Иллюстрированный самоучитель по Java

Пакеты и интерфейсы

Введение

- **Что такое Java**
  Книга, которую вы держите в руках, возникла из курса лекций, читаемых автором в течение последних лет для студентов младших курсов. Подобные книги рождаются после того, как студенты в сотый раз зададут один и тот же вопрос, который лектор уже несколько раз разъяснял в разных вариациях.
- **Структура книги**
  Книга состоит из четырех частей и приложения. | Первая часть содержит три главы, в которых рассматриваются базовые понятия языка. По прочтении ее вы сможете свободно разбираться в понятиях объектно-ориентированного программирования и их реализации на языке Java, создавать свои объектно-ориентированные программы, рассчитанные на консольный ввод/вывод.
- **Выполнение Java-программы**
  Как вы знаете, программа, написанная на одном из языков высокого уровня, к которым относится и язык Java, так называемый исходный модуль ("исходник" или "сырец" на жаргоне, от английского "source"), не может быть сразу же выполнена. Ее сначала надо откомпилировать, т. е.
- **Что такое JDK. Что такое JRE.**
  Набор программ и классов JDK содержит: | компилятор javac из исходного текста в байт-коды; | интерпретатор java, содержащий реализацию JVM; | облегченный интерпретатор jre (в последних версиях отсутствует); | программу просмотра апплетов appietviewer, заменяющую браузер; | отладчик jdt>;
- **Как установить JDK**
  Набор JDK упаковывается в самораспаковывающийся архив. Раздобыв каким-либо образом этот архив: "выкачав" из Internet, с http://java.sun.com/produets/jdk/ или какого-то другого адреса, получив компакт-диск, вам остается только запустить файл с архивом на выполнение.
- **Как использовать JDK**
  Несмотря на то, что набор JDK предназначен для создания программ, работающих в графических средах, таких как MS Windows или X Window System, он ориентирован на выполнение из командной строки окна MS-DOS Prompt в Windows 95/98/ME или окна Command Prompt в Windows NT/2000.
- **Интегрированные среды Java**
  Сразу же после создания Java, уже в 1996 г., появились интегрированные среды разработки программ для Java, и их число все время возрастает. Некоторые из них являются просто интегрированными оболочками над JDK, вызывающими из одного окна текстовый редактор, компилятор и интерпретатор.

-
- **Особая позиция Microsoft**
  Вы уже, наверное, почувствовали смутное беспокойство, не встречая название этой фирмы. Дело в том, что, имея свою операционную систему, огромное число приложений к ней и богатейшую библиотеку классов, компания Microsoft не имела нужды в Java.
-
- **Java в Internet**
  Разработанная для применения в сетях, Java просто не могла не найти отражения на сайтах Internet. Действительно, масса сайтов полностью посвящена или содержит информацию о технологии Java. Одна только фирма SUN содержит несколько сайтов с информацией о Java:
-
- **Литература по Java**
  Перечислим здесь только основные, официальные и почти официальные издания, более полное описание чрезвычайно многочисленной литературы дано в списке литературы в конце книги. | Полное и строгое описание языка изложено в книге The Java Language Specification, Second Editio

Что такое Java

Книга, которую вы держите в руках, возникла из курса лекций, читаемых автором в течение последних лет для студентов младших курсов. Подобные книги рождаются после того, как студенты в сотый раз зададут один и тот же вопрос, который лектор уже несколько раз разъяснял в разных вариациях. Возникает желание отослать их к какой-нибудь литературе. Пересмотрев еще раз несколько десятков книг, использованных при подготовке лекций, порывшись в библиотеке и на прилавках книжных магазинов, лектор с удивлением обнаруживает, что не может предложить студентам ничего подходящего. Остается сесть за стол и написать книгу самому. Такое происхождение книги накладывает на нее определенные особенности. Она:

- представляет собой сгусток практического опыта, накопленного автором и его студентами с 1996 г.;
- содержит ответы на часто задаваемые вопросы, последние "компьютерщики" называют**FAQ (Frequency Asked Questions);**
- написана кратко и сжато, как конспект лекций, в ней нет лишних слов (за исключением, может быть, тех, что вы только что прочитали);
- рассчитана на читателей, стремящихся быстро и всерьез ознакомиться с новинками компьютерных технологий;
- содержит много примеров применения конструкций Java, которые можно использовать как фрагменты больших производственных разработок в качестве "How to?";
- включает материал, являющийся обязательной частью подготовки специалиста по информационным технологиям;
- не предполагает знание какого-либо языка программирования, а для знатоков выделяются особенности языка Java среди других языков;
- предлагает обсуждение вопросов русификации Java.

Прочитав эту книгу, вы вступите в ряды программистов на Java – разработчиков технологии начала XXI века.

Если спустя несколько месяцев эта книга будет валяться на вашем столе с растрепанными страницами, залитыми кофе и засыпанными пеплом, с массой закладок и загнутых углов, а вы начнете сетовать на то, что книга недостаточно полна

и слишком проста, и ее содержание тривиально и широко известно, тогда автор будет считать, что его скромный труд не пропал даром.

Ну что же, начнем!

Это остров Ява в Малайском архипелаге, территория Индонезии. Это сорт кофе, который любят пить создатели Java (произносится "Джава", с ударением на первом слоге). А если серьезно, то ответить на этот вопрос трудно, потому что границы Java, и без того размытые, все время расширяются. Сначала Java (официальный день рождения технологии Java – 23 мая 1995 г.) предназначалась для программирования бытовых электронных устройств, таких как телефоны. Потом Java стала применяться для программирования браузеров – появились **апплеты**. Затем оказалось, что на Java можно создавать полноценные приложения. Их графические элементы стали оформлять в виде компонентов – появились**JavaBeans, с** которыми Java вошла в мир распределенных систем и промежуточного программного обеспечения, тесно связавшись с технологией CORBA.

