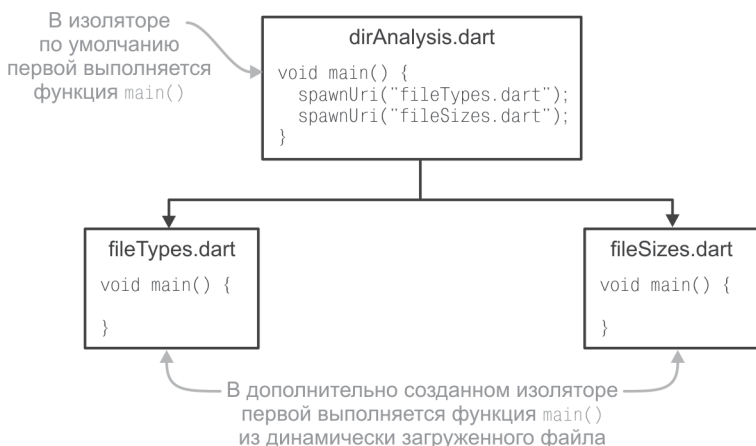


Рис. 15.9. В динамически загружаемом исходном файле должна быть функция `main()`, с которой начинается выполнение

В функции `main()` из динамически загруженного исходного файла мы можем присоединить обработчик к порту приема изолятора `ReceivePort`, доступному через свойство верхнего уровня `port`, — точно так же, как поступали при создании изолятора с помощью функции

`spawnFunction()`. В листинге ниже показана функция `main()` из файла `fileTypes.dart`; она получает список файлов от изолятора по умолчанию, анализирует его и возвращает результат.

Листинг 15.8. Добавление в изолятор обработчика принимаемых данных

```
import "dart:isolate";

void main() {
    ReceivePort receivePort = port;
    receivePort.receive( (data, replyTo) {
        Map<String, int> typeCount = getFileTypes(data);
        replyTo.send(typeCount);
    });
}
// функция getFileTypes(List<String> fileList) опущена
```

Читаем свойство верхнего уровня,
 ← в котором хранится ReceivePort данного изолятора

Присоединяем обработчик

← принимаемых данных

← Анализируем переданный
 список файлов

← Возвращаем результат
 изолятору по умолчанию

Мы реализовали механизм, с помощью которого пользователи смогут подключать свой код анализа файлов к приложению «Анализ каталога», не модифицируя его. Вторым таким модулем анализа является `fileSizes.dart`, а пользователь может написать, к примеру, модуль, который будет вычислять средний размер файлов каждого типа.

Благодаря динамической загрузке кода в изолятор мы получили возможность безопасно исполнять сторонний код, не опасаясь, что он повредит общее внутреннее состояние или будет напрямую читать значения переменных в нашем приложении. Механизм передачи сообщений обеспечивает безопасную связь между участками программы, написанными разными людьми, причем модель параллелизма оказывается проще, чем в многопоточных программах с разделяемым состоянием, как в C# и Java.

Памятка

- Функция `spawnFunction()` динамически загружает исходный Dart-файл в новый изолятор.
 - В динамически загружаемом Dart-файле должна быть определена функция `main()`, с которой начинается выполнение.
-

В следующем разделе мы продолжим знакомство с изоляторами и покажем, как создать несколько одинаковых изоляторов для распределения некоторой работы между несколькими исполнителями.

15.3. Запуск нескольких исполнителей

Часто бывает так, что серверный процесс должен решить какую-то пакетную задачу. Сейчас приложение «Анализ каталога» подсчитывает количество файлов разного типа в одном каталоге, не спускаясь в его подкаталоги. Работающие параллельно изоляторы прекрасно приспособлены для пакетной обработки, в том числе обработки списка каталогов. Каждый каталог в списке не зависит от всех остальных; изолятор по умолчанию хотел бы получить результаты анализа как можно быстрее. Тут-то и приходят на помощь изоляторы-исполнители. Запустив в фоновом режиме несколько исполнителей, мы можем обработать список элементов, передавая очередной элемент освободившемуся исполнителю. На рис. 15.10 показан список подлежащих обработке каталогов.

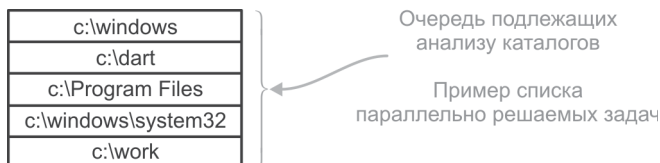


Рис. 15.10. Имеется список задач, которые можно решать параллельно

Это небольшой список, но легко можно представить себе, что требуется проанализировать тысячи каталогов. Невозможно создать новый исполнитель для каждого элемента такого списка – у компьютера просто не хватит ресурсов. Вместо этого мы создадим небольшое фиксированное количество изоляторов-исполнителей и будем передавать каждому по одной задаче. В скрипте «Анализ каталога» таких исполнителей будет три.

Совет. Обычно требуется настройка, чтобы найти оптимальный компромисс между производительностью и потреблением ресурсов. Если изоляторов будет слишком много, то может не хватить памяти, а процессоры будут тратить на переключение между изоляторами столько же времени, сколько на полезную работу. Напротив, если изоляторов окажется слишком мало, то мы не сможем получить заметный выигрыш в производительности. В качестве эвристического правила можно начать с числа изоляторов, равного числу процессорных ядер в компьютере.

В листинге 15.9 показана отправная точка, с которой мы начнем разработку пакетной версии приложения «Анализ каталога». В нем появилась новая функция `getFileListEntryPoint()`, которая играет роль изолятора-исполнителя. Этой функции передается сообщение, содержащее имя обрабатываемого каталога. Она строит список файлов в нем и передает его функции анализа, которая подсчитывает количество файлов разных типов и возвращает результат. Вместе с результатом функция возвращает также свой `SendPort`, в который изолятор по умолчанию пошлет следующий подлежащий обработке данным исполнителем каталог.

Листинг 15.9. `batchDirAnalysis.dart`: отправная точка пакетного приложения

```
import "dart:io";
import "dart:isolate";

void main() {
  Queue<String> foldersToProcess = new Queue<String>();
  foldersToProcess.add(r"c:\windows");
  foldersToProcess.add(r"c:\dart");
  foldersToProcess.add(r"c:\windows\system32");
  foldersToProcess.add(r"c:\Program Files");
  foldersToProcess.add(r"c:\work\");

  // todo: поручить исполнителям обработку каталогов
}

void getFileListEntryPoint() {
  var workerReceivePort = port;
  var workerSendPort = receivePort.toSendPort();

  workerReceivePort.receive( (folderToProcess, defaultReplyTo) {
    print("Обрабатывается каталог: $folderToProcess");

    getFileList(folderToProcess).then( (fileList) {
      var results = countFileTypes(fileList);
      defaultReplyTo.send(results, workerSendPort);
    });
  });
}

List<String> getFileList(folderName) {
  // реализация опущена
```

Создаем список подлежащих обработке каталогов

← Остальная часть `main()` будет реализована в листинге 15.10

Получаем `SendPort` и `ReceivePort` изолятора-исполнителя

Присоединяем ← обработчик принимаемых данных

Получаем список файлов в каталоге и передаем его функции анализа

← Исполнитель отправляет результат изолятору по умолчанию, передавая также свой `SendPort`

```
// возвращает список файлов в каталоге
}

Map<String,int> countFileTypes(fileList) {
    // реализация опущена
    // возвращает количество файлов каждого типа, например:
    // .TXT: 3, .DOC: 10, .PDF: 5
}
```

В изоляторе по умолчанию мы должны запустить фиксированное число изоляторов-исполнителей и отправить каждому задачу – в данном случае имя каталога из очереди ожидающих обработки. Загрузив всех исполнителей работой, изолятор по умолчанию ждет от них результатов, как показано на рис. 15.11.

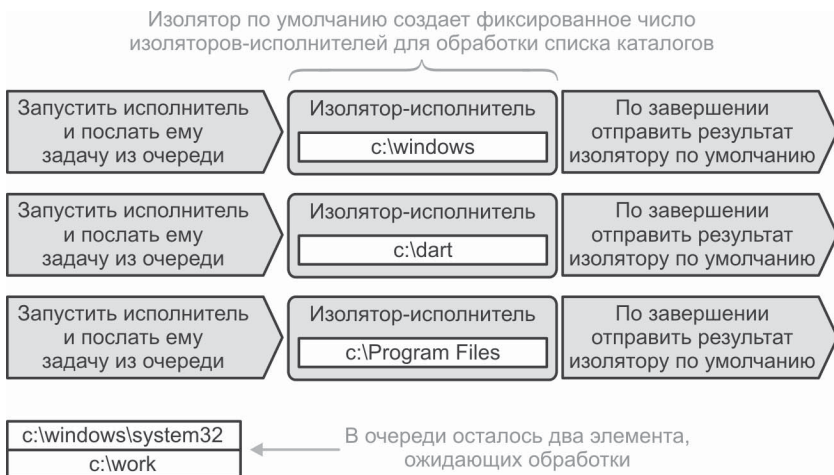


Рис. 15.11. Приложение запускает фиксированное число исполнителей и посылает каждому имя подлежащего обработке каталога

В листинге 15.10 показано, как функция `spawnFunction()` создает три изолятора-исполнителя, после чего им передается начальная задача. Отметим, что мы не храним список исполнителей – достаточно просто запустить их и передать данные, потому что каждый исполнитель возвращает вместе с результатом свой `SendPort`. Посылая исполнителю имя очередного каталога, мы ожидаем от него ответа, чтобы можно было вести учет полученным ответам. Как только

число полученных ответов сравнивается с ожидаемым, изолятор по умолчанию закрывает свой порт приема и дает приложению возможность завершиться.

Листинг 15.10. Запуск изоляторов-исполнителей и передача им начальных задач

```
main() {
    // создание очереди foldersToProcess опущено

    var expectedReplyCount = 0;    ← Переменная для учета пришедших ответов
    var defaultReceivePort = port;
    var defaultSendPort = defaultReceivePort.toSendPort();    ← Получаем SendPort
                                                                и ReceivePort изолятора
                                                                по умолчанию

    for (int i=0; i<3; i++) {    ← Создаем три изолятора-исполнителя
        SendPort worker = spawnFunction(getFileListEntryPoint);    ← Запускаем функцию
                                                                    обработки

        expectedReplyCount++;    ← Увеличиваем счетчик ответов: по одному ответу
                                на каждую отправленную задачу

        var folderName = foldersToProcess.removeFirst();    ← Посылаем исполнителю имя
        worker.send(folderName, defaultSendPort);    ← подлежащего обработке каталога
    }
}
```

Теперь у нас есть три изолятора, занимающихся обработкой каталогов. Последний шаг – изменить изолятор по умолчанию, так чтобы он мог принимать данные от исполнителей и передавать исполнителю следующую задачу, если они еще остались. Именно для этой цели исполнитель возвращает свой `SendPort` в аргументе `replyTo` – чтобы изолятор по умолчанию мог передать ему дополнительные данные. На рис. 15.12 показана эта заключительная последовательность событий.

Этот цикл «возврат результата – отправка новой задачи» продолжается, пока в очереди еще есть задачи. Поскольку код, исполняемый в изоляторе, однопоточный, думать о синхронизации доступа к очереди, как в Java или C#, не нужно. Когда изолятор по умолчанию просматривает очередь и удаляет из нее элемент, никакой другой поток или процесс не обращается к ней. Это одно из основных преимуществ взаимодействия изоляторов путем передачи сообщений: мы получаем защищенную среду выполнения, в которой посторонний код не может ни читать, ни изменять память, так что ошибки из-за одновременного доступа невозможны.

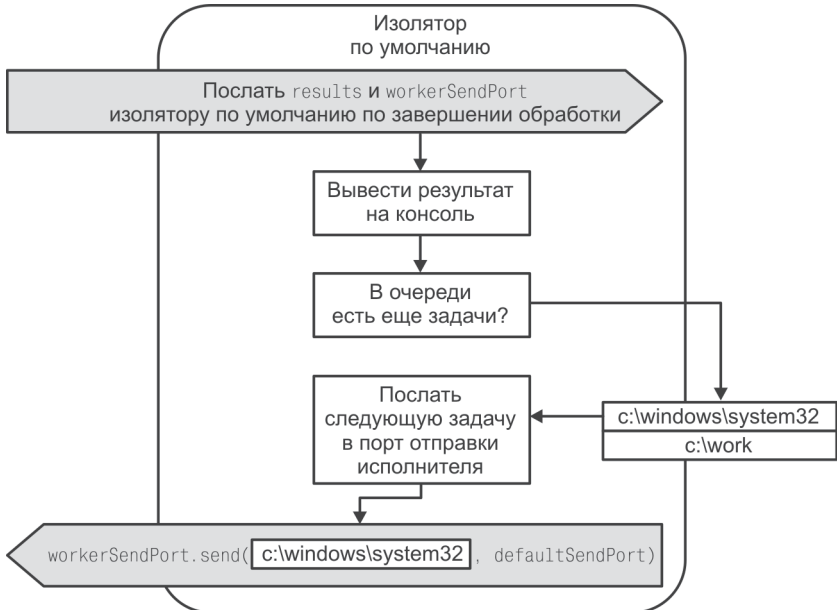


Рис. 15.12. Изолятор по умолчанию получает от исполнителя результаты и отправляет ему следующую задачу

В листинге 15.11 показан последний этап: включение в обработчик `receive()` изолятора по умолчанию кода, который посылает исполнителю новую задачу, если очередь еще не пуста. Если `expectedReplyCount` равно `actualReplyCount`, то все ответы получены, после чего приложение прекращает ждать ответы и вызывает `defaultReceivePort.close()`.