Остался один шаг до программирования серверов – этот шаг был сделан – появились**сервлеты** и **EJB** (**Enterprise JavaBeans**). Серверы должны взаимодействовать с базами данных – появились драйверы **JDBC** (**Java DataBase Connection**). Взаимодействие оказалось удачным, и многие системы управления базами данных и даже операционные системы включили, Java в свое ядро, например **Oracle**, **Linux**, **MacOS X**, **AIX**. Что еще не охвачено? Назовите, и через полгода услышите, что Java уже вовсю применяется и там. Из-за этой размытости самого понятия его описывают таким же размытым словом –**технология.**

Такое быстрое и широкое распространение технологии Java не в последнюю очередь связано с тем, что она использует новый, специально созданный язык программирования, который так и называется – язык Java. Этот язык создан на базе языков Smalltalk, Pascal,**C++** и др., вобрав их лучшие, по мнению создателей, черты и отбросив худшие. На этот счет есть разные мнения, но бесспорно, что язык получился удобным для изучения, написанные на нем программы легко читаются и отлаживаются: первую программу можно написать уже через час после начала изучения языка. Язык Java становится языком обучения объектно-ориентированному программированию, так же, как язык Pascal был языком обучения структурному программированию. Недаром на Java уже написано огромное количество программ, библиотек классов, а собственный апплет не написал только уж совсем ленивый.

Для полноты картины следует сказать, что создавать приложения для технологии Java можно не только на языке Java, уже появились и другие языки, есть даже компиляторы с языков Pascal и C++, но лучше все-таки использовать язык Java; на нем все аспекты технологии излагаются проще и удобнее. По скромному мнению автора, язык Java будет использоваться для описания различных приемов объектно-ориентированного программирования так же, как для реализации алгоритмов применялся вначале язык **Algol**, а затем язык **Pascal.**

Ясно, что всю технологию Java нельзя изложить в одной книге, полное описание ее возможностей составит целую библиотеку. Эта книга посвящена только языку Java. Прочитав ее, вы сможете создавать Java-приложения любой сложности, свободно разбираться в литературе и листингах программ, продолжать изучение аспектов технологии Java по специальной литературе. Язык Java тоже очень бурно развивается, некоторые его методы объявляются устаревшими (**deprecated**), появляются новые

конструкции, увеличивается встроенная библиотека классов, но есть устоявшееся.ядро языка, сохраняется его дух и стиль. Вот это-то устоявшееся и излагается в книге.

Структура книги

Книга состоит из четырех частей и приложения.

Первая часть содержит три главы, в которых рассматриваются базовые понятия языка. По прочтении ее вы сможете свободно разбираться в понятиях объектно-ориентированного программирования и их реализации на языке Java, создавать свои объектно-ориентированные программы, рассчитанные на консольный ввод/вывод.

В **главе 1** описываются типы исходных данных, операции с ними, выражения, массивы, операторы управления потоком информации, приводятся примеры записи часто встречающихся алгоритмов на Java. После знакомства с этой главой вы сможете писать программы на Java, реализующие любые вычислительные алгоритмы, встречающиеся в вашей практике.

В **главе 2 вводятся** основные понятия объектно-ориентированного программирования: объект и метод, абстракция, инкапсуляция, наследование, полиморфизм, контракты методов и их поручения друг другу. Эта глава призвана привить вам "объектный" взгляд на реализацию сложных проектов, после ее прочтения вы научитесь описывать проект как совокупность взаимодействующих объектов. Здесь же предлагается реализация всех этих понятий на языке Java. Тут вы, наконец, поймете, что же такое эти объекты и как, они взаимодействуют друг с другом,;

К главе 3 определяются пакеты классов и интерфейсы, ограничения доступа к классам и методам, на примерах подробно разбираются правила их использования. Объясняется структура встроенной библиотеки классов Java API.

Во **второй части** рассматриваются пакеты основных классов, составляющих неотъемлемую часть Java, разбираются приемы работы с ними и приводится примеры практического использования основных классов. Здесь вы увидите, как идеи объектно-ориентированного программирования реализуются на практике в сложных

производственных библиотеках классов. После изучения этой части вы сможете реализовывать наиболее часто встречающиеся ситуации объектно-ориентированного программирования с помощью стандартных классов.

Глава 4 прослеживает иерархию стандартных классов и интерфейсов Java, на этом примере показано, как в профессиональных системах программирования реализуются концепции абстракции, инкапсуляции и наследования.

В **главе 5** подробно излагаются приемы работы со строками символов, которые, как и все в Java, являются объектами, приводятся примеры синтаксического анализа текстов.

В **главе 6** показано, как в языке Java реализованы контейнеры, позволяющие работать с совокупностями объектов и создавать сложные структуры данных.

Глава 7 описывает различные классы-утилиты, полезные во многих ситуациях при работе с датами, случайными числами, словарями и другими необходимыми элементами программ.

В **третьей части** объясняется создание графического интерфейса пользователя (ГИП) с помощью стандартной библиотеки классов **AWT** (**Abstract Window Toolkit**) и даны многочисленные примеры построения интерфейса. Подробно разбирается принятый в Java метод обработки событий, основанный на идее делегирования. Здесь же появляются апплеты как программы Java, работающие в окне браузера. Подробно обсуждается система безопасности выполнения апплетов. После прочтения третьей части вы сможете создавать полноценные приложения под графические платформы MS Windows, X Window System и др., а также программировать браузеры.

Глава 8 описывает иерархию классов библиотеки AWT, которую необходимо четко себе представлять для создания удобного интерфейса. Здесь же рассматривается библиотека графических классов Swing, постепенно становящаяся стандартной наряду с AWT.

В **главе 9** демонстрируются приемы рисования с помощью графических примитивов, способы задания цвета и использование шрифтов, а также решается вопрос русификации приложений Java.

В **главе 10** обсуждается понятие графической составляющей, рассматриваются готовые компоненты AWT и их применение, а также создание собственных компонентов.

В **главе 11** показано, какие способы размещения компонентов в графическом контейнере имеются в AWT, и как их применять в разных ситуациях.

В **главе 12** вводятся способы реагирования компонентов на сигналы от клавиатуры и мыши, а именно, модель делегирования, принятая в Java.

В **главе 13** описывается создание системы меню – необходимой составляющей графического интерфейса.

В **главе 14**, наконец-то, появляются апплеты – Java-программы, предназначенные для выполнения в окне браузера, и обсуждаются их особенности.

В **главе 15** рассматривается работа с изображениями и звуком средствами AWT.

В **четвертой части** изучаются конструкции языка Java, не связанные общей темой. Некоторые из них необходимы для создания надежных программ, учитывающих все нештатные ситуации, другие позволяют реализовывать сложное взаимодействие объектов. Здесь же рассматривается передача потоков данных от одной программы Java к другой. Внимательное изучение четвертой части позволит вам дополнить свои разработки гибкими средствами управления выполнением приложения, создавать сложные клиент-серверные системы.

Глава 16 описывает средства обработки исключительных ситуаций, возникающих во время выполнения готовой программы, встроенные в Java.

Глава 17 рассказывает об уникальном свойстве языка Java – способности создавать подпроцессы (**threads**) и управлять их взаимодействием прямо из программы.

В **главе 18** обсуждается концепция потока данных и ее реализация в Java для организации ввода/вывода на внешние устройства.

Глава 19, последняя по счету, но не по важности, рассматривает сетевые средства языка Java, позволяющие скрыть все сложности протоколов Internet и максимально облегчить написание клиент-серверных приложений.

В **приложении** описываются дополнительные аспекты технологии Java: компоненты JavaBeans, сервлеты, драйверы соединения с базами данных JDBC, и прослеживаются пути дальнейшего развития технологии Java. Ознакомившись с этим приложением, вы сможете ориентироваться в информации о современном состоянии технологии Java и выбрать себе материал для дальнейшего изучения.

Выполнение Java-программы

Как вы знаете, программа, написанная на одном из языков высокого уровня, к которым относится и язык Java, так называемый **исходный модуль** ("исходник" или "сырец" на жаргоне, от английского "source"), не может быть сразу же выполнена. Ее сначала надо откомпилировать, т. е. перевести в последовательность машинных команд – **объектный модуль**. Но и он, как правило, не может быть сразу же выполнен: объектный модуль надо еще скомпоновать с библиотеками использованных в модуле функций и разрешить перекрестные ссылки между секциями объектного модуля, получив в результате**загрузочный модуль –** полностью готовую к выполнению программу.

Исходный модуль, написанный на Java, не может избежать этих процедур, но здесь проявляется главная особенность технологии Java – программа компилируется сразу в машинные команды, но не команды какого-то конкретного процессора, а в команды так называемой виртуальной машины Java (**JVM**, Java Virtual Machine). **Виртуальная машина Java –** это совокупность команд вместе с системой их выполнения. Для специалистов скажем, что виртуальная машина Java полностью стековая, так что не требуется сложная адресация ячеек памяти и большое количество регистров. Поэтому команды JVM короткие, большинство из них имеет длину 1 байт, отчего команды JVM называют **байт-кодами**(bytecodes), хотя имеются команды длиной 2 и 3 байта. Согласно статистическим исследованиям средняя длина команды составляет 1.8 байта. Полное описание команд и всей архитектуры JVM содержится в **спецификации виртуальной машины Java** (VMS, Virtual Machine Specification). Если вы хотите в точности узнать, как работает виртуальная машина Java, ознакомьтесь с этой спецификацией.

Другая особенность Java – все стандартные функции, вызываемые в программе, подключаются к ней только на этапе выполнения, а не включаются в байт-коды. Как говорят

специалисты, происходит **динамическая компоновка** (dynamic binding). Это тоже сильно уменьшает объем откомпилированной программы.

Итак, на первом этапе программа, написанная на языке Java, переводится компилятором в байт-коды. Эта компиляция не зависит от типа какого-либо конкретного процессора и архитектуры некоего конкретного компьютера. Она может быть выполнена один раз сразу же после написания программы. Байт-коды записываются в одном или нескольких файлах, могут храниться во внешней памяти или передаваться по сети. Это особенно удобно благодаря небольшому размеру файлов с байт-кодами. Затем полученные в результате компиляции байт-коды можно выполнять на любом компьютере, имеющем систему, реализующую JVM. При этом не важен ни тип процессора, ни архитектура компьютера. Так реализуется принцип Java: "Write once, run anywhere" – "Написано однажды, выполняется где угодно".

Интерпретация байт-кодов и динамическая компоновка значительно замедляют выполнение программ. Это не имеет значения в тех ситуациях, когда байт-коды передаются по сети, сеть все равно медленнее любой интерпретации, но в других ситуациях требуется мощный и быстрый компьютер. Поэтому постоянно идет усовершенствование интерпретаторов в сторону увеличения скорости интерпретации. Разработаны **JIT-компиляторы** (Just-In-Time), запоминающие уже интерпретированные участки кода в машинных командах процессора и просто выполняющие эти участки при повторном обращении, например, в циклах. Это значительно увеличивает скорость повторяющихся вычислений. Фирма SUN разработала целую технологию Hot-Spot и включает ее в свою виртуальную машину Java. Но, конечно, наибольшую скорость может дать только специализированный процессор.

Фирма SUN Microsystems выпустила микропроцессоры PicoJava, работающие на системе команд JVM, и собирается выпускать целую линейку все более мощных Java-процессоров. Есть уже и Java-процессоры других фирм. Эти процессоры непосредственно выполняют байт-коды. Но при выполнении программ Java на других процессорах требуется еще интерпретация команд JVM в команды конкретного процессора, а значит, нужна программа-интерпретатор, причем для каждого типа процессоров, и для каждой архитектуры компьютера следует написать свой интерпретатор.

Эта задача уже решена практически для всех компьютерных платформ. На них реализованы виртуальные машины Java, а для наиболее распространенных платформ имеется несколько реализаций JVM разных фирм. Все больше операционных систем и систем управления базами данных включают реализацию JVM в свое ядро. Создана и специальная операционная система JavaOS, применяемая в электронных устройствах. В большинство браузеров встроена виртуальная машина Java для выполнения апплетов.