Листинг 15.11. Реализация функции `receive()` в изоляторе по умолчанию

```

main() {
    // создание очереди foldersToProcess опущено

    var expectedReplyCount = 0;

    var defaultReceivePort = port;
    var defaultSendPort = defaultReceivePort.toSendPort();

    for (int i=0; i<3; i++) {
        SendPort worker = spawnFunction(getFileListEntryPoint);

```

```

expectedReplyCount++;

var folderName = foldersToProcess.removeFirst();
worker.send(folderName, defaultSendPoint);
}

var actualReplyCount = 0;

defaultReceivePort.receive( (results, replyToWorker) {
  print(results);
  actualReplyCount++;
  if (foldersToProcess.isEmpty == false) {
    expectedReplyCount++;
    var folderName = foldersToProcess.removeFirst();
    replyToWorker.send(folderName, defaultSendPoint);
  }
  if (actualReplyCount == expectedReplyCount) {
    defaultReceivePort.close();
    print("Конец работы");
  }
});
}

```

← Присоединяем обработчик принятых данных к изолятору по умолчанию

← Выводим полученный результат

← Увеличиваем счетчик ответов

Если в очереди еще есть необработанные каталоги, отправить исполнителю следующий каталог

Когда все ответы получены, закрыть ReceivePort изолятора по умолчанию

Приложение продолжает отправлять исполнителям задачи, пока в очереди остаются необработанные каталоги. В рассмотренной программе очередь фиксирована в начале работы, но ничто не мешает добавлять в нее элементы по ходу дела. Представьте, например, что после анализа каталога возвращается не только результат подсчета файлов, но и список подкаталогов. Тогда мы сможем добавлять новые задачи в конец очереди, пока исполнители забирают задачи из ее начала.

Совет. Мы ожидаем, что изолятор возвращает результат, но этого может и не произойти из-за какой-то внутренней ошибки. Приложение должно уметь справляться и с таким случаем.

Памятка

- Изолятор-исполнитель может вернуть собственный SendPort изолятору по умолчанию, чтобы получать через него дополнительные задачи.
- Изоляторы-исполнители не имеют прямого доступа к очереди работ в памяти изолятора по умолчанию.

- Каждый изолятор может обращаться только к своей собственной памяти, поэтому синхронизировать доступ к изолятору не нужно.
-

15.4. Резюме

Изоляторы в Dart, со своей моделью раздельной памяти и механизмом передачи сообщений, дают более простую реализацию параллелизма, чем в Java и C#, потому что в каждый момент времени к любой ячейке памяти может обращаться лишь один поток. Поскольку несколько изоляторов способны одновременно выполнять несколько задач, то мы получаем возможность в полной мере задействовать потенциал современного оборудования для повышения производительности. А создавая изоляторы-исполнители, мы можем безопасно распределить пакет задач между ними.

При создании изолятора задается точка входа – функция верхнего уровня. Если она находится в уже загруженном Dart-файле, то применяется функция `spawnFunction()`, а если в динамически загружаемом – то `spawnUri()`. Динамическая загрузка исходного файла в изолятор дает возможность создать модель подключаемых модулей, позволяющую пользователям включать в ваше приложение свой код, о котором приложение заранее не знает. Это действенный способ создания расширяемых настраиваемых приложений.

Вот вы и добрались до конца книги. Dart быстро развивается, а это значит, что вам придется учиться и дальше, но базовые принципы языка вы теперь понимаете. Поздравьте себя и принимайтесь за создание замечательных приложений – и не забудьте всем поведать, что они написаны на Dart.



Приложение А. Справочное руководство по языку

В основе своей язык Dart состоит из переменных, операторов, нескольких встроенных типов, предложений управления потоком выполнения и функций. Как и JavaScript, Dart – язык с динамической типизацией, наличие информации о типе в объявлениях переменных и параметров необязательно. Виртуальная машина исполняет Dart-код одинаково вне зависимости от того, есть информация о типах или нет. Но если такая информация имеется, то инструменты могут ей воспользоваться для проверки намерений разработчика, а кроме того, это документация для тех, кто будет читать и сопровождать ваш код.

А.1. Объявления переменных

Имени объявляемой переменной может предшествовать:

- ☐ ключевое слово `var` (означающее, что переменная имеет динамический тип или что никакой информации о типе не предоставлено);
- ☐ имя типа (означающее, что информация о типе предоставлена);
- ☐ ключевое слово `final` (означающее, что это динамическая переменная, доступная только для чтения);
- ☐ ключевое слово `final`, за которым следует имя типа (означающее, что это типизированная переменная, доступная только для чтения).

Динамическая переменная может содержать значение любого типа, типизированная проверяется инструментальными средствами Dart, которые контролируют, что тип значения действительно совпадает с объявленным типом переменной.

Для присваивания переменной значения служит оператор `=`:

```
var myNumber = 123;    ← Переменной myNumber присваивается значение 123
int anInteger = 456;  ← Переменной anInteger присваивается значение 456
final otherNumber = 789;
final int anotherNumber = 246; ← Переменной anotherNumber присваивается значение 246
```

А.1.1. Объявление переменных с ключевым словом `var` или именем типа

Если в объявлении переменной указано ключевое слово `var`, то информация о типе отсутствует. Если вместо `var` указан конкретный тип, например `int`, значит, предполагается, что переменная будет принимать целочисленное значение. Инструментальные средства Dart могут проверить, что переменная содержит целое число, но сам язык этого не гарантирует.

Примечание. Ключевое слово `var` и имена типов можно использовать попеременно, на работу приложения это не влияет.

Вне зависимости от способа объявления — с `var` или с именем типа (например, `int`), переменная не обязана иметь значение. Если значение не указано, то переменная содержит `null`:

<pre>var myUninitializedNumber; var myUninitializedNumber = null;</pre>	Эквивалентны, в обоих случаях присваивается значение <code>null</code>

<pre>int myUninitializedInt; int myUninitializedInt = null;</pre>	В обоих случаях переменная имеет тип <code>int</code> и значение <code>null</code>

Переменная, объявленная с ключевым словом `var` или с именем типа, *изменяема*. Это означает, что ей можно присвоить новое значение:

```
var myNumber = 123;    ← myNumber инициализируется значением 123
myNumber = myNumber + 1; ← Значение myNumber модифицируется путем прибавления 1
myNumber = 456;        ← myNumber присваивается новое значение
int myOtherNumber = myNumber; ← myNumber инициализируется значением другой переменной
```

А.1.2. Объявление финальных (доступных только для чтения) переменных

Вместо ключевого слова `var` можно написать в объявлении ключевое слово `final` само по себе или в сочетании с именем типа. Объявленная таким образом переменная будет доступна только для чтения. Действуют следующие правила:

- ❑ финальная переменная должна быть инициализирована в момент объявления;
- ❑ значение финальной переменной нельзя изменить после объявления.

Например:

```
final myReadOnlyNumber = 123;  ← Объявляется переменная, доступная только для чтения

final myOtherReadOnlyNumber = myReadOnlyNumber;  ← Финальной переменной нельзя
                                                    присвоить значение другой переменной

final int myReadOnlyInteger = 456;  ← Ключевое слово final может сопровождаться
                                     информацией о типе, например int

final myUninitializedVariable;  ← Ошибка: переменная должна быть инициализирована явно

myReadOnlyNumber = 789;  ← Ошибка: изменять значение переменной запрещено
```

А.1.3. Синтаксис литералов

В Dart все типы объектные, примитивных типов не существует. В сам язык встроена поддержка литеральной нотации для пяти типов, то есть возможность указать значение без явного конструирования объекта:

- ❑ Числа — `-123`, `45.6`, `0xAB`
- ❑ Строки — `"Hello"`, `'Dart'`
- ❑ Булевы величины — `true`, `false`
- ❑ Списки — `[123, "Hello", true]`
- ❑ Словари — `{"key1": "value1", "key2": "value2"}`

Во всех остальных случаях необходимо явно конструировать объект с помощью ключевого слова `new`, например:

```
var myVariable = new Object();
var myDateVariable = new Date();
```

Создаются объекты, и ссылки на них
запоминаются в переменных

Числа

Базовый тип чисел — `num`. У него есть подтипы `int` и `double`. Любое число, содержащее десятичную точку, считается принадлежащим типу `double`, остальные числа трактуются как `int`:

- ❑ допускаются целые числа произвольной длины;
- ❑ значения типа `double` представляются 64-разрядным числом с плавающей точкой.

Например:

```
var myNum = 123;
var myInteger = 234;
var myDouble = 123.45;
var myHex = 0xABCD1234;
```

С помощью
ключевого слова `var`
объявляются числа
с литеральными значениями

```
num myNum = 123;
num myDoubleNum = 123.45;
int myInteger = 234;
double myDouble = 123.45
int myHex = 0xABCD1234;
```

В объявлении чисел
с литеральными значениями
указан тип

Строки

Строки в Dart — это последовательности символов Unicode, заключенные в одиночные или двойные кавычки. Если строка заключена в двойные кавычки, то внутри могут находиться символы одиночных кавычек, и наоборот:

- ❑ символ одиночной кавычки может входить в состав строки, заключенной в двойные кавычки;
- ❑ символ двойной кавычки может входить в состав строки, заключенной в одиночные кавычки;
- ❑ для включения управляющей последовательности следует использовать символ `\` (например, символ перехода на новую строку обозначается управляющей последовательностью `\n`);
- ❑ чтобы игнорировать символы управляющих последовательностей, строке следует предпослать префикс `r`.

Например:

```
var mySingleQuotedString = 'Hello Dart';
var myDoubleQuotedString = "Hello Dart";
```

Объявляется
строка

```

var embeddedDoubleQuotes = 'Hello "Dart" World';
var embeddedSingleQuotes = 'Hello 'Dart' World';

var escapedQuote = 'Hello Dart\'s World';
var newLine = "Hello \nDart";
var rawString = r"Hello \nDart";

```

Объявляется строка, содержащая внутри кавычки

← Одиночная кавычка экранируется

← Управляющая последовательность перехода на новую строку: выводится Hello Dart

← Управляющая последовательность перехода на новую строку игнорируется: выводится Hello \nDart

Можно объявлять многострочные строки, поставив с обеих сторон три одиночные кавычки:

```

var myMultiLineString = '''Это
многострочный
строковый литерал''';

```

Для лучшего форматирования первый символ новой строки игнорируется, если в строке нет текста:

```

var myFormattedMultiLineString = '''
    Это
    отформатированный
    многострочный
    строковый литерал''';

```

Строковая интерполяция

Конкатенация строк с помощью оператора + не работает.

```

var foo = "Dart";
var bar = "World";
var foobar = foo + bar;
var helloFooBar = "Hello " + foo + " " + bar;

```

Ошибка: «складывать» строки нельзя

Вместо этого Dart позволяет использовать строковую интерполяцию при объявлении строк. Вместо имени переменной, которому предшествует знак доллара \$, будет подставлено значение этой переменной:

```

var foo = "Dart";
var bar = "World";
var foobar = "$foo$bar";
var helloFooBar = "Hello $foo $bar";

```

← «DartWorld»

← «Hello Dart World»

Если требуется вычислить выражение, например преобразовать значение переменной `foo` в верхний регистр с помощью метода `toUpperCase()` из класса `String`, то выражение следует заключить в блок `${ }`:

```
var foo = "Dart";
var helloFooBar = "Hello ${foo.toUpperCase()}"; ← «Hello DART»
```

У переменной любого типа имеется метод `toString()` (унаследованный от базового типа `Object`), который вызывается неявно, если тип интерполируемой в строку переменной отличается от `String`:

```
var myNumber = 123;
var helloNumber = "Hello ${myNumber.toString()}"; ← «Hello 123»
var helloNumber = "Hello $myNumber"; ← «Hello 123»; toString() вызывается неявно
```

Булевы величины

В отличие от JavaScript, в Dart истинным может быть только значение `true`. Все значения, отличные от литерала `true`, считаются ложными (`false`):

<pre>var myTrueValue = true; bool myTypedTrueValue = true;</pre>	Истинным может быть только значение <code>true</code>
<pre>var myFalseValue = false; bool myTypedFalseValue = false; var myNumber = 1; var myZero = 0; var myObject = new Object(); var myNull = null;</pre>	Любое другое значение не есть <code>true</code> , а значит, равно <code>false</code>

Операторы `==` (равно) и `!=` (не равно) порождают булевское значение — результат сравнения:

```
var isTrue = 2 == (1+1); ← Сохраняется значение true
var isFalse = 123 != 123; ← Сохраняется значение false
```

Примечание. В отличие от JavaScript, в условных предложениях (например, в предложении `if`) допускаются только булевские выражения, то есть `myNumber` или `myObject` нельзя использовать в качестве условного выражения. Необходимо явно сравнить два значения, например: `if (myNumber == 1)` или `if (myObject != null)`. В разделе «Управление потоком выполнения и итерирование» условные предложения рассматриваются более подробно.

Создание литеральных списков (массивов)

В Dart нет явного типа массива, а есть тип `List`, представляющий упорядоченный набор объектов. Тип `List` реализует интерфейс `Collection` — так же, как типы `Set` и `Queue`. Интерфейс `Collection` допускает обход коллекции элементов. Но только для списков определена литеральная нотация объявления. Литеральный список задается в виде перечисления объектов через запятую в квадратных скобках, при этом создается список фиксированной длины:

Список, содержащий три элемента:	
<code>var myList = [123, 'Hello', true];</code>	← число, строка и булевское значение
<code>List otherList = [1,2,3,4,5,6];</code>	← Список, содержащий шесть элементов: все — числа

Элементом списка может быть другой список:

<pre>var listOfLists = [[1,2,3], ['a','b','c','d'], [true, false]];</pre>	Внешний список содержит несколько внутренних

Для доступа к элементам списка применяется индексатор `[]`, нумерация индексов начинается с нуля:

<code>var myList = [123, 'Hello', true];</code>	← Объявляется список
<code>var myNumber = myList[0];</code>	Читаются значения элементов списка с помощью индексатора
<code>var myString = myList[1];</code>	
<code>var myBool = myList[2];</code>	
<code>myList[1] = 'Hello World';</code>	← Обновляется значение элемента списка
 <code>var notAvailable = myList[3];</code> ← Ошибка: в списке нет значения с таким индексом	

Количество элементов в списке возвращает свойство `length`:

<code>int listLength = myList.length;</code>	← Опрашивается длина списка

С помощью литеральной нотации создается список фиксированной длины, уже содержащий ряд объектов. Список фиксированной длины можно создать и с помощью ключевого слова `new`, указав количество элементов. В этом случае все элементы созданного списка будут равны `null`:

```
var myList = new List(3); ← Создается список, содержащий три элемента со значением null
myList[0] = 123;
myList[1] = 'Hello';
myList[2] = true;
```

Элементам присваиваются значения

Добавлять новые элементы в список фиксированной длины нельзя:

```
var myList = new List(3); ← Создается список фиксированной длины
myList.add(123);          ← Ошибка: нельзя добавить в список четвертый элемент
```

Если не указывать количество элементов при создании списка, то получится список переменной длины, в который можно добавлять новые элементы:

```
var myList = new List(); ← Создается список переменной длины
myList.add(123);
myList.add('Hello');
myList.add(true);
print(myList.length);
myList.add(456);
print(myList.length);
```

В него добавляются три элемента

← Длина равна 3

← Добавляется четвертый элемент

← Длина равна 4

Примечание. В разделе «Управление потоком выполнения и итерирование» приведены дополнительные сведения о последовательном доступе к элементам списка.