Внимательный читатель уже заметил, что кроме реализации JVM для выполнения байт-кодов на компьютере еще нужно иметь набор функций, вызываемых из байт-кодов и динамически компонующихся с байт-кодами. Этот набор оформляется в виде библиотеки классов Java, состоящей из одного или нескольких **пакетов**. Каждая функция может быть записана байт-кодами, но, поскольку она будет храниться на конкретном компьютере, ее можно записать прямо в системе команд этого компьютера, избегнув тем самым интерпретации байт-кодов. Такие функции называют **родными" методами** (native methods). Применение "родных" методов ускоряет выполнение программы.

Фирма **SUN Microsystems** – создатель технологии Java – бесплатно распространяет набор необходимых программных инструментов для полного цикла работы с этим языком программирования: компиляции, интерпретации, отладки, включающий и богатую библиотеку классов, под названием **JDK** (Java Development Kit). Есть наборы инструментальных программ и других фирм. Например, большой популярностью пользуется JDK фирмы IBM.

Что такое JDK. Что такое JRE.

Набор программ и классов JDK содержит:

- компилятор **javac** из исходного текста в байт-коды;
- интерпретатор **java**, содержащий реализацию JVM;
- облегченный интерпретатор **jre** (в последних версиях отсутствует);
- программу просмотра апплетов **appietviewer**, заменяющую браузер;
- отладчик **jdt>**;
- дизассемблер **javap**;
- программу архивации и сжатия **jar**;
- программу сбора документации **javadoc**;
- программу **javah** генерации заголовочных файлов языка С;
- программу **javakey** добавления электронной подписи;
- программу **native2ascii**, преобразующую бинарные файлы в текстовые;
- программы **rmic** и **rmiregistry** для работы с удаленными объектами;
- программу **seriaiver**, определяющую номер версии класса;
- библиотеки и заголовочные файлы "родных" методов;
- библиотеку классов Java **API** (Application Programming Interface).

В прежние версии JDK включались и отладочные варианты исполнимых программ:**javac_g**, **java_g** и т. д.

Компания SUN Microsystems постоянно развивает и обновляет JDK, каждый год появляются новые версии.

В 1996 г. была выпущена первая версия JDK 1.0, которая модифицировалась до версии с номером 1.0.2. В этой версии библиотека классов Java API содержала 8 пакетов. Весь набор JDK 1.0.2 поставлялся в упакованном виде в одном файле размером около 5 Мбайт, а после распаковки занимал около 8 Мбайт на диске.

В 1997 г. появилась версия JDK 1.1, последняя ее модификация, 1.1.8, выпущена в 1998 г. В этой версии было 23 пакета классов, занимала она 8.5 Мбайт в упакованном виде и около 30 Мбайт на диске.

В первых версиях JDK все пакеты библиотеки Java API были упакованы в один архивный файл classes.zip и вызывались непосредственно из этого архива, его не нужно распаковывать.

Затем набор инструментальных средств JDK был сильно переработан.

Версия JDK 1.2 вышла в декабре 1998 г. и содержала уже 57 пакетов классов. В архивном виде это файл размером почти 20 Мбайт и еще отдельный файл размером более 17 Мбайт с упакованной документацией. Полная версия располагается на 130 Мбайтах дискового пространства, из них около 80 Мбайт занимает документация.

Начиная с этой версии, все продукты технологии Java собственного производства компания SUN стала называть **Java 2 Platform, Standard Edition**, сокращенно J2SE, а JDK переименовала в **Java 2 SDK, Standard Edition** (Software Development Kit), сокращенно J2SDK, поскольку выпускается еще **Java 2 SDK Enterprise Edition** и **Java 2 SDK Micro Edition**. Впрочем, сама компания SUN часто пользуется и старым названием, а в литературе утвердилось название Java 2.

Кроме 57 пакетов классов, обязательных на любой платформе и получивших название**Core API**, в Java 2 SDK vl.2 входят еще дополнительные пакеты классов, называемые Standard Extension API. В версии Java 2 SDK SE, vl.3, вышедшей в 2000 г., уже 76 пакетов классов, составляющих Core API. В упакованном виде это файл размером около 30 Мбайт, и еще файл с упакованной документацией размером 23 Мбайта. Все это распаковывается в 210 Мбайт дискового пространства. Эта версия требует процессор Pentium 166 и выше и не менее 32 Мбайт оперативной памяти.

В настоящее время версия JDK 1.0.2 уже не используется. Версия JDK 1.1.5 с графической библиотекой AWT встроена в популярные браузеры Internet Explorer 5.0 и Netscape

Communicator 4.7, поэтому она применяется для создания апплетов. Технология Java 2 широко используется на серверах и в клиент-серверных системах.

Кроме JDK, компания SUN отдельно распространяет еще и набор **JRE** (Java Runtime Environment).

Что такое JRE

Набор программ и пакетов классов JRE содержит все необходимое для выполнения байт-кодов, в том числе интерпретатор java (в прежних версиях облегченный интерпретатор **jre**) и библиотеку классов. Это часть JDK, не содержащая компиляторы, отладчики и другие средства разработки. Именно JRE или его аналог других фирм содержится в браузерах, умеющих выполнять программы на Java, операционных системах и системах управления базами данных.

Хотя JRE входит в состав JDK, фирма SUN распространяет этот набор и отдельным файлом.

Версия JRE 1.3.0 – это архивный файл размером около 8 Мбайт, разворачивающийся в 20 Мбайт на диске.

Как установить JDK

Набор JDK упаковывается в самораспаковывающийся архив. Раздобыв каким-либо образом этот архив: "выкачав" из Internet, с http://java.sun.com/produets/jdk/ или какого-то другого адреса, получив компакт-диск, вам остается только запустить файл с архивом на выполнение. Откроется окно установки, в котором среди всего прочего вам будет предложено выбрать каталог (**directory**) установки, например, C:\jdkl.3.