Литеральный список можно использовать в объявлениях объектов Set и Queue — в виде аргумента конструктора `.from(...)`:

```
var set = new Set.from([123, 'Hello', true]);
var queue = new Queue.from([123, 'Hello', true]);
```

Различия между объектами List, Set и Queue приведены в табл. А.1.

Примечание. Тип List может быть также обобщенным. Дополнительные сведения см. в разделе «Обобщенные словари и списки».

Таблица А.1. Сравнение List, Set и Queue

List	Set	Queue
Литеральная нотация	Нет литеральной нотации	Нет литеральной нотации
Доступ к элементам по индексу возможен	Доступ к элементам по индексу невозможен	Доступ к элементам по индексу невозможен
Фиксированная или переменная длина	Только переменная длина	Только переменная длина
	Не может содержать дубликатов	Располагает методами, позволяющими обращаться как к очереди (addLast(), removeFirst()) или как к стеку (addLast(), removeLast())

Создание literalных словарей

Словарем называется набор пар ключ–значение. Для определения литерального словаря пары "key": "value" перечисляются через запятую и заключаются в фигурные скобки. Для доступа к элементам словаря применяется индексатор, в котором указывается ключ:

```
Map myMap = {"key1": "value1", "key2": "value2"}; ← Создается литеральный словарь
```

```
var value1 = myMap["key1"];
var value2 = myMap["key2"];
```

Значения из литерального словаря читаются по ключу

```
myMap["key2"] = "a new value 2";
```

← Обновляется уже существующее значение
В словарь вставляется

```
myMap["key3"] = "a new key value pair"; ← новая пара ключ-значение
```

Значениями в словаре могут быть произвольные объекты:

```
var aValue = new Object();    ← Создается объект
var myMap = {"key", value};  ← Этот объект записывается в словарь в качестве значения
```

С другой стороны, если при определении словаря применяется литеральная нотация, то ключами должны быть строковые литералы (не переменные). Показанный ниже код некорректен:

```
String myKey = "key";      ← В строковой переменной сохраняется строковый литерал
var myMap = {myKey: "value"}; ← Ошибка: ключ литерального словаря
                             не может быть переменной
```

Это ограничение применяется только к словарям, создаваемым с помощью литеральной нотации. По умолчанию интерфейс `Map` реализован классом `HashMap`. Один раз создав словарь с помощью литеральной нотации, мы затем можем добавлять в него новые ключи, при условии что тип ключа реализует интерфейс `Hashable`; для строк это условие выполняется:

```
var myMap = {"key1": "value1"}; ← Создается литеральный словарь

String myKey = "key2";          ← Создается строковая переменная

myMap[myKey] = "value2";        ← Строковая переменная выступает в роли ключа словаря
```

После первоначального создания литерального словаря в него можно добавлять новые пары ключ–значение и модифицировать существующие. Если присвоить значение существующему ключу, старое значение будет затерто. При попытке прочитать значение несуществующего ключа возвращается `null`:

```
var myMap = {"key1": "value1"}; ← Создается словарь с одним ключом

myMap["key2"] = "value2";       ← Добавляется вторая пара ключ–значение

myMap["key1"] = "a new value1"; ← Существующее значение изменяется

var value3 = myMap["key3"];      ← Чтение несуществующего ключа возвращает null
```

Чтение из словаря может вернуть `null` по двум причинам: потому что именно такое значение хранится или потому что ключ не существует. По счастью, в интерфейсе словаря объявлен метод `containsKey()`, проверяющий существование ключа:

```
var myMap = {"key1": "value1"};

var someValue = myMap["someKey"]; ← Чтение несуществующего значения возвращает null

bool containsSomeKey = myMap.containsKey("someKey"); ← Возвращается false

myMap["someKey"] = null;          ← Добавляется новая пара ключ–значение,
                                  в которой значение равно null

containsSomeKey = myMap.containsKey("someKey"); ← Возвращается true
```

Создать словарь можно также с помощью конструктора, без литеральной нотации. В качестве ключа можно использовать любой объект типа `Hashable`, поскольку конструктор создает пустой словарь. Следующие два предложения создают одинаковые словари, в которые можно добавлять ключи и значения:

```
var myLiteralMap = {};           ← Создается пустой словарь с помощью литеральной нотации
var myConstructedMap = new Map(); ← Создается пустой словарь с помощью конструктора
```

Для доступа к коллекциям ключей и значений можно использовать свойства `keys` и `values`, которые возвращают объекты, реализующие интерфейс `Collection`:

```
var myMap = {"key1": "value1", "key2": "value2"};
var keys = myMap.keys;      ← Доступ к коллекции ключей
var values = myMap.values;  ← Доступ к коллекции значений
```

Примечание. В разделе «Управление потоком выполнения и итерирование» приведены дополнительные сведения о последовательном доступе к ключам и значениям словаря.

А.1.4. Обобщенные списки и словари

В списке и словаре можно хранить произвольные объекты в любых сочетаниях. Например, в список можно поместить строку, целое число и булевское значение. Но в Dart имеется также синтаксис обобщенных типов, который позволяет объявить, что список или словарь должны содержать значения конкретного типа. Информация о типе задается в угловых скобках `< >`.

Ниже определен список, в котором можно хранить только строки (или целые, или булевские и т. д.), и этот список сохранен в динамической переменной:

```
var myStringList = new List<String>(); ← Объявляется список,
var myIntegerList = new List<int>();    который может содержать только строки
var myBoolList = new List<bool>();
```

```
myStringList.add("строка");  ← В список добавляется строка
```

Можно использовать также строго типизированные переменные, чтобы модуль проверки типов мог проконтролировать, что в список добавляются только строки:

```
List<String> myStringList = new List<String>();
```

← С помощью строгой типизации объявляется, что список myStringList будет содержать только строки

```
myStringList.add("строка");
```

← В список добавляется строка

Ошибка: инструментальные средства не позволяют это сделать,

```
myStringList.add(123);
```

← потому что 123 — не строка

Можно также объявить литеральный список, содержащий элементы определенного типа:

```
List<String> myStringList = <String>["item1","item2","item3"];
```

← Объявляется литеральный список, в который инструментальные средства не позволят добавить ничего, кроме строк

Можно объявить также строго типизированный словарь с ключами и значениями определенных типов:

```
Map<String,bool> myMap = new Map<String,bool>();
```

← Объявляется словарь, в котором ключи — строки, а значения — булевские величины

```
myMap["key1"] = true;
```

← Инструментальные средства контролируют соответствие типов

```
myMap["key2"] = false;
```

Наконец, литеральные словари, как и списки, могут быть строго типизированными:

```
Map<String,int> myMap = <String,Int>{"key1":123,"key2":456};
```

← Объявляется литеральный словарь, в который инструментальные средства не позволят добавить ничего, кроме пар «строка — целое число»

А.2. Функции

Функции — это повторно используемые строительные блоки, из которых составляются приложения. Выполнение любого Dart-приложения начинается с функции верхнего уровня `main()`, которая не принимает параметров и ничего не возвращает.

```
main() {
```

← Функция `main()` является точкой входа в приложение

← Отсюда вызываются все остальные функции

```
}
```

Обязательными являются следующие части функции:

- имя функции (если только вы не собираетесь использовать анонимную функцию);

- ❑ список параметров (может быть пустым);
- ❑ тело функции.

Функции в Dart поддерживают следующие возможности:

- ❑ именованные необязательные параметры;
- ❑ необязательные параметры со значениями по умолчанию;
- ❑ присваивание функции переменной;
- ❑ ссылка на функцию по имени (в противоположность вызову функции);
- ❑ лексические замыкания, позволяющие при вызове функции обращаться к значениям, не находящимся в ее области видимости.

Благодаря факультативной типизации в Dart объявления функций могут быть весьма разнообразными. Если указаны типы параметров и возвращаемого значения, то инструментальные средства Dart воспользуются ими для контроля правильности кода; заодно это полезная документация для коллег-разработчиков. Если информации о типах нет, то считается, что задан тип `dynamic`.

Примечание. Считается хорошим тоном явно указывать типы параметров и возвращаемых значений для всех открытых интерфейсов – таких, к которым могут обращаться другие разработчики.

В Dart есть два варианта синтаксической записи функции: длинная – допускающая несколько строк внутри блока `{ }`, и короткая – однострочная, автоматически возвращающая значение выражения в своей единственной строке или `null`, если никакое значение не порождается.

Ниже приведена простейшая функция, записанная в обеих синтаксических формах:

```
( ) => ;    ← Короткая форма без параметров, без имени и без вычисляемого выражения;
              неявно возвращается null
( ) {      ← Длинная форма без параметров, без имени и без возвращаемого значения
}
```

Для вызова функции необходимо указать ее имя и список параметров в круглых скобках:

```
sayHello("Dart");    ← Вызывается функция sayHello(), которой передается параметр "Dart"
```

Функция может – хотя это и необязательно – возвращать одно значение, которое можно присвоить переменной:

```
var greeting = getGreeting("Hello", "Dart");
```

← Результат функции `getGreeting()` возвращается в переменной `greeting`

Если функция не возвращает значения явно, то автоматически возвращается значение `null`:

```
var message = print("Hello Dart");
```

← Переменная `message` содержит `null`, потому что функция `print()` не возвращает значения

А.2.1. Длинная синтаксическая форма

Длинная форма подразумевает, что код функции находится внутри фигурных скобок `{ }`, которые располагаются после имени функции и списка параметров. Ниже показано несколько способов вернуть сумму `a + b` с указанием и без указания типов. Для обозначения возвращаемого значения применяется ключевое слово `return`:

<pre>sum(a, b) { return a + b; }</pre>	Типы параметров и возвращаемого значения не указаны
<pre>dynamic sum(dynamic a, dynamic b) { return a + b; }</pre>	То же, что и раньше, но явно указан тип <code>dynamic</code>
<pre>num sum(num a, num b) { return a + b; }</pre>	Параметры и возвращаемое значение имеют тип <code>num</code>

Если функция не возвращает значения, то в качестве ее типа можно указать `void`:

```
void showSum(a, b) {
  print("a равно $a, b равно $b");
```

← Значение не возвращается, поэтому указан тип `void`

А.2.2. Короткая синтаксическая форма

В короткой форме используется символ `=>`, обозначающий, что возвращается значение следующего за ним выражения. Все показанные ниже функции допустимы, имеют имя `sum` и возвращают сумму `a + b`:

<pre>sum(a,b) => a + b; dynamic sum(a,b) => a + b; dynamic sum(Dyanmic a, dynamic b) => a + b;</pre>	Все три функции эквиваленты, принимают значения любого типа и возвращают их сумму Принимает параметры типа num, ← возвращает значение типа dynamic ← Принимает параметры типа num, возвращает значение типа num

Невозможно сказать, что функция, записанная в короткой форме, возвращает `void`. Связано это с тем, что любая функция в короткой форме возвращает какое-то значение, `null` — если не указано ничего другого:

```
void sum(a,b) => print("a равно $a, a b равно $b");
```

← Ошибка: функция в короткой форме все равно возвращает `null`

А.2.3. Параметры функции

Функция может иметь обязательные и необязательные параметры. Обязательные параметры разделяются запятыми и должны предшествовать необязательным, которые записываются единым блоком, заключенным в квадратные скобки. Если при вызове какой-то необязательный параметр не задан, то он принимает значение `null`:

```
sum(a, b, [c, d]) {
    if (c == null) c = 0;
    if (d == null) d = 0;

    return a + b + c + d;
}
```

← Функция `sum()` с необязательными параметрами `c` и `d`
← Проверяем, передан ли параметр `c`


```
num sum(num a, num b, [num c, num d]) {
    if (c == null) c = 0;
    if (d == null) d = 0;

    return a + b + c + d;
}
```

← Та же функция `sum()` с информацией о типе

Задание значений по умолчанию для необязательных параметров

Если нежелательно, чтобы незадаанный параметр принимал значение `null`, можно указать для него другое значение по умолчанию:

```
sum(a,b, [c=0, d=0]) {
    return a + b + c + d;
}
```

Параметры c и d по умолчанию принимают значение 0

```
num sum(num a,num b, [num c=0, num d=0]) {
    return a + b + c + d;
}
```

Вызов функции с необязательными параметрами

При вызове функции, имеющей необязательные параметры, необходимо сначала указать все обязательные параметры, а затем необязательные — позиционно или по имени:

```
sum(a,b,[c=0,d=0]) => a + b + c + d;  ← определяется однострочная функция
                                     ← с необязательными параметрами c и d,
                                     которые по умолчанию равны нулю
```

```
sum(1, 2);                               ← c и d не заданы, поэтому принимают значения 0
                                     Параметр c задан позиционно и равен 3;
sum(1, 2, 3);                             ← d не задан, поэтому по умолчанию равен 0
sum(1, 2, 3, 4);                         ← c и d заданы позиционно
sum(1, 2, d:4);                           ← d задан по имени, c по умолчанию равен 0
```

A.2.4. Функции как полноценные объекты

Мы можем объявить функцию в области видимости верхнего уровня или в объемлющем блоке, где она может стать замыканием, запоминая другие видимые в том же блоке переменные.

Функции как переменные

Чтобы сослаться на функцию по имени, нужно указать это имя без последующих круглых скобок, обозначающих список параметров:

```
sum(a, b) {
    return a + b;
}
```

Функция sum()
определяется
на верхнем уровне

```
main() {
    var sumFunction = sum;
    var result = sumFunction(1,2);
}
```

← Функция sum() сохраняется в переменной

← Функция sum() вызывается с помощью переменной, в которой хранится ссылка на нее

Можно создать функцию внутри области видимости другой функции:

```
main() {
  sum(a, b) {
    return a + b;
  }
  var result1 = sum(1,2);
  var sumFunction = sum;
  var result2 = sumFunction(1,2);
}
```

Функция `sum()` определяется внутри функции `main()`

← Функция `sum()` вызывается

← Функция `sum()` сохраняется в переменной

← Функция `sum()` вызывается с помощью переменной, в которой хранится ссылка на нее

Сохранить в переменной можно также функцию, не имеющую имени. Такие функции называются *анонимными*:

```
main() {
  var sumFunction = (a, b) {
    return a + b;
  };
  var shorthandSumFunction = (a, b) => a + b;
}
```

Анонимная функция сохраняется в переменной

В переменной сохраняется анонимная функция, записанная в короткой форме

Функции как аргументы функций

Функции можно не только сохранять в переменных, но и передавать в качестве параметров другим функциям:

```
doCalculation(calcFunction, a, b) {
  return calcFunction(a, b);
}
main() {
  sum(a, b) => a + b;
  var multiply = (a, b) => a * b;
  var result1 = doCalculation(sum, 1, 2);
  var result2 = doCalculation(multiply, 1, 2);
}
```

Определяется функция `doCalculation()`, которая принимает другую функцию

← в качестве первого параметра

← Переданная функция используется для возврата значения

← Определяется функция `sum()`

Определяется функция, которая сохраняется в переменной `multiply`

Функция `sum()` передается функции `doCalculation()`

Функция `multiply` передается функции `doCalculation()`

```
var result3 = doCalculation((a, b) => b - a, 1, 2);
}
```

Анонимная функция
← вычитания передается
функции doCalculation()

Функции как возвращаемые значения

Функцию можно также возвращать в качестве значения другой функции:

```
getSumFunction() {
  var sum = (a, b) => a + b;
  return sum;
}

main() {
  var sum = getSumFunction();
  var result = sum(1,2);
}
```

← Объявляем функцию sum()
← И возвращаем ее

← Возвращенное значение сохраняется в переменной sum
← Вызываем функцию sum() с помощью переменной,
в которой хранится ссылка на нее

Функции как замыкания

Функция «замыкает» переменные, объявленные в объемлющей области видимости. В примере ниже в функции `getSumFunction()` объявлена переменная `multiplier` (которая не видна в месте, где вызывается `getSumFunction()`), и на эту переменную есть ссылка в возвращенной функции `sum()`. Функция `sum()` удерживает ссылку на `multiplier`, несмотря на то что функция `getSumFunction()` уже вернула управление и переменная `multiplier` на находится в области видимости:

```
getSumFunction() {
  var multiplier = 100;

  sum (a, b) {
    return (a + b) * multiplier;
  }

  return sum;
}

main() {
  var sum = getSumFunction();
  var result = sum(1,2);
}
```

Функция sum() замыкает переменную multiplier,
которая объявлена вне собственной области
видимости sum()

← Возвращается (1 + 2) * 100 = 300

А.3. Управление потоком выполнения и итерирование

В части управления потоком выполнения Dart очень похож на JavaScript и Java. В нем имеются следующие предложения ветвления:

- ☐ `if/else`;
- ☐ условный оператор `?`;
- ☐ предложение `switch`.

А для обхода коллекций объектов Dart располагает следующими средствами:

- ☐ циклы `for`;
- ☐ циклы `for/in`;
- ☐ циклы `do while`;
- ☐ циклы `while do`;
- ☐ выход из цикла и продолжение цикла;
- ☐ метод словаря `forEach()`.

А.3.1. Ветвление потока выполнения

Программам все время приходится принимать решения. Основной конструкцией для этой цели является предложение `if/else`, но когда требуется записать короткие ветви, применяется условный оператор, а когда ветвей много – предложение `switch`.

Предложения `if/else`

В основном варианте предложения `if` проверяется, равна ли переменная `true` или `false`, и если условие равно `true`, то выполняется блок, следующий за `if`:

```
var someCondition = true;
```

```
if (someCondition) {  
  print("было равно true");  
  print("несколько строчек кода");  
}
```

← Проверяется условие и выполняется код
Если условие было равно `true`,
то выполняется код в этом блоке

Если требуется выполнить всего одну строку кода, то фигурные скобки можно опустить:

```
if (someCondition) print("было равно true");
```

← Печатается "было равно true",
если условие `someCondition` истинно

Вычисление проверяемого условия должно давать булевское значение, в противном случае возбуждается исключение. Например, в следующем фрагменте проверяемое выражение равно `null` – в JavaScript это допустимо, а в Dart приводит к ошибке:

```
var myValue = new Object();
if (myValue) {
    print("значение не равно null");
}
```

← `myValue` – не булевское значение, поэтому возбуждается исключение

В этом случае необходимо явно осуществлять проверку с помощью операторов сравнения, перечисленных в табл. A.2:

```
var myValue = new Object();
if (myValue != null) {
    print("значение не равно null");
}
```

Явно проверяется, что `myValue` не равно `null`,
← при этом возвращается булевское значение `true`

Таблица A.2. Операторы сравнения

Оператор	Пояснение
<code>a == b</code>	Значения равны (не обязательно представляют один и тот же экземпляр)?
<code>a != b</code>	Не равны
<code>a > b</code>	Больше
<code>a >= b</code>	Больше или равно
<code>a < b</code>	Меньше
<code>a <= b</code>	Меньше или равно
<code>a is String</code>	Является экземпляром указанного типа
<code>a is! String</code>	Не является экземпляром указанного типа
<code>identical(a,b)</code>	Один и тот же экземпляр

Предложение может содержать необязательную ветвь `else`, которая выполняется, когда проверяемое выражение равно `false`. Ветвь `else` должна следовать за блоком или предложением, которое выполняется, когда выражение равно `true`:

```
var someCondition = false;
if (someCondition) {
    print("было равно true");
}
else {
    print("было равно false");
}
```

Условие
сравнивается с `true`

Ветвь
else

```
if (someCondition) print("было равно true");  
else print("было равно false");
```

Каждая ветвь if-else
содержит по одному предложению

Примечание. Хотя язык позволяет, считается дурным стилем смешивать в одном предложении if/else блоки, заключенные в фигурные скобки, и однострочные предложения.

Можно сочетать else с if и таким образом записывать цепочку условий:

```
var someValue = 3;  
if (someValue == 1) {  
    print("значение равно 1");  
}  
else if (someValue == 2) {  
    print("значение равно 2");  
}  
else if (someValue == 3) {  
    print("значение равно 3");  
}  
else {  
    print("значение равно $someValue");  
}  
  
if (someValue == 1) print("значение равно 1");  
else if (someValue == 2) print("значение равно 2");  
else if (someValue == 3) print("значение равно 3");  
else print("значение равно $someValue");
```

Цепочка ветвей if / else и else
для проверки нескольких условий
(блочный синтаксис)

Цепочка ветвей if / else
и else (однострочный
синтаксис)

При условии, что конечный результат – булевское значение, можно объединять условия с помощью логических операторов (см. табл. А.3) и круглых скобок.

Таблица А.3. Логические операторы

Логический оператор	Пояснение
	ИЛИ
&&	И
!	НЕ (превращает true в false, а false в true)

Ниже приведены примеры употребления логических операторов и скобок:

```

var value1 = 1;
var value2 = 200;

if (value1 == 1 || value2 == 200) print("true");
if (value1 < 10 && value2 > 100) print("true");
if ( !(value1 > 0= && value1 =< 100) || value2 == 200) {
    print("true");
}

```

value1 равно 1
 ← ИЛИ value2 равно 200
 value1 меньше 10
 ← И value2 больше 100
 ← value1 НЕ между 0 и 100 ИЛИ value2 равно 200

Использование условного оператора для сокращенной записи условных предложений

Часто результатом принятия решения является присваивание того или иного значения одной переменной. Например:

```

var myColor = null;
var myValue = 23;
if (myValue == 23) {
    myColor = "blue";
}
else {
    myColor = "red";
}

```

Присвоить myColor значение, зависящее от значения myValue

Это слишком длинно. Запись можно упростить, воспользовавшись условным оператором `?`, который возвращает одно из двух значений в зависимости от того, чему равно условие: `true` или `false`. Синтаксически этот оператор устроен так:

условие ? значение в случае true : значение в случае false;

Например, приведенный выше код можно переписать следующим образом:

```

var myColor = null;
var myValue = 23;
myColor = (myValue == 23) ? "blue" : "red";

```

Если myValue равно 23, ← вернуть "blue", иначе "red"

Использование предложения *switch* для нескольких сравнений

Ключевые слова `switch` и `case` позволяют сравнить выражение с несколькими значениями. Взгляните на следующую проверку условия, записанную с использованием нескольких предложений `if / else if / else`:

```
var someValue = 3;
if (someValue == 1) {
    print("значение равно 1");
}
else if (someValue == 2) {
    print("значение равно 2");
}
else if (someValue == 3) {
    print("значение равно 3");
}
else {
    print("значение равно $someValue");
}
```

Ее можно переписать с использованием предложения `switch` с несколькими ветвями `case`, в каждой из которых выражение, указанное после слова `switch`, сравнивается с одним значением. Непустые ветви обязаны содержать предложение `break`, обозначающее выход из `switch`:

```
var someValue = 3;
switch (someValue) { ← Сравниваем значение someValue

    case 1: ← Оно равно 1?
        print("значение равно 1"); ← Да, выполняем этот код

        break; ← Выходим из switch, встретив совпадающее значение

    case 2: ← Оно равно 2 (и т. д.)?
        print("значение равно 2");
        break;

    case 3:
        print("значение равно 3");
        break;

    default: ← Выполняем действие по умолчанию, если ни одной подходящей ветви не нашлось
        print("значение равно $someValue");
}
```

Отсутствие предложения `break` в непустой (содержащей хотя бы одно предложение) ветви `case` считается ошибкой. Например:

```
var someValue = 3;
switch (someValue) {
  case 1:
    print("значение равно 1");
  case 2:
    print("значение равно 2");
    break;
}
```

← Существует ветвь, в которой значение равно 1
← Ошибка: отсутствует предложение `break`

Но разрешается располагать подряд несколько пустых ветвей `case` без `break`. Это позволяет сравнивать выражение с каждым из нескольких значений:

```
var someValue = 3;
switch (someValue) {
  case 1:
  case 2:
  case 3:
    print("значение равно 1, 2 или 3");
    break;
}
```

Несколько
пустых ветвей `case`
могут идти подряд

А.3.2. Циклы и итерирование

В Dart имеются четыре способа итерирования на уровне языка, а в словарях есть специальный метод для перебора ключей и значений. Первые три способа – это циклы, выполняемые, пока некоторое условие истинно:

- ☐ циклы `do while`;
- ☐ циклы `while`;
- ☐ циклы `for`.

Поддержка цикла вида `«for in»` предоставляется интерфейсами `Iterable` и `Iterator`, которые реализованы типами коллекций, например: `List`, `Set` и `Queue`.

В словарях имеется метод обхода списка пар ключ–значение – в дополнение к обходу коллекций ключей и значений по отдельности.

Циклы *do while*

В цикле *do while* сначала выполняется некоторый блок, а затем проверяется истинность условия. Это означает, что цикл *do while* выполняется по меньшей мере один раз.

Следующий цикл выполняется ровно один раз:

```
var someCondition = true;
do {
    someCondition = false;
} while (someCondition);
```

Блок, выполняемый
в начале цикла

← После каждой итерации проверяется истинность условия

Можно включить счетчик, который увеличивается на единицу на каждой итерации:

```
var counter = 0;
do {
    counter ++;
}
```

← Счетчик увеличивается на единицу

При необходимости можно выйти из цикла с помощью предложения *break*:

```
var counter = 0;
do {
    counter ++;
    if (counter == 42) {
        break;
    }
} while (counter < 100);
```

← Если счетчик равен 42, выйти из цикла

Циклы *while*

Цикл *while* похож на цикл *do while*, только условие проверяется перед входом в цикл. Это означает, что цикл может не выполниться ни разу:

```
var counter = 0;
while (counter < 100) {
    counter ++;
}
```

← Если условие истинно, выполнить итерацию

Как и раньше, для выхода из цикла можно использовать предложение *break*.

Циклы *for*

В цикле *for* тоже используется счетчик, но конструкция состоит из трех частей: инициализация счетчика, условие и увеличение счетчика:

```
for (int counter = 0; counter < 100; counter++) { ← Заголовок цикла
    print("Hello Dart"); ← Строка "Hello Dart" печатается 100 раз
}
```

Циклы *for* часто применяются для обхода списков, потому что счетчик может выступать в роли индекса списка. Для получения длины списка в нашем распоряжении есть свойство *length*:

```
var myList = ["Dart", "JavaScript", "Java"]; ← Определяется литеральный список
for(int i = 0; i < myList.length; i++) { ← Определяется счетчик, начинающийся
    var language = myList[i]; ← Читается с нуля, и производится столько итераций,
    print(language);           элемент списка какова длина списка
}                               с указанным индексом
```

Как и в циклах *while*, из цикла *for* можно выйти досрочно с помощью предложения *break*. Однако для обхода списка имеется более удобный способ – цикл *for/in*.

Циклы *for/in*

Цикл *for/in* используется совместно с интерфейсами *Iterator* и *Iterable*. Иначе говоря, любой объект, предоставляющий итератор, можно обойти с помощью конструкции *for in*. Идею такого цикла можно выразить фразой «для каждого элемента списка выполнить», а синтаксически это выглядит следующим образом:

```
var myList = ["Dart", "JavaScript", "Java"]; ← Определяется литеральный список
for (var language in myList) { ← Каждый элемент списка поочередно присваивается
    print(language);           переменной language
}                               На каждой итерации используется переменная language
```

Обход словарей с помощью метода *forEach()*

В словарях есть два способа перебрать ключи и значения. Можно получить коллекцию ключей с помощью свойства *keys* и обойти ее, как любой список, получая значения каждого ключа:

```
var myMap = {"key1", "value1", "key2", "value2"}; ← Определяется литеральный словарь
for (var key in myMap.keys) { ← Обходятся все ключи словаря
```

```
var value = myMap[key];    ← Получаем доступ к значению ключа
print(value);             ← Используем значение
}
```

Но в словарях определен также метод `forEach()`, который принимает функцию обратного вызова. Этой функции передаются ключи и значения из каждой хранящейся в словаре пары. В примере ниже методу `forEach()` передается анонимная функция

```
(key, value) {
  print(key);
  print(value);
}
```

Эта функция, которая выделена полужирным шрифтом, получает каждую пару ключ–значение из словаря:

```
var myMap = {"key1", "value1", "key2", "value2"}; ← Определяется литеральный словарь
myMap.forEach( (key, value) { ← Передается анонимная функция, которая
  print(key);    | Ключ и значение           получает поочередно все пары ключ–значение
  print(value);  | используются
});
```



Приложение В. Определение классов и библиотек

В этом приложении рассматриваются вопросы создания классов, использования библиотек и ограничения доступа.