Если вы согласитесь с предлагаемым каталогом, то вам больше не о чем беспокоиться. Если вы указали собственный каталог, то проверьте после установки значение переменной PATH, набрав в командной строке окна **MS-DOS Prompt (или** окна **Command Prompt** в Windows NT/2000, а тот, кто работает в UNIX, сами знают, что делать) команду **set**. Переменная PATH должна содержать полный путь к подкаталогу bin этого каталога. Если нет, то добавьте этот путь, например, C:\jdkl.3\bin. Надо определить и специальную переменную CLASSPATH, содержащую пути к архивным файлам и каталогам с библиотеками классов. Системные библиотеки Java 2 подключаются автоматически, без переменной CLASSPATH.

Еще одно предупреждение: не следует распаковывать zip- и jar-архивы.

После установки вы получите каталог с названием, например, jdkl.3, а в нем подкаталоги:

- **bin**, содержащий исполнимые файлы;
- **demo**, содержащий примеры программ;
- **docs**, содержащий документацию, если вы ее установили;
- **include**, содержащий заголовочные файлы "родных" методов;
- **jre**, содержащий набор JRE;
- **old-include**, для совместимости со старыми версиями;
- **lib**, содержащий библиотеки классов и файлы свойств;

- **src**, с исходными текстами программ JDK. В новых версиях вместо каталога имеется упакованный файл src.jar.

Да-да! Набор JDK содержит исходные тексты большинства своих программ, написанные на Java. Это очень удобно. Вы всегда можете в точности узнать, как работает тот или иной метод обработки информации из JDK, посмотрев исходный код данного метода. Это очень полезно и для изучения Java на "живых" работающих примерах.

Как использовать JDK

Несмотря на то, что набор JDK предназначен для создания программ, работающих в графических средах, таких как MS Windows или X Window System, он ориентирован на выполнение из командной строки окна **MS-DOS Prompt** в Windows 95/98/ME или окна**Command Prompt** в Windows NT/2000. В системах UNIX можно работать и в текстовом режиме и в окне **Xterm**.

Написать программу на Java можно в любом текстовом редакторе, например, Notepad, WordPad в MS Windows, редакторах vi, emacs в UNIX. Надо только сохранить файл в текстовом формате и дать ему расширение Java.

Пусть для примера, именем файла будет MyProgram.java, а сам файл сохранен в текущем каталоге.

После создания этого файла из командной строки вызывается компилятор javac и ему передается исходный файл как параметр:

```
javac MyProgram.java
```

Компилятор создает в том же каталоге по одному файлу на каждый класс, описанный в программе, называя каждый файл именем класса с расширением **.class**. Допустим, в нашем примеру имеется только один класс, названный MyFrogram, тогда получаем файл с именем MyProgram.class, содержащий байт-коды.

Компилятор молчалив – если компиляция прошла успешно, он ничего не сообщит, на экране появится только приглашение операционной системы. Если же компилятор заметит ошибки, то он выведет, на экран сообщения о них. Большое достоинство компилятора JDK в том что он "отлавливает" много ошибок и выдает подробные и понятные сообщения о них.

Далее из командной строки вызывается интерпретатор байт-кодов jaya, которому передается файл с байт-кодами, причем его имя записывается без расширения (смысл этого вы узнаете позднее):

```
Java MyProgram
```

На экране появляется вывод результатов работы программы или сообщения об ошибках времени выполнения.

Если работа из командной строки, столь милая сердцу "юниксоидов", кажется вам несколько устаревшей, используйте для разработки интегрированную среду.

Интегрированные среды Java

Сразу же после создания Java, уже в 1996 г., появились интегрированные среды разработки программ для Java, и их число все время возрастает. Некоторые из них являются

просто интегрированными оболочками над JDK, вызывающими из одного окна текстовый редактор, компилятор и интерпретатор.

Эти интегрированные среды требуют предварительной установки JDK. Другие содержат JDK в себе или имеют собственный компилятор, например, **Java Workshop** фирмы SUN Microsystems, **JBuilder** фирмы Inprise, **Visual Age for Java** фирмы IBM и множество других программных продуктов. Их можно устанавливать, не имея под руками JDK. Надо заметить, что перечисленные продукты написаны полностью на Java.

Большинство интегрированных сред являются средствами визуального программирования и позволяют быстро создавать пользовательский интерфейс, т е. относятся к классу средств **RAD** (Rapid Application Development).

Выбор какого-либо средства разработки диктуется, во-первых, возможностями вашего компьютера, ведь визуальные среды требуют больших ресурсов, во-вторых, личным вкусом, в-третьих, уже после некоторой практики, достоинствами компилятора, встроенного в программный продукт.

В России по традиции, идущей от TurboPascal к Delphi, большой популярностью пользуется JBuilder, позволяющий подключать сразу несколько JDK разных версий и использовать их компиляторы кроме собственного. Многие профессионалы предпочитают**Visual Age for Java**, в котором можно графически установить связи между объектами.

К технологии Java подключились и разработчики CASE-средств. Например, популярный во всем мире продукт **Rational Rose** может сгенерировать код на Java.

Особая позиция Microsoft

Вы уже, наверное, почувствовали смутное беспокойство, не встречая название этой фирмы. Дело в том, что, имея свою операционную систему, огромное число приложений к ней и богатейшую библиотеку классов, компания Microsoft не имела нужды в Java. Но и пройти мимо технологии, распространившейся всюду, компания Microsoft не могла и создала свой компилятор Java, а также визуальное средство разработки, включив его в Visual Studio.

Этот компилятор включает в байт-коды вызовы объектов ActiveX. Следовательно, выполнять эти байт-коды можно только на компьютерах, имеющих доступ к ActiveX. Эта "нечистая" Java резко ограничивает круг применения байт-кодов, созданных компилятором фирмы Microsoft. В результате судебных разбирательств с SUN Microsystems компания Microsoft назвала свой продукт Visual J++. Виртуальная машина Java фирмы Microsoft умеет выполнять байт-коды, созданные "чистым" компилятором, но не всякий интерпретатор выполнит байт-коды, написанные с помощью Visual J++.