В.1. Классы и интерфейсы

Dart – объектно-ориентированный язык, основанный на классах с одиночным наследованием классов и множественным наследованием интерфейсов. В Dart классы определяются явно, а интерфейсы – неявно. Это означает, что определение класса неявно определяет также интерфейс, состоящий из открытых свойств и методов, который могут реализовать и другие классы тоже.

Примечание. В первоначальной версии Dart существовали явные интерфейсы, определяемые с помощью ключевого слова `interface`. Но самые первые пользователи указали, что определение абстрактного класса определяет также интерфейс, поэтому слово `interface` было сочтено излишним.

В.1.1. Определение классов

Для определения класса служит ключевое слово `class`. Классы определяются только в области видимости верхнего уровня (то есть нельзя определить класс внутри функции, метода или другого класса):

```
class MyClass {  ← Определяется класс
    ← В теле класса определяются свойства и методы
}
```

Для создания экземпляра класса применяется ключевое слово `new`:

```
main() {
  var anInstanceOfMyClass = new MyClass(); ← Создается экземпляр класса MyClass
}
```

Имя класса определяет тип переменной или параметра функции. Указывать имя класса, как и имя встроенного типа (`String`, `int`), необязательно, но если информация о типе присутствует, то инструментальные средства Dart в состоянии проверить правильность кода:

```
class MyClass { ← Определяет класс с именем MyClass

}

void someFunction(untypedParameter, MyClass typedParameter) { ←
  print(untypedParameter is MyClass); | Оба предложения
  print(typedParameter is MyClass); | печатают true
}

main() {
  var anInstanceOfMyClass = new MyClass();
  MyClass aTypedInstance = new MyClass();
  final aFinalInstance = new MyClass();
  final MyClass aTypedFinalInstance = new MyClass();

  someFunction(anInstanceOfMyClass, aTypedInstance);
  someFunction(aTypedInstance, anInstanceOfMyClass);
}
```

MyClass указывается
в качестве типа параметра

Создаются типизированный
и нетипизированный,
финальный и нефинальный
экземпляры MyClass

Результат не зависит от того,
был указан тип переменной
или нет

Свойства

У классов могут быть свойства, соответствующие хранящимся в классе данным. Показанный ниже класс описывает животное, характеризующееся количеством ног и цветом. Количество ног `legCount` строго типизировано, это целое число, а цвет `color` динамически типизирован и может содержать значение любого типа:

```
class Animal {
  int legCount; ← Целочисленное свойство legCount
  var color; ← Динамически типизированное свойство count (см. примечание)
}
```

Примечание. На ранней стадии разработки приложения свойства классов можно оставить динамическими, особенно если вы еще не решили, какой тип свойство будет иметь в конечном итоге. Но по мере продвижения динамические типы лучше заменить строгими, так чтобы инструментальные средства могли проверить правильность программы. В примере ниже показано, почему динамическая типизация полезна на ранних стадиях разработки.

Для обращения к свойствам экземпляра класса применяется точечная нотация, как в Java и JavaScript. Синтаксис чтения и записи свойств одинаков:

```
main() {
  var dog = new Animal();    ← Создается экземпляр класса Animal
  dog.legCount = 4;          ← Свойству legCount присваивается значение;
                              инструменты могут проверить, что значение – целое число
  dog.color = 0xFF;          ← Свойство color может иметь любой тип,
                              в данном случае указано 16-ричное значение

  var bird = new Animal();
  bird.legCount = 2;
  bird.color = "blue";      ← А здесь цвет задан строкой

  var snake = new Animal();
  snake.legcount = 0;
  snake.color = "diamond patterned"; ← Здесь тоже строкой

  var totalLegs = dog.legCount + bird.legCount + snake.legCount; ←

}
```

Читаем значения
свойства всех
объектов-животных

Свойства можно инициализировать начальным значением и делать константными, добавив ключевое слово `final`. Как обычно, строгую и факультативную типизацию можно употреблять interchangeably. У всех экземпляров класса начальные значения будут одинаковы. Если свойство не инициализировано явно, ему по умолчанию присваивается значение `null`, как и переменной:

```
class Animal {
  final isVegetable = false;
  final bool isAnimal = true;
  final bool isMineral = false

  var int legCount = 4;
  var color = "black";

}
```

Финальные свойства нельзя модифицировать, они должны быть инициализированы в момент создания объекта

Другие свойства тоже можно инициализировать, но в дальнейшем разрешается изменять их значения

Примечание. О том, как задавать начальные значения во время выполнения, а не в определении класса, см. раздел, посвященный конструкторам.

Акцессоры чтения и установки

Свойства класса могут быть представлены акцессорами чтения и установки, играющими роль посредников при доступе к стоящему за свойством полю. Это позволяет писать код, исполняемый при попытке доступа к свойству, например код контроля допустимости значения при записи или код генерации значения при чтении. В примере ниже свойству `legCount` соответствует поле `_legCount`, в котором хранятся данные; при чтении и установке `legCount` вызываются акцессоры. Акцессор установки проверяет, что количество ног не меньше нуля:

```
class Animal {
    int _legCount;  ← Поле, стоящее за свойством

    get legCount => _legCount;  ← для возврата значения _legCount

    set legCount(value) {
        if (value < 0) _legCount = 0;
        else _legCount = value;
    }
}
```

В акцессоре чтения использована короткая форма
 Акцессор установки
 принимает значение
 и присваивает его полю `_legCount`

Для акцессоров также можно указывать строгие типы:

```
class Animal {
    int _legCount;

    int get legCount => _legCount;  ← инструментам проверить, что возвращается целое число
    set legCount(int value) => _legCount = value < 0 ? 0 : value;  ←

}
```

Строго типизированный акцессор чтения позволяет
 инструментам проверить, что возвращается целое число
 Строго типизированный акцессор установки позволяет
 инструментам проверить, что свойству присваивается
 целое число

Доступ к свойству через акцессоры в вызывающей программе выглядит точно так же, как доступ к полю. Это означает, что в начале разработки класса можно работать с полями, а затем заменить их акцессорами свойств – больше никакой код менять не придется.

```
main() {
  var dog = new Animal();
  dog.legCount = 4; ← Устанавливается legCount; не важно, что это: поле или акцессор установки
  int totalLegs = dog.legCount; ← Читается legCount; не важно, что это:
}                                     поле или акцессор чтения
```

Включая в класс только акцессор чтения или только акцессор установки, мы можем сделать свойство доступным только для чтения или только для записи:

```
class Animal {
  int _legCount;
  int get legCount => _legCount;
}                                     ← Нет акцессора установки, поэтому свойство можно только читать
```

```
class Animal {
  int _legCount;
  set legCount(int value) => _legCount = value;
}                                     ← Нет акцессора чтения, поэтому свойство можно только записывать
```

Примечание. Знак подчеркивания в имени `_legCount`, означает, что поле закрыто. В Dart ограничения доступа реализованы на уровне библиотеки, а не класса. Дополнительные сведения см. в разделе о библиотеках.

Методы

В классах могут быть определены функции, которые в этом контексте называются *методами*. Методы ведут себя как обычные функции – с тем отличием, что имеют доступ ко всем полям, свойствам и методам класса, в котором определены. Определение метода ничем не отличается от определения функции, если не считать того, что методы определяются в области видимости класса. Если ключевое слово `this` не употребляется явно, то оно подразумевается и является ссылкой на конкретный экземпляр класса:

```
class Animal {
  int legCount;

  void printLegCount() {
    print(legCount);
    print(this.legCount);
  }
}
```

Объявляется метод `printLegCount()`,
 ← который не принимает параметров и не возвращает значения
 ← Печатается значение, хранящееся в свойстве `legCount` текущего экземпляра
 ← Ключевое слово `this` необязательно, но его наличие иногда делает программу понятнее

Для методов, как и для функций, допустима короткая форма:

```
class Animal {  
    int legCount;  
  
    void printLegCount() => print(legCount); ← Короткая форма записи функции  
}
```

Ключевое слово `this` позволяет разрешить конфликт имен, возникающий, когда имя параметра совпадает с именем свойства:

```
class Animal {  
    int legCount; ← Свойство legCount  
  
    void addLegs(int legCount) {  
        this.legCount += legCount; ← В методе addLegs() параметр legCount  
        ← называется так же, как свойство  
        ← Использование this позволяет  
        ← отличить свойство от параметра  
    }  
}
```

Для вызова метода от имени экземпляра класса употребляется точечная нотация — как и для свойств, — но необходимы еще круглые скобки — как и при вызове функций:

```
main() {  
    var dog = new Animal();  
    dog.legCount = 1; ← Устанавливается свойство legCount  
    dog.addLegs(3); ← Вызывается метод addLegs()  
    dog.printLegCount(); ← Вызывается метод printLegCount()  
}
```

Как и в случае функций, отсутствие круглых скобок после имени метода означает, что речь идет о самом объекте-функции, который можно сохранить в переменной:

```
main() {  
    var dog = new Animal();  
    var addLegsMethod = dog.addLegs; ← Метод addLegs сохраняется в переменной  
    addLegsMethod(4); ← Вызывается метод addLegs(4) объекта dog  
}
```

Методы имеют доступ ко всем прочим свойствам и метода данного экземпляра класса:

```
class Animal {
    int legCount;

    void addLegs(int legCount) {
        this.legCount += legCount;

        printLegCount();
        this.printLegCount();
    }

    void printLegCount() => print(legCount);
}
```

Один метод вызывает другой метод,
слово `this` необязательно

К методам применимы те же правила касательно параметров, что и к функциям; параметры могут быть обязательными и необязательными, возможно, со значением по умолчанию. Указывать типы параметров и возвращаемого значения необязательно, но их наличие может рассматриваться коллегами и инструментальными средствами как документация:

```
class Animal {
    String sayHello([String greeting="гав"]) {
        return "$greeting $greeting $greeting";
    }
}

main() {
    var dog = new Animal();
    print(dog.sayHello());

    var bird = new Animal();
    print(bird.sayHello("чирик"));
}
```

Печатается «гав гав гав»

← (используется значение по умолчанию)

← Печатается «чирик чирик чирик»

(используется переданное значение)

Конструкторы

Когда экземпляр класса создается с помощью ключевого слова `new`, вызывается специальный метод, называемый конструктором класса. Имя конструктора совпадает с именем класса. Отсутствие явно определенного конструктора эквивалентно определению пустого конструктора.

Определение

```
class Animal {
    ← Конструктор не определен
}
```

эквивалентно такому:

```
class Animal {
    Animal() { ← Определен
                пустой
            }   ← конструктор
}
```

В конструкторе можно выполнить инициализацию объекта:

```
class Animal {
    int legCount;

    Animal() { ← Конструктор
        legCount = 4; ← устанавливает
    }           значение свойства
}
```

Параметры конструктора

Как и другие методы, конструктор может принимать параметры, в том числе необязательные и со значениями по умолчанию. Этим можно воспользоваться для определения значений свойств на этапе инициализации объекта:

```
class Animal {
    int legCount;

    Animal([int legCount=0]) { ← Необязательный параметр конструктора
        this.legCount = legCount; ← со значением по умолчанию 0
    }                               ← Свойству legCount присваивается значение параметра
}

main() {
    var dog = new Animal(4); ← Значение legCount явно передается конструктору
    var snake = new Animal(); ← Используется значение по умолчанию
}
```

В Dart имеется также полезная короткая форма инициализации свойств в конструкторе. Если в определении параметра конструктора встречается ключевое слово `this`, Dart понимает, что мы хотим установить свойство с указанным именем:

```
class Animal {
  int legCount;

  Animal([this.legCount = 0]) {
    ← Автоматически инициализируется свойство legCount
    ← Конструктор с пустым телом
  }
}

main() {
  var dog = new Animal(4);
  print(dog.legCount);
  ← Печатается 4
}
```

Инициализация в конструкторе

У конструктора Dart может быть также блок инициализации, в котором присваиваются значения финальным свойствам. Финальные свойства должны быть инициализированы до начала выполнения тела конструктора.

Блок инициализации расположен между списком параметров и открывающей фигурной скобкой, за которой идет тело конструктора. Он представляет собой список предложений через запятую, которому предшествует двоеточие. В примере ниже блок инициализации выделен полужирным шрифтом:

```
class Animal {
  final int legCount;
  final bool hasBackLegs;
  final bool hasFrontLegs;

  Animal(frontLegCount, backLegCount) :
    legCount = frontLegCount + backLegCount,
    hasBackLegs = backLegCount > 0,
    hasFrontLegs = frontLegCount > 0 {
    print(legCount);
    print(hasFrontLegs);
    print(hasBackLegs);
  }
}
```

Финальные свойства
нельзя модифицировать
после создания объекта

← Блок инициализации начинается двоеточием

Финальные значения
инициализируются
до входа в тело конструктора

В теле конструктора
финальные свойства
уже имеют значения

Финальные свойства
нельзя модифицировать
после создания объекта

← Блок инициализации начинается двоеточием

Финальные значения
инициализируются
до входа в тело конструктора

В теле конструктора
финальные свойства
уже имеют значения

```

    }
}

main() {
    var dog = new Animal(2,2); ← Код вызывающей программы не изменился
    dog.hasBackLegs = false; ← Ошибка: финальные свойства нельзя изменять
}

```

Именованные конструкторы

В классе может быть несколько именованных конструкторов, помимо конструктора по умолчанию. Например, экземпляр животного можно создать, читая значения свойств из словаря:

```

class Animal {
    int legCount;
    var color;

    Animal() { } ← Конструктор по умолчанию с пустым телом

    Animal.fromMap(Map values) { ← Объявляется именованный конструктор fromMap()
        this.legCount = values["legCount"]; ← Устанавливаются свойства
        this.color = values["color"]; ← на основе значений в словаре
    }
}

main() {
    var dogMap = {"legCount":4,"color":0xFF}; ← Объявляется литеральный словарь
    var dog = new Animal.fromMap(dogMap); ← Объект dog создается с помощью
                                           именованного конструктора

    var snake = new Animal(); ← Объект snake создается с помощью конструктора по умолчанию
    snake.legCount = 0;
    snake.color = "diamond pattern";
}

```

Если конструктор по умолчанию не определен, то для создания экземпляров класса можно использовать только именованные конструкторы. К именованным конструкторам применимы те же правила касательно необязательных параметров и блоков инициализации, что и к конструктору по умолчанию.