Чтобы прекратить появление несовместимых версий Java, фирма SUN разработала концепцию "чистой" Java, назвав ее **Pure Java**, и систему проверочных тестов на "чистоту" байт-кодов. Появились байт-коды, успешно прошедшие тесты, и средства разработки, выдающие "чистый" код и помеченные как "**100% Pure Java**".

Кроме того, фирма SUN распространяет пакет программ Java Plug-in, который можно подключить к браузеру, заменив тем самым встроенный в браузер JRE на "родной".

Java в Internet

Разработанная для применения в сетях, Java просто не могла не найти отражения на сайтах Internet. Действительно, масса сайтов полностью посвящена или содержит информацию о технологии Java. Одна только фирма SUN содержит несколько сайтов с информацией о Java:

- http://www.sun.com/– здесь все ссылки, отсюда можно скопировать JDK;
- http://java.sun.com/– основной сайт Java, отсюда тоже можно скопировать JDK;
- http://developer.java.sun.com/– масса полезных вещей для разработчика;
- http://industry.java.sun.com/– новости технологии Java;
- http://www.javasoft.com/– сайт фирмы JavaSoft, подразделения SUN;
- http://www.gamelan.com/.

На сайте фирмы IBM есть большой раздел http://www.ibm.com/developer/Java/, где можно найти очень много полезного для программиста.

Компания Microsoft содержит информацию о Java на своем сайте:http://www.microsoft.com/java/.

Большой вклад в развитие технологии Java вносит корпорация Oracle:http://www.oracle.coin/.

Существует множество специализированных сайтов:

- http://java.iba.com.by/– Java team IBA (Белоруссия);
- http://www.artima.com/;
- http://www.freewarejava.com/;
- http://www.jars.com/– Java Review Service;
- http://www.javable.com – русскоязычный сайт;
- http://www.javaboutique.com/;
- http://www.javalobby.com/;
- http://www.javalogy.com/;
- http://www.javaranch.com/;
- http://www.javareport.com/– независимый источник информации для разработчиков;
- http://www.javaworld.com – электронный журнал;
- http://www.jfind.com/ – сборник программ и статей;
- http://www.jguru.com/ – советы специалистов;
- http://www.novocode.com/;
- http://www.sigs.com/jro/ – Java Report Online;

- http://www.sys-con.com/java/;
- http://theserverside.com/ – вопросы создания серверных Java-приложений;
- http://servlets.chat.ru/;
- http://javapower.da.ru/ – собрание FAQ на русском языке;
- http://www.purejava.ru/;
- http://java7.da.ru/;
- http://codeguru.earthweb.com/java/ – большой сборник апплетов и других программ;
- http://securingjava.com/ – обсуждаются вопросы безопасности;
- http://www.servlets.com/ – вопросы по написанию апплетов;
- http://www.servletsource.com/;
- http://coolservlets.com/;
- http://www.servletforum.com/;
- http://www.javacats.com/.

Персональные сайты:

- http://www.bruceeckel.com/ – сайт Bruce Eckel;
- http://www.davidreilly.com/java/;
- http://www.comita.spb.ru/users/sergeya/java/ – хозяин, Сергей Астахов, собрал здесь буквально все, касающееся русификации Java.

К сожалению, адреса сайтов часто меняются. Возможно, вы и не найдете некоторые из перечисленных сайтов, зато возникнет много других.

Литература по Java

Перечислим здесь только основные, официальные и почти официальные издания, более полное описание чрезвычайно многочисленной литературы дано в списке литературы в конце книги.

Полное и строгое описание языка изложено в книге **The Java Language Specification**, Second Edition. **James Gosling**, **Bill Joy**, **Guy Steele**, **Gilad Bracha**. Эта книга в электронном виде находится по адресу http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html и занимает в упакованном виде около 400 Кбайт.

Столь же полное и строгое описание виртуальной машины Java изложено в книге **The Java Virtual Machine Specification**, Second Edition. **Tim Lindholm**, **Frank Yellin**. В электронном виде она находится по адресу http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html.

Здесь же необходимо отметить книгу "отца" технологии Java Джеймса Гослинга, написанную вместе с Кеном Арнольдом. Имеется русский перевод **Гослинг Дж.**, **Арнольд К.**,**Язык программирования Java**: Пер. с англ. – СПб.: Питер, 1997. – 304 с.: ил.

Компания SUN Microsystems содержит на своем сайте постоянно обновляемый электронный учебник **Java Tutorial**, размером уже более 14 Мбайт:http://java.sun.com/docs/books/tutorial/. Время от времени появляется его печатное издание**The Java Tutorial**, Second Edition: Object-Oriented Programming for the Internet. **Mary Campione**, **Kathy Walrath**.

Полное описание Java API содержится в документации, но есть печатное издание **The Java Application Programming Interface**. **James Gosling**, **Frank Yellin and the Java Team**, Volume 1: Core Packages; Volume 2: Window Toolkit and Applets.

Встроенные типы данных, операции над ними

- **Первая программа на Java**
  Приступая к изучению нового языка, полезно поинтересоваться, какие исходные данные могут обрабатываться средствами этого языка, в каком виде их можно задавать, и какие стандартные средства обработки этих данных заложены в язык.
- **Комментарии**
  В текст программы можно вставить комментарии, которые компилятор не будет учитывать. Они очень полезны для пояснений по ходу программы. В период отладки можно выключать из действий один или несколько операторов, пометив их символами комментария, как говорят программисты, "закомментарив" их.
- **Константы**
  В языке Java можно записывать константы разных типов в разных видах. Перечислим их. | Целые | Целые константы можно записывать в трех системах счисления: | в десятичной форме: +5, -7, 12345678; | в восьмеричной форме, начиная с нуля: 027, -0326, 0777; в записи таких констант недопустимы цифры 8 и 9;
- **Имена**
  Имена (names) переменных, классов, методов и других объектов могут быть простыми (общее название – идентификаторы (idenifiers)) и составными (qualified names). Идентификаторы в Java составляются из так называемых букв Java (Java letters) и арабских цифр 0-9, причем первым символом идентификатора не может быть цифра.
- **Примитивные типы данных и операции**
  Все типы исходных данных, встроенные в язык Java, делятся на две группы: примитивные типы (primitive types) и ссылочные типы (reference types). | Ссылочные типы делятся на массивы (arrays), классы (classes) и интерфейсы (interfaces). | Примитивных типов всего восемь.
- **Логический тип. Логические операции.**
  Значения логического типа boolean возникают в результате различных сравнений, вроде 2 > 3, и используются, главным образом, в условных операторах и операторах циклов. Логических значении всего два: true (истина) и false (ложь). Это служебные слова Java.
- **Целые типы**
  Спецификация языка Java, JLS, определяет разрядность (количество байтов, выделяемых для хранения значений типа в оперативной памяти) и диапазон значений каждого типа. Для целых типов они приведены в табл. 1.2. | Таблица 1.2. Целые типы. | Тип | Разрядность (байт) | Диапазон | byte | 1
- **Операции над целыми типами**