Фабричные конструкторы

Фабричный конструктор позволяет классу решить, как получить объект, например не создавать новый, а взять экземпляр из кэша.

Такой конструктор обозначается ключевым словом `factory` и должен возвращать экземпляр класса. Фабричные конструкторы, как и прочие, могут быть именованными. Вызывающая программа обращается к фабричному конструктору, как к любому другому, и даже не знает, что этот конструктор фабричный.

```
class Animal {
    String name;

    factory Animal.withName() {
        var animal = new Animal("fido");
        return animal;
    }

    Animal(this.name) { }

main() {
    var dog = new Animal.withName();
    print(dog.name);
}
```

← Именованный фабричный конструктор
Отвечает за создание и возврат экземпляра класса `Animal`

← Конструктор ко умолчанию

← Конструктор вызывается точно так же, как обычный

Фабричные конструкторы полезны, когда объекты требуется возвращать из кэша:

```
class Animal {
    String name;

    factory Animal.fromCache(String name, [List animalCache = []]) {
        for (var existingAnimal in animalCache) {
            if (existingAnimal.name == name) {
                return existingAnimal;
            }
        }

        return new Animal(name);
    }

    Animal(this.name) { }
}

main() {
    var fido = new Animal.fromCache("fido");
}
```

Фабричный конструктор с необязательным параметром `animalCache`

←

Если животное с таким же именем уже есть в кэше, вернуть его

← В кэше не найдено, создаем новый экземпляр

← Создается новый объект `fido`

```

List animalCache = new List();
animalCache.add(fido); ← fido помещается в кэш

var fido2 = new Animal.fromCache("fido", animalCache); ← Возвращается кэшированная версия fido
print(fido == fido2); ← Печатается true, это один и тот же экземпляр
}

```

В.1.2. Наследование классов

Классы могут образовывать иерархию наследования. Например, у всех животных есть ноги, но у собаки не такие свойства и методы, как у птицы (птицы летают, собаки бегают). Благодаря наследованию мы можем определить базовый класс `Animal`, в котором имеются свойства и методы, общие для всех животных, а затем наследующие ему подклассы с собственными свойствами и методами. Тем самым создается отношение «является» между дочерним и родительским классом, например: `Dog` является `Animal`. Это отношение одностороннее — `Animal` не является `Dog`.

Для расширения существующего класса применяется ключевое слово `extends`:

```

class Animal { ← Определяется базовый класс
    int legCount;

    void eat() {
        print("ням ням ням"); ← Метод базового класса
    }
}

class Dog extends Animal { ← Dog является Animal
    void run() {
        print("Я бегу - вот здорово"); ← Новый метод, специфичный для Dog
    }
}

class Bird extends Animal { ← Bird является Animal
    int wingCount; ← Новое свойство, специфичное для Bird

    void fly() {
        print("Улетаю, за мной гонится собака");
    }
}

main() {

```

```

var dog = new Dog();    ← Создается экземпляр Dog
dog.legCount = 4;      ← Используются свойства и методы,
dog.eat();              | унаследованные от Animal
dog.run();              ← Используется метод, существующий только в классе Dog

var bird = new Bird();  ← Создается экземпляр Bird
bird.wingCount = 2;     ← Используется свойство, существующее только в классе Bird
bird.legCount = 4;      ← Используются свойства и методы,
bird.eat();              | унаследованные от Animal
bird.fly();              ← Используется метод, существующий только в классе Bird
}

```

Благодаря отношению «является» мы можем использовать в качестве строгого типа классы, находящиеся на разных уровнях иерархии:

```

main() {
    Dog fido = new Dog();      ← Указан тип самого
    Bird tweety = new Bird();  ← созданного объекта
    Animal someDog = new Dog(); ← Указан тип
    Animal someBird = new Bird(); ← базового класса

    print(fido is Dog);
    print(someDog is Dog);
    print(fido is Animal);
    print(tweety is Animal);
    print(someDog is Animal);

    print(tweety is Dog);
    print(someBird is Dog);

    Dog someDogAnimal = new Animal(); ← Ошибка: Animal не является Dog
}

```

Наследование полезно и тогда, когда требуется строго типизировать списки, словари и функции или методы:

```

void eatFood(Animal animal) { ← Функция принимает Animal в качестве параметра
    animal.eat();              ← Можно использовать любое свойство или метод базового класса Animal...
    animal.fly();              ← ...но не методы его подклассов
}

main() {

```

```

Dog fido = new Dog();
Bird tweety = new Bird();

eatFood(fido);      | fido и tweety
eatFood(tweety);    | являются Animal

List<Animal> animals = new List<Animal>(); | В список объектов
animals.add(fido);      | типа Animal можно
animals.add(tweety);    | поместить fido и tweety
}

```

Глубина уровня наследования не ограничена. Так, можно определить подклассы Dog, например Poodle (пудель) и Husky (хаски).

Переопределение методов

Подклассы могут предоставлять собственные реализации методов и свойств родительского класса. Например, птица может «тукать» клювом во время еды:

```

class Animal {
    int legCount;

    void eat() {
        print("ням ням ням");
    }
}

class Bird extends Animal {
    int _legCount = 2;
    int get legCount => _legCount;

    void eat() {
        print("тук тук тук");
    }
}

void eatFood(Animal animal) {
    animal.eat();
}

main() {
    var tweety = new Bird();
    eatFood(tweety);
}

```

В базовом классе Animal определена реализация со звуком «ням ням ням»

Переопределяется акцессор
← доступного только для чтения свойства legCount

Переопределенная реализация eat()

Если передан объект Bird, то будет использоваться метод eat() для птиц

← Печатается «тук тук тук»

В.1.3. Абстрактные классы

Иногда нам нужен строгий тип базового класса, но при этом в каждом подклассе реализации могут быть совершенно различны. Например, собаки и птицы едят по-разному, поэтому имеет смысл потребовать, чтобы классы `Dog` и `Bird` предоставляли собственные реализации метода `eat()`. Это можно сделать с помощью абстрактного класса, который не имеет своей реализации, но заставляет определять ее в подклассах. Чтобы показать, что класс или метод является абстрактным, применяется необязательное ключевое слово `abstract`:

```
abstract class Animal {
    int legCount;

    abstract void eat();
}

main() {
    var dog = new Animal();
}
```

← Ключевое слово `abstract` показывает, что разработчик намеревался сделать этот класс абстрактным

← У метода `eat()` нет тела, реализацию должны предоставить подклассы

← Ошибка: нельзя создать экземпляр абстрактного класса

Ключевое слово `abstract` необязательно, следующий класс эквивалентен показанному выше:

```
class Animal {
    int legCount;

    void eat();
}
```

← Ключевое слово `abstract` отсутствует

← Метод не помечен ключевым словом `abstract`, но отсутствие тела означает, что он абстрактный

Если подкласс не предоставляет реализацию абстрактных методов, то он сам считается абстрактным, так что создать его экземпляр тоже невозможно:

```
class Dog extends Animal {
    void eat() => print("хрум");
}

class Bird extends Animal {
}

main() {
    var dog = new Dog();
    var bird = new Bird();
}
```

← Предоставляет реализацию `eat()`

← Не предоставляет реализацию и потому тоже является абстрактным

← Экземпляр `Dog` создать можно

← Экземпляр `Bird` создать нельзя

Абстрактные классы могут содержать какую-то функциональность, но основная их ценность — в том, чтобы заставить реализовать недостающую функциональность в подклассах. Например, класс `Animal` мог бы содержать реализацию метода `sleep()` по умолчанию, но метод `eat()` все равно должен быть реализован в каждом подклассе:

```
class Animal {  
    int legCount;  
  
    void sleep() {  
        print("xppp");  
    }  
  
    void eat();  
}
```

Предоставляется
реализация
`sleep()`

← У `eat()` нет реализации, поэтому класс оказывается абстрактным

В.1.4. Неявные интерфейсы

У каждого класса имеется неявный интерфейс, а именно список всех свойств и методов, которые класс обещает предоставить. Другие программы могут полагаться на это обещание. Особенно это полезно, когда создавать подкласс нежелательно, а предпочтительнее предложить альтернативную реализацию, например подставной класс.

Представим себе стороннюю библиотеку, содержащую класс «корпоративной собаки», которая, чтобы покушать, подключается к серверу. Выглядит он так:

```
class EnterpriseDog {  
    String eat(String connectionString, String food) {  
        // подключиться к базе данных  
        // поесть  
        return foodResult;  
    }  
}
```

Сделать нечто сложное,
полагаясь на внешние
зависимости

Если в своей программе мы пишем метод, который распечатывает результат вызова `eat()`, и хотим его протестировать, то, конечно, можно было бы передать экземпляр `EnterpriseDog`, но тогда потребуется настраивать базу данных и удовлетворить все внешние зависимости:

```

void showFoodResult(EnterpriseDog dog) {
    var result = dog.eat("192.168.2.99:8080", "dogfood");
    print(result);
}

main() {
    showFoodResult(new EnterpriseDog());
}

```

Но можно вместо этого предоставить собственную подставную реализацию интерфейса `EnterpriseDog` в какой-нибудь другой библиотеке, воспользовавшись ключевым словом `implements`:

```

class MockDog implements EnterpriseDog {
    String eat(String connectionString, String food) {
        return "ням";
    }
}

main() {
    showFoodResult(new MockDog());
}

```

Реализация абстрактных классов

Любой класс определяет неявный *интерфейс*, и абстрактные классы — не исключение. Тут четко прослеживается аналогия с определением и реализацией интерфейса в Java или C#:

```

class IEater {
    void eat();
}

class Dog implements IEater {
    void eat() {
        print("хрум");
    }
}

```

Реализация нескольких интерфейсов

Класс может реализовывать несколько интерфейсов. Например, класс `Dog` может реализовывать два интерфейса:

```
class IEater {
    void eat();
}
```

Определяется
абстрактный класс

```
class IRunner {
    void run();
}
```

Определяется
абстрактный класс

```
class Dog implements IEater, IRunner {
    void eat() => print("хрум");
    void run() => print("Я бегу...");
}
```

← Реализует два абстрактных класса
← Реализация IEater
← Реализация IRunner

В.1.5. Статические свойства и методы

У классов могут быть *статические* свойства и методы, которые принадлежат самому классу, а не его экземплярам. Чтобы сделать свойство или метод статическим, необходимо добавить в его определение ключевое слово `static`. Для доступа к статическим свойствам и методам употребляется имя класса, а не имя экземпляра:

```
class Animal {
    static int animalCount = 0;
    static void incAnimalCount() {
        animalCount++;
    }
}
```

← Статическое свойство принадлежит классу в целом,
а не отдельному его экземпляру
← Статический метод может оперировать
статическими свойствами

```
Animal() {
    incAnimalCount();
}
```

Конструктор по умолчанию
вызывает статический метод

```
main() {
    var dog = new Animal();
    print(Animal.animalCount);

    var bird = new Animal();
    print(Animal.animalCount);
}
```

← Обращение к статическому свойству, печатается 1
← Обращение к статическому свойству, печатается 2

В.2. Библиотеки и ограничение доступа

Библиотекой в Dart называется один или несколько связанных вместе файлов. Библиотека является наименьшей единицей доступа. Библиотека может содержать несколько исходных файлов, несколько функций и несколько классов. Библиотека, содержащая функцию `main()`, может использоваться в качестве главного Dart-скрипта.

В.2.1. Определение библиотек

Библиотека определяется с помощью предложения `library`, которое должно находиться в начале файла; именно в нем задается имя библиотеки. Библиотека может состоять из нескольких файлов, содержащих классы и функции, но при этом главный библиотечный файл также может содержать классы и функции. Библиотека – это агрегат, включающий библиотечный файл и связанные с ним частичные исходные файлы.

В библиотеку можно также импортировать другие библиотеки, для чего служит предложение `import`, в котором задается путь к библиотеке. Импортируемые библиотеки содержат код, используемый вашей библиотекой. В листингах В.1, В.2 и В.3 показаны два библиотечных и один исходный файл. Библиотека `my_library`, определенная в файле `my_library.dart` (листинг В.1), включает еще один исходный файл `source.dart` (листинг В.2) и импортирует другую библиотеку `other_library.dart` (листинг В.3).

Листинг В.1. `my_library.dart`

```
library my_library;           ← Определение библиотеки
                                Импортируется другая библиотека
import "path/to/other/other_library.dart"; ← с указанным путем
import "dart:html";          ← Импортируется встроенная в Dart библиотека

part "src/source.dart";      ← Ссылка на исходный файл, являющийся частью библиотеки my_library

class SomeClass {            ← Класс, определенный в my_library
}

void someFunction {          ← Функция, определенная в my_library
```

```
}

main() {
  someFunction();      ← Вызывается функция, определенная в my_library
  someOtherFunctionInMyLibrary(); ← Вызывается функция, определенная в исходном файле,
                                являющемся частью my_library
  someOtherFunctionInOtherLibrary(); ← Вызывается функция, определенная
                                в импортированной библиотеке other_library
}
```

Листинг В.2. source.dart

```
part of my_library; ← Указывает, частью какой библиотеки является этот файл

class SomeOtherClassInMyLibrary {
}

void someOtherFunctionInMyLibrary {
}
```

Функции и классы, определенные в частичном файле, являются такой же частью библиотеки, как функции и классы, определенные в главном библиотечном файле

Листинг В.3. other_library.dart

```
library other_library; ← Определяется other_library

class SomeClassInOtherLibrary {
}

void someFunctionInOtherLibrary {
}
```

Функции и классы принадлежат other_library, но доступны другим библиотекам

В.2.2. Ограничение доступа к элементам библиотеки

Любое имя, начинающееся знаком подчеркивания, считается закрытым на уровне библиотеки. В листинге В.4 показана библиотека `other_library.dart`, которая содержит закрытый класс, закрытую функцию и открытый класс с закрытыми методами и свойствами. Ни один закрытый элемент недоступен импортирующей библиотеке `my_library`.