Все операции, которые производятся над целыми числами, можно разделить на следующие группы. | Арифметические операции | К арифметическим операциям относятся: | сложение + (плюс); | вычитание - (дефис); | умножение * (звездочка); | деление / (наклонная черта – слэш);

- **Вещественные типы**
  Вещественных типов в Java два: float и double. Они характеризуются разрядностью, диапазоном значений и точностью представления, отвечающим стандарту IEEE 754-1985 с некоторыми изменениями. К обычным вещественным числам добавляются еще три значения:
- **Операции присваивания. Условная операция.**
  Простоя операция присваивания (simple assignment operator) записывается знаком равенства =, слева от которого стоит переменная, а справа выражение, совместимое с типом переменной: | x = 3.5, y = 2 * (x -0.567) / (x + 2), b = x < y, bb = x >= y && b.
- **Выражения**
  Из констант и переменных, операций над ними, вызовов методов и скобок составляются выражения (expressions). Разумеется, все элементы выражения должны быть совместимы, нельзя написать, например, 2 + true. При вычислении выражения выполняются четыре правила: | 1.
- **Приоритет операций**
  Операции перечислены в порядке убывания приоритета. Операции на одной строке имеют одинаковый приоритет. | Постфиксные операции ++ и --. | Префиксные операции ++ и --, дополнение ~ и отрицание !. | Приведение типа (тип). | Умножение *, деление / и взятие остатка %. | Сложение + и вычитание -.
- **Операторы. Блок. Операторы присваивания.**
  Как вы знаете, любой алгоритм, предназначенный для выполнения на компьютере, можно разработать, используя только линейные вычисления, разветвления и циклы. | Записать его можно в разных формах: в виде блок-схемы, на псевдокоде, на обычном языке, как мы записываем кулинарные рецепты, или как-нибудь еще "алгоритмы".,-. | Всякий язык программирования должен иметь средства записи алгоритмов.
- **Условный оператор**
  Условный оператор (if-then-else statement) в языке Java записывается так: | if (логВыр) оператор1 else оператор2 | …и действует следующим образом. Сначала вычисляется логическое выражение логвыр. Если результат true, то действует оператор1 и на этом действие условного оператора завершается, оператор2 не действует, далее будет выполняться следующий за if оператор.
- **Операторы цикла**
  Основной оператор цикла – оператор while – выглядит так: | while (логВыр) оператор | Вначале вычисляется логическое выражение логВыр; если его значение true, то выполняется оператор, образующий цикл. Затеем снова вычисляется лог-выр и действует оператор, и так до тех пор, пока не получится значение false. Если логВыр изначально равняется false, то оператор не будет выполнен ни разу.
- **Оператор continue и метки. Оператор break.**
  Оператор continue используется только в операторах цикла. Он имеет две формы. Первая форма состоит только из слова continue и осуществляет немедленный переход к следующей итерации цикла. | В очередном фрагменте кода оператор continue позволяет обойти деление на нуль: | for (int i = 0; i < N;
- **Оператор варианта**
  Оператор варианта switch организует разветвление по нескольким направлениям. Каждая ветвь отмечается константой или константным выражением какого-либо целого типа (кроме long) и выбирается, если значение определенного выражения совпадет с этой константой. Вся конструкция выглядит так.
- **Массивы**
  Как всегда в программировании массив – это совокупность переменных одного типа, хранящихся в смежных ячейках оперативной памяти. | Массивы в языке Java относятся к ссылочным типам и описываются своеобразно, но характерно для ссылочных типов. Описание производится в три этапа.

- **Многомерные массивы**
  Элементами массивов в Java могут быть снова массивы. Можно объявить: | char[] [] c; | …что эквивалентно: | char c[] c[]; | …или: | char c[][]; | Затем определяем внешний массив: | c = new char[3][]; | Становится ясно, что c – массив, состоящий из трех элементов-массивов.

Первая программа на Java

Приступая к изучению нового языка, полезно поинтересоваться, какие исходные данные могут обрабатываться средствами этого языка, в каком виде их можно задавать, и какие стандартные средства обработки этих данных заложены в язык. Это довольно скучное занятие, поскольку в каждом развитом языке программирования множество типов данных и еще больше правил их использования. Однако несоблюдение этих правил приводит к появлению скрытых ошибок, обнаружить которые иногда бывает очень трудно. Ну что же, в каждом ремесле приходится сначала "играть гаммы", и мы не можем от этого уйти.

Все правила языка Java исчерпывающе изложены в его спецификации, сокращенно называемой JLS. Иногда, чтобы понять, как выполняется та или иная конструкция языка Java, приходится обращаться к спецификации, но, к счастью, это бывает редко, правила языка Java достаточно просты и естественны.

В этой главе перечислены примитивные типы данных, операции над ними, операторы управления, и показаны "подводные камни", которых следует избегать при их использовании. Но начнем, по традиции, с простейшей программы.

По давней традиции, восходящей к языку C, учебники по языкам программирования начинаются с программы "Hello, World!". Не будем нарушать эту традицию. В листинге 1.1 эта программа в самом простом виде, записанная на языке Java.