Листинг В.4. other_library.dart с закрытыми элементами

```

library other_library;

class SomeClassInOtherLibrary {

}

void someFunctionInOtherLibrary {

}

void _aPrivateFunction() {
    print("Внутри закрытой функции");
}

class _SomePrivateClass {
    _SomePrivateClass () {
        print("Внутри конструктора закрытого класса");
    }
}

class SomePublicClass {
    aPublicMethod() {
        print("Внутри открытого метода");
        _aPrivateMethod();
        _aPrivateFunction();
        var aPrivateClass = new _SomePrivateClass();
    }

    _aPrivateMethod() => print("Внутри закрытого метода");

    int _aProperty;
    int get aProperty => _aProperty;
    set(value) => _aProperty = value;
}

```

Закрытая функция недоступна
 ← из внешнего по отношению к библиотеке кода

← Закрытый класс недоступен внешней библиотеке
 ← Закрытый конструктор

← Создается экземпляр закрытого класса
 Определяется закрытый метод

← Определяется закрытое свойство
 Определяются открытые аксессоры чтения и установки для доступа к закрытому свойству



Предметный указатель

Символы

`{ }`, выражение, 32
`$`, знак, 32
`$()`, функция, 301
`+`, конкатенация, 33
`+`, оператор, 249
`<canvas>`, тег, 48
`<script>`, тег, 42, 76, 356
 динамическое
 добавление, 364
`_rev`, свойство, 430
`_` (знак подчеркивания), 37, 156

А

`abstract`, ключевое слово, 180
`action`, поле, 351
`addEventListener()`, метод, 355
`addRequestHandler()`, метод, 395
`add()`, метод, 90, 226, 235, 242
`AppCache`, 368
 переход к локальному
 файлу, когда сетевой
 недоступен, 372
 файл манифеста, 368
 файлы, доступные только
 при наличии подключения
 к сети, 371
 файлы, подлежащие
 локальному
 кэшированию, 370

`AppController`, класс, 292, 333,
414, 421
`app()`, метод, 297
`Array`, тип, 226, 230, 241
`as`, ключевое слово, 226
`AuthError`, класс, 196
`AuthService`, интерфейс, 178,
182, 189
`AuthService`, класс, 177, 180, 189
`auth()`, метод, 177, 199

В

`Bamboo`, 58
`BaseMatcher`, класс, 107
`ButtonElement`, класс, 59, 72,
77, 299

С

`CanvasElement`, 82
`chain()`, метод, 276, 284
`ChartView`, класс, 354, 359
`Chrome`
 Web Store, 349, 373, 376
 BM JavaScript, 55
 установка пакета
 приложения, 373
`close()`, метод, 392, 450
`CoffeeScript`, 52
`Collection`, интерфейс, 226, 230,
233, 252

Completer, объект, 272
console.log(), метод, 62
const, ключевое слово, 194, 196
containsKey(), метод, 343, 473
contains(), метод, 47
CORS (cross-origin resource sharing), 363, 367, 377, 401
CouchDB, база данных, 426, 430
CruiseControl, 58
CSS-класс, 93, 101, 312
CSS-селекторы, 71
CSS-файл, 407

D

dart:html, библиотека, 68, 289, 295, 350, 380, 383
 io, библиотека, 380, 383
 json, библиотека, 238, 252, 295
 реагирование на события
 браузера, 90
 создание элементов, 77
Dart
 архитектура одностраничного приложения, 30
 библиотека io, 386
 виртуальная машина, 51
 запись в файл, 392
 запрос HTML-элементов, 92
 и HTML5, 48
 общие сведения, 27
 перечисление файлов
 и папок, 387
 Редактор, 50
 событийно-управляемая модель, 55
 события браузера, 90
 сходство с другими языками, 29
 чтение файлов, 389

dart2js, инструмент, 52, 63
dart_analyzer, инструмент, 58, 68
dartdoc, инструмент, 66
dartexpense.appcache, файл, 375
DartExpense.html, файл, 303, 355
DartExpense, пример приложения, 288
 обзор, 289
 обработка браузерных событий, 313
 модификация фазы всплытия, 315
 модификация фазы спуска, 316
 типы событий, 317
 передача с сервера, 411
 пользовательский интерфейс, 299
 интерфейс Element, 299
 конструкторы элементов, 303
 обобщенная сетка, 309
 представления, 305
 структура приложения, 293
Dartium, 28, 51
Dart SDK, 59, 67
DataAccess, интерфейс, 292, 345
Date, тип, 338
defaultReceivePort.close(), метод, 461
defaultRequestHandler(), метод, 395
delete(), метод, 387
DivElement, класс, 77, 299
document.body, свойство, 82
document.cookie, свойство, 330, 332
document.head, свойство, 82
document.query(), метод, 301
DOMException, 343

DOM (объектная модель документа), 54
drawChart(), функция, 356
dynamic, тип, 219

E

ElementEvents, интерфейс, 313
Element.html, конструктор, 78, 92, 187, 299, 312
element.on, свойство, 90, 319
element.queryAll(), метод, 92
elements.add(), метод, 305
elements.clear(), метод, 305
element.style, свойство, 84
Element.tag(), конструктор, 80, 187, 299, 312
Element, класс, 74, 77, 81, 313
 конструкторы, 303
 общие сведения, 299
EnterpriseAuthService, класс, 180
EnterpriseUser, класс, 202, 205, 215
EventListenerList, 90, 313
event.stopPropagation(), метод, 315
Event, интерфейс, 317
existsSync(), метод, 387
exists(), метод, 387
expectAsync(), метод, 282
expect(), метод, 106, 109
Expense, класс, 292, 308, 338, 424, 430
expires, свойство, 331
external, ключевое слово, 426

F

factory, ключевое слово, 194
FALLBACK, секция, 372

FileContentHandler, класс, 399, 403
file.exists(), метод, 391, 398
file.openInputStream(), метод, 391
file.readAsString(), метод, 390
filter(), метод, 227, 252
final, ключевое слово, 35
FolderListHandler, класс, 399
forEach(), метод, 252, 338, 482, 490
fromMap(), метод, 340, 345, 500
Function, класс, 135, 140
function, ключевое слово, 88, 114
future.chain(), метод, 275
Futures.wait(), метод, 276, 284
future.then(), метод, 275
Future, класс, 278

G

get, ключевое слово, 183
GET, метод запроса, 393, 429
GitHub, 69, 168
Google Chart, библиотека, 350
Google Chrome, 349, 367, 374
GWT (Google Web Toolkit), 52, 66, 349

H

handler(), метод, 402
hashCode, свойство, 417
hasNext, свойство, 229
HTML5 и Dart, 48
HttpClient.open(), метод, 432
HttpClient класс, сохранение данных, 426
 добавление поддержки сохранения данных, 431

- краткое введение
 - в CouchDB, 427
- совместное использование
 - класса в коде клиента
 - и сервера, 430
- HttpRequest.path, свойство, 397
- HttpRequest, класс, 261
- HttpResponse, объект, 397
- HttpServer, класс, 395, 416, 425
- HTTP-запросы
 - класс HttpServer, 395
 - передача статических файлов, 397

I

- IDL (язык определения интерфейсов), 49
- if/else, предложение, 482
- implements, ключевое слово, 38, 178, 184, 197
- import, предложение, 153, 171
- IndexedDB, 334
- init(), метод, 63
- InputElement, интерфейс, 302
- InputStream, тип, 390
- Iterable, интерфейс, 228, 233
- iterator.next(), метод, 229
- iterator(), метод, 229

J

- JavaScript
 - библиотека для построения графиков, 353
 - отправка данных из Dart
 - в JavaScript, 352
 - отправка данных из JavaScript в Dart, 358
 - получение данных от Dart, 355

- файлы, 52, 61, 66, 70, 150, 294, 367
- JSNI (JavaScript Native Interface), 349
- JSON (JavaScript Object Notation), 223, 238, 335
 - отправка данных
 - REST-клиенту, 402
 - преобразование в объекты Dart, 340
 - преобразование объектов Dart в, 335
 - реализация интерфейса Map, 338
 - функция toMap(), 336
- JSON.parse(), метод, 335, 340, 346
- JSONP (JSON с набивкой), 348, 361, 364
 - динамическое добавление элемента <script>, 364
- JSON.stringify(), метод, 252, 335, 346, 430

L

- library, ключевое слово, 40
- List.add(), метод, 131
- listen(), метод, 396
- List.from(), метод, 232
- ListView.refreshUI(), метод, 308
- ListView, класс, 292, 297, 303, 312
- List, интерфейс, 226, 305
- List, класс, 60
- LocalStorageData, класс, 345
- loglib, библиотека, 149, 152, 157, 162, 165, 171
- lottery, библиотека, 258, 263, 269, 273

M

main(), функция, 42, 130
manifest.json, файл, 374
manifest, атрибут, 368
Map, интерфейс, 223, 237, 251, 344
Map, класс, 335, 346
matcher(), метод, 402
matches(), метод, 107
max-age, свойство, 331
Milestone 1, 336
MongoDB, 334
MouseEvent, тип, 318

N

navigate(), функция, 306, 322, 333, 405
navigator.webkitStartDart(), метод, 62, 294
NETWORK, секция, 371
new, ключевое слово, 185
next(), метод, 229, 235
NoSQL, класс баз данных, 409, 426
noSuchMethod(), метод, 59, 215, 221
num, тип, 467

O

Object, класс, 36, 199
onClosed(), метод, 391, 418
onDataLoaded(), метод, 364
onData(), метод, 390
onDir(), метод, 399
onFile(), метод, 399
onMessage(), метод, 416, 418, 423, 426
on.message, событие, 415, 424
onOpen(), метод, 416

onPopState(), метод, 327, 332
onRequest(), метод, 417, 432, 437
onResponse(), метод, 432, 437
open(), метод, 437
operator, ключевое слово, 252
Options, класс, 384, 452
OutputStream, тип, 392, 426

P

part, ключевое слово, 164, 165
Permission, класс, 225, 234, 242, 245
pipe(), метод, 397
popState, событие, 405
port.receive(), метод, 444
port.toSendPort(), метод, 448, 454
postMessage(), метод, 350
pub install, команда, 104
pubspec.lock, файл, 171
pubspec.yaml, файл, 105, 168
pub, менеджер пакетов, 52
pushState(), метод, 323, 346, 405
putIfAbsent(), метод, 241

Q

queryAll(), метод, 47, 92, 95
query(), метод, 47, 70, 95
queue.removeFirst(), метод, 233

R

Random(), метод, 256, 261, 265
readAsStringSync(), метод, 389
readAsString(), метод, 389, 409
read(), метод, 390
removeFirst(), метод, 233
removeLast(), метод, 472
replyTo, аргумент, 444, 448, 460
requestAnimationFrame(), функция, 266

REST (Representational State Transfer)

- REST-интерфейс, 408
- отправка содержимого каталога в формате JSON, 402
- отправка содержимого файла в формате JSON, 403
- пример пользовательского интерфейса, 404
- return, ключевое слово, 116, 125, 140
- Role, класс, 242, 248, 251

S

- send(), метод, 416, 417, 421, 426, 443, 447
- Set.from(), метод, 232
- setTimeout(), функция, 260
- set, ключевое слово, 183
- Socket, тип, 426
- some(), функция, 228
- spawnFunction(), метод, 440, 448, 454, 456, 459, 463
- spawnUri(), метод, 454, 463
- SQL, база данных, 334
- src, свойство, 364
- StateError, исключение, 229
- StaticFileHandler, класс, 398
- stopImmediatePropagation(), метод, 316
- stopPropagation(), метод, 315
- StringBuffer, тип, 391, 432
- String.fromCharCode(), метод, 390
- stringify(), метод, 338
- string.split(), метод, 332
- String, тип, 390, 469
- substring(), функция, 402
- switch, предложение, 486

T

- test(), функция, 281
- then(), метод, 273, 282
- this, ключевое слово, 195
- toGMTString(), функция, 331
- toJson(), метод, 341
- toMap(), метод, 336
- toString(), метод, 32, 199, 213, 338, 391
- toUpperCase(), метод, 469
- transform(), метод, 278, 284
- trueIfNull(), функция, 35
- typedef, ключевое слово, 135, 140

U

- uiElement, свойство, 100, 102
- UnimplementedError, исключение, 338
- unittest, библиотека, 105
- useHtmlConfiguration(), метод, 104, 108, 281
- User, класс, 176, 226, 242, 244, 249
- useVmConfiguration(), метод, 104

V

- validate(), метод, 215
- var, ключевое слово, 98, 465

W

- wait(), метод, 276, 285
- WebGL, 48
- WebKit IDL, 49
- WebSocketConnection, объект, 416, 417
- WebSocketHandler, класс, 416

WebSocketStatus, класс, 419
WebSocket, класс, 425
Web Storage, 334
 преобразование JSON-строк
 в Dart-объекты, 340
 преобразование объектов
 Dart в формат JSON, 335
 реализация интерфейса
 Map, 338
 функция toMap(), 336
 реализация, 345
 сохранение данных в, 343
 чтение данных из, 344
window, объект, 350
 addEventListener(),
 метод, 355
 addListener(),
 метод, 360
 localStorage свойство, 342
 location.href, 351
 navigator.onLine,
 свойство, 371
 postMessage(), метод, 352,
 358, 361
 sessionStorage свойство, 341
 setTimeout(), метод, 261, 265
write, метод, 392

Y

YAML-файл, 168

A

Абстрактные классы
 наследование, 206
 общие сведения, 180, 505, 507
Автозавершение
в Редакторе, 60
Автоматизация прогона
тестов, 107

Автономное тестирование, 103
 асинхронное
 программирование, 280
 тестирование будущих
 значений, 283
 тестирование функций
 обратного вызова, 282
определение ожидаемых
результатов, 106
создание пользовательского
сравнителя, 107
создание тестов, 105
Аксессуары чтения и установки,
182, 494
Анализ каталога,
приложение, 440
Аргументы командной строки
доступ, 384
Асинхронное
программирование, 254
 setTimeout(), функция, 260
автономное
тестирование, 280
 будущих значений, 283
 функций обратного
 вызова, 282
важность
для веб-приложений, 256
задержки, 260
использование будущих
значений, 271
 ожидание получения
 будущих значений, 276
 передача будущих
 значений, 274
 преобразование обычных
 значений в будущее, 278
упорядочение
асинхронных вызовов
путем сцепления будущих
значений, 275

использование обратных вызовов, 263
 вложенность как средство управления порядком выполнения, 269
 добавление асинхронных обратных вызовов в приложение, 265

Б

Базы данных, взаимодействие с, 426
Библиотека для построения графиков
 Google, 350
Библиотеки, 142
 импорт, 147
 в Редакторе, 69
 использование импортированных классов, 152
 использование импортированных функций, 149
 одной библиотеки в другую, 149
 предотвращение конфликта имен, 153
 и скрипты, 171
 область видимости, 40
 ограничение доступа, 510
 ограничение доступа в классах, 157
 доступ к закрытым полям, 157
 закрытые классы, 159
 закрытые методы, 158
 определение, 145, 509
 организация, 163
 упаковка, 168

Библиотечный файл, 165
Браузерные приложения, не требующие сервера, 367
 подготовка пакета для установки в Chrome, 373
 применение AppCache для запуска в автономном режиме. *См.* AppCache

Браузеры

веб-сокеты
 отслеживание отключений и ошибок, 418
 учет подключений, 417
навигация, 323
 добавление элементов в историю, 324
 реагирование на события перехода, 326
обработка событий, 313
 модификация фазы всплытия, 315
 модификация фазы спуска, 316
 типы событий, 317

Будущие значения

в асинхронном программировании, 271
 ожидание всех, 276
 передача будущих значений, 274
 преобразование обычных значений в будущее, 278
 упорядочение асинхронных вызовов путем сцепления, 275
тестирование асинхронных функций обратного вызова, 282

Булевские величины,

объявление переменных, 469

В

- Веб-сокеты, 411
 - межбраузерная синхронизация, 419
 - отправка данных из браузера, 421
 - получение данных в других браузерах, 424
 - получение данных от браузера, 423
 - обработка запроса, 417
 - отслеживание отключений и ошибок, 418
 - соединение на стороне клиента, 414
 - учет подключений, 417
- Виджетов библиотеки, 299
- Виртуальная машина Dart, 51
- Вложенные обратные вызовы, 269, 275, 285
- Внедрение зависимости, 178
- Внешние серверы, взаимодействие, 361
 - использование JSONP, 364
 - ограничения безопасности, 363

Д

- Данные
 - и навигация в браузере, 321
 - добавление элементов в историю, 324
 - реагирование на события перехода, 326
 - использование веб-хранилища, 334
 - вставка данных, 343
 - преобразование JSON-строк в Dart-объекты, 340

- преобразование объектов Dart в формат JSON, 335
- реализация, 345
- чтение данных, 344
- использование куков, 329
- сохранение данных в, 330
- чтение данных из, 332
- Динамическая загрузка кода в изолятор, 452
 - определение загружаемого исходного файла, 455
 - создание изолятора для загружаемого файла, 454
- Документация, в Редакторе, 66

З

- Задержки, и асинхронное программирование, 260
- Замыкания, 137
 - общие сведения, 112
 - функции как, 481
- Запрет редактирования, 421
- Запутывание кода, 155

И

- Изоляторы, 27, 45, 438
 - двусторонняя связь, 446
 - заккрытие порта приема, 450
 - использование аргумента replyTo, 448
 - отправка и получение данных, 448
- динамическая загрузка кода, 452
 - определение загружаемого исходного файла, 455
 - создание изолятора для загружаемого файла, 454

- запуск нескольких исполнителей, 457
- односторонняя связь, 442
 - отправка сообщений, 443
 - прием данных, 443
- параллелизм, 45
- создание, 440
- Инверсия управления, 178
- Инструменты
 - dart2js, 51
 - pub, менеджер пакетов, 52
 - браузер Dartium, 51
 - виртуальная машина Dart, 51
 - встроенные в Редактор, 58
 - Редактор Dart, 50
- Интерполяция строковая, 32, 468
- Интерфейсы, 175, 506
 - абстрактные классы, 507
 - константные классы, 194
 - ключевое слово const, 196
 - конструктор, 195
 - свойства, 195
 - конструирование, 184
 - несколько
 - конструкторов, 187
 - статические методы и свойства, 191
 - фабричные
 - конструкторы, 187
 - несколько, 181, 507
 - объявление аксессуаров свойств, 182
 - программирование относительно, 177
 - расширение с помощью наследования, 200
 - включение абстрактных классов в иерархию, 206
 - наследование
 - конструкторов, 203
 - переопределение методов и свойств, 205
 - явное определение, 180
- История навигация, добавление элементов, 324
- Итерирование. См. Циклы
- классы, 175
 - абстрактные, 505
 - аксессуары чтения и установки, 182, 494
 - импортированные, использование, 152
 - инкапсуляция структуры и функциональности, 95
 - использование на стороне клиента и сервера, 430
 - константные, 194
 - const, ключевое слово, 196
 - конструктор, 195
 - свойства, 195
- конструкторы
 - именованные, 500
 - инициализация в, 499
 - несколько, 187
 - общие сведения, 185, 497
 - параметры, 498
 - статические методы и свойства, 191
 - фабричные, 187, 500
- методы, 495
- наследование, 502
- обобщенные, 242
 - использование, 244
 - ограничение на типы, 245
 - определение, 242
- ограничение доступа, 157

программирование
относительно
интерфейса, 177
расширение с помощью
наследования, 200
 включение абстрактных
 классов в иерархию, 206
наследование
 конструкторов, 203
 общие сведения, 201
реализация нескольких
интерфейсов, 181
статические свойства
и методы, 508
явное определения
интерфейса, 180

К

Ключ-значение, пары, 353
Коллекции
 множества, создание, 232
 наложение ограничений
 с помощью обобщенных
 типов, 233
 объектов, 226
 методы, 227
 обход, 228
 обход вручную, 229
 очереди, создание, 233
 словари, 237
 вставка элементов, 240
 доступ, 239
 списки, создание, 230
Константные классы.
См. Классы, константные
Конструкторы. См. Классы,
конструкторы
Конфликты имен
в библиотеках,
предотвращение, 153

Куки, 329
 извлечение значения, 332
 просмотр, 331
 свойство cookie, 330
 сохранение данных в, 330
 чтение данных из, 332

Л

Локальное хранение,
технологии, 334
Локальные функции.
См. Функции, локальные
Лотерея Dart, приложение, 255,
260, 278

М

Манифеста файл,
AppCache, 368
Межбраузерная синхронизация
с помощью веб-сокетов, 419
 отправка данных
 из браузера, 421
 получение данных
 в других браузерах, 424
 получение данных
 от браузера, 423
Методы
 закрытые, 158
 классов, 495
 коллекций, 227
 переопределение, 205, 504
 статические, 191, 508
Минимизация, 155
Множества, создание, 232

Н

Наследование
 включение абстрактных
 классов в иерархию, 206

конструкторов, 203
переопределение методов
и свойств, 205
расширение классов
с помощью, 200
Неблокирующий
ввод-вывод, 389, 394

О

Область видимости
библиотеки, 40
Обобщенная сетка, 309
Обобщенные классы
использование, 244
ограничение на типы, 245
определение, 242
Обобщенные списки
и словари, 474
Обратные вызовы, 263, 364
использование
в асинхронном
программировании, 263
тестирование, 282
Обтрясывание деревьев, 291
Объекты, 210
коллекции. См. Коллекции,
объектов
метод toString(), 213
перехват обращений
к несуществующим методам
и свойствам, 218
проверка, 210
функциональность класса
Object, 218
Объявление анонимной
функции
для локальных функций, 130
общие сведения, 228
Ожидаемые результаты теста,
определение, 106

Операторы сравнения
перегрузка, 247
Отладка в Редакторе, 66
Очереди, создание, 233

П

Параметры
конструктора, 498
функции, 119, 478
значения
по умолчанию, 478
необязательные, 479
передача по ссылке, 120
позиционные, 121
Перегрузка операторов, 246
доступа по индексу, 249
предостережение, 249
сравнения, 247
Переменные, 464
булевские, 469
ключевое слово var, 465
обобщенные списки
и словари, 474
словари, 472
списки, 470
строки
интерполяция, 468
общие сведения, 467
финальные, объявление, 466
функции как, 479
числовые, 467
Перечисления, 306
Подразумеваемый интерфейс,
определение, 38
Подставные объекты, 178, 291
Подчеркивания знак, 37, 99,
155
Пользовательский интерфейс
(ГИП), 75

- добавление элементов
 - в документ, 82
 - и интерфейс Element, 299
 - конструкторы
 - элементов, 303
 - начальный HTML-файл, 76
 - обобщенная сетка, 309
 - представления, 305
 - создание элементов
 - из фрагмента HTML, 78
 - создание элементов по имени тега, 80
 - создание элементов с помощью dart
 - html, 77
 - Правило одного домена, 363
 - Преобразование
 - JSON-строка
 - в Dart-объекты, 340
 - объектов Dart в формат JSON, 335
 - реализация интерфейса Map, 338
 - функция toMap(), 336
 - Префикс импорта, 153
 - Приложения, разработка
 - инкапсуляция с помощью классов, 95
 - методы чтения и установки, 99
 - структура класса, 97
 - пользовательский интерфейс, 75
 - добавление элементов в документ, 82
 - создание элементов из фрагмента HTML, 78
 - создание элементов по имени тега, 80
 - события браузера, 86
 - запрос HTML-элементов, 92
 - нажатие кнопки, 86
 - рефакторинг прослушивателя для повторного использования, 91
 - синтаксис функций, 87
 - Проклятье обратных вызовов, 270, 275, 285
- Р**
- Редактор Dart, 50
 - Dart-файлы, 61
 - HTML-файлы, 61
 - автозавершение, 60
 - встроенные инструменты, 58
 - генерация документации, 66
 - динамическое добавление элементов на страницу, 71
 - доступ к элементам DOM, 70
 - запуск приложения, 62
 - импорт библиотек, 69
 - использование dart2js, 63
 - отладка, 66
 - Рекурсивная функция, 130
- С**
- Свойства
 - классов, 182, 492
 - константных классов, 194
 - переопределение, 205
 - статические, 191, 508
 - Серверные скрипты, 381
 - Синтаксическая глазурь, 88
 - Синхронизация, механизм, 421
 - Синхронная модель, 254

Словари

- вставка элементов, 240
- доступ, 239
- обобщенные, 474
- объявление переменных, 472
- создание, 238

События браузера, 86

- запрос HTML-элементов, 92
- нажатие кнопки, 86
- рефакторинг прослушивателя для повторного использования, 91
- синтаксис функций, 87

Совместное**редактирование, 421****Соглашения****об именовании, 156****Создание элементов с помощью dart:html, 77****Сохранение данных**

- в куках, 330
- в локальном хранилище, 343
- с помощью класса HttpClient, 426

Списки

- обобщенные, 474
- объявление переменных, 470
- создание, 230

Сравнитель, создание, 107**Статические файлы, передача по HTTP, 397****Строгий тип функции, 134****Строки**

- интерполяция, 32, 468
- общие сведения, 467
- объявление, 79

Сцепление будущих значений, 275**Т****Типы**

- ограничения в обобщенных классах, 245
- факультативные, 34

Точка входа в приложение, 76**Точки останова в Редакторе, 66****У****Управление потоком выполнения**

- предложение if/else, 482
- предложение switch, 486
- условный оператор, 485

Условный оператор, 485**Ф****Фабричные****конструкторы, 39, 187, 500****Фаза всплытия события****браузера, 315****Фаза спуска события****браузера, 316****Файлы**

- запись, 392
- перечисление, 387
- чтение, 389

Финальные свойства, 195**Функции, 113**

- длинная синтаксическая форма, 477
- для обработки событий браузера, 87
- закрытые, 162
- использование импортированных, 149
- как полноценные объекты, 43, 479



- как аргументы функций, 480
- как возвращаемые значения, 481
- как замыкания, 481
- как переменные, 479
- короткая синтаксическая форма, 477
- локальные, 128
 - объявление анонимной функции, 130
 - объявление именованной функции, 132
 - объявление простой функции, 129
- область видимости, 128
- определение строгого типа, 134
- параметры, 119, 478
 - значения
 - по умолчанию, 478
 - именованные, 123
 - необязательные, 479
 - передача по ссылке, 120
 - позиционные, 121
- тип void, 118
- тип возвращаемого значения, 116

Ц

Цикл обработки событий, 382, 384, 394, 396

Циклы

- do while, 488
- for, 489
- forEach, 489
- for in, 489
- while, 488

Ч

Частичные файлы, 165, 295, 298

Э

Элементы

- добавление
 - в HTML-документ, 82
- запрос с помощью dart:html, 92
- создание из фрагмента HTML, 78
- создание по имени тега, 80
- создание с помощью dart:html, 77

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «АЛЬЯНС БУКС» наложенным платежом, выслав открытку или письмо по почтовому адресу: 123242, Москва, а/я 20 или по электронному адресу: **orders@alians-kniga.ru**.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: **www.alians-kniga.ru**.

Оптовые закупки: тел. (499) 725-54-09, 725-50-27; электронный адрес **books@alians-kniga.ru**.

Крис Баккет

Dart в действии

Главный редактор *Мовчан Д. А.*
dm@dmk-press.ru

Корректор *Синяева Г. И.*

Верстка *Чаннова А. А.*

Дизайн обложки *Мовчан А. Г.*

Подписано в печать 16.04.2013. Формат 60×90 1/16.

Гарнитура «Петербург». Печать офсетная.

Усл. печ. л. 33. Тираж 300 экз.

Веб-сайт издательства: www.dmk-press.ru