Листинг 1.1. Первая программа на языке Java.

```
class HelloWorld{
public static void main(String[] args){
System.out.println("Hello, XXI Century World!");
}
}
```

Вот и все, всего пять строчек! Но даже на этом простом примере можно заметить целый ряд существенных особенностей языка Java.

- Всякая программа представляет собой один или несколько классов, в этом простейшем примере только один **класс** (class).
- Начало класса отмечается служебным словом class, за которым следует имя класса, выбираемое произвольно, в данном случае Helloworld. Все, что содержится в классе, записывается в фигурных скобках и составляет **тело класса** (class body).

- Все действия производятся с помощью методов обработки информации, коротко говорят просто **метод** (method). Это название употребляется в языке Java вместо названия "функция", применяемого в других языках.
- Методы различаются по именам. Один из методов обязательно должен называться main, с него начинается выполнение программы. В нашей простейшей программе только один метод, а значит, имя ему main.
- Как и положено функции, метод всегда выдает в результате (чаще говорят,**возвращает** (returns)) только одно значение, тип которого обязательно указывается перед именем метода. Метод может и не возвращать никакого значения, играя роль процедуры, как в нашем случае. Тогда вместо типа возвращаемого значения записывается слово void, как это и сделано в примере.
- После имени метода в скобках, через запятую, перечисляются **аргументы** (arguments) – или **параметры** метода. Для каждого аргумента указывается его тип и, через пробел, имя. В примере только один аргумент, его тип – массив, состоящий из строк символов. Строка символов – это встроенный в Java API тип string, а квадратные скобки – признак массива. Имя массива может быть произвольным, в примере выбрано имя args.
- Перед типом возвращаемого методом значения могут быть записаны **модификаторы**(modifiers). В примере их два: слово public означает, что этот метод доступен отовсюду; слово static обеспечивает возможность вызова метода main () в самом начале выполнения программы. Модификаторы вообще необязательны, но для метода main () они необходимы.

  Замечание

  В тексте этой книги после имени метода ставятся скобки, чтобы подчеркнуть, что это имя именно метода, а не простой переменной.
- Все, что содержит метод, **тело метода** (method body), записывается в фигурных скобках.

Единственное действие, которое выполняет метод **main ()** в примере, заключается в вызове другого метода со сложным именем System.out.println и передаче ему на обработку одного аргумента, текстовой константы "Hello, 21th century world!". Текстовые константы записываются в кавычках, которые являются только ограничителями и не входят в состав текста.

Составное имя System.out.println означает, что в классе System, входящем в Java API, определяется переменная с именем out, содержащая экземпляры одного из классов Java API, класса PrintStream, в котором есть метод **println()**. Все это станет ясно позднее, а пока просто будем писать это длинное имя.

Действие метода println () заключается в выводе своего аргумента в выходной поток, связанный обычно с выводом на экран текстового терминала, в окно **MS-DOS Prompt** или**Command Prompt** или **Xterm**, в зависимости от вашей системы. После вывода курсор переходит на начало следующей строки экрана, на что указывает окончание ln, слово println – сокращение слов print line. В составе Java API есть и метод print (), оставляющий курсор в конце выведенной строки. Разумеется, это прямое влияние языка Pascal.

Сделаем сразу важное замечание. Язык Java различает строчные и прописные буквы, имена main, Main, MAIN различны с "точки зрения" компилятора Java. В примере важно писать String, System с заглавной буквы, а main с маленькой. Но внутри текстовой константы неважно, писать Century или century, компилятор вообще не "смотрит" на нее, разница будет видна только на экране.

Замечание

Язык Java различает прописные и строчные буквы.

Свои имена можно записывать как угодно, можно было бы дать классу имя helloworld или helloworld, но между Java-программистами заключено соглашение, называемое "Code Conventions for the Java Programming Language", хранящееся по адресуhttp://java.sun.com/docs/codeconv/index.html. Вот несколько пунктов этого соглашения:

- имена классов начинаются с прописной буквы; если имя содержит несколько слов, то каждое слово начинается с прописной буквы;
- имена методов и переменных начинаются со строчной буквы; если имя содержит несколько слов, то каждое следующее слово начинается со строчной буквы;
- имена констант записываются полностью прописными буквами; если имя состоит из нескольких слов, то между ними ставится знак подчеркивания.

Конечно, эти правила необязательны, хотя они и входят в JLS, п. 6.8, но сильно облегчают понимание кода и придают программе характерный для Java стиль.

Стиль определяют не только имена, но и размещение текста программы по строкам, например, расположение фигурных скобок: оставлять ли открывающую фигурную скобку в конце строки с заголовком класса или метода или переносить на следующую строку? Почему-то этот пустяшный вопрос вызывает ожесточенные споры, некоторые средства разработки, например JBuilder, даже предлагают выбрать определенный стиль расстановки фигурных скобок. Многие фирмы устанавливают свой, внутрифирменный стиль. В книге мы постараемся следовать стилю "Code Conventions" и в том, что касается разбиения текста программы на строки (компилятор же рассматривает всю программу как одну длинную строку, для него программа – это просто последовательность символов), и в том, что касается отступов (indent) в тексте.

Итак, программа написана в каком-либо текстовом редакторе, например, Notepad. Теперь ее надо сохранить в файле, имя которого совпадает с именем класса, содержащего метод main (), и дать имени файла расширение Java. Это правило очень желательно выполнять. При этом система исполнения Java будет быстро находить метод main() для начала работы, просто отыскивая класс, совпадающий с именем файла.

Совет

Называйте файл с программой именем класса, содержащего метод main (), соблюдая регистр букв.

В нашем примере, сохраним программу в файле с именем HelloWorld.java в текущем каталоге. Затем вызовем компилятор, передавая ему имя файла в качестве аргумента:

```
javac HelloWorld.java
```

Компилятор создаст файл с байт-кодами, даст ему имя Helloworld.class и запишет этот файл в текущий каталог.

Осталось вызвать интерпретатор, передав ему в качестве аргумента имя класса (а не файла):

```
Java HelloWorld
```

На экране появится:

```
Hello, 21st Century World!
```

Замечание

Не указывайте расширение .class при вызове интерпретатора.

На рис. 1.1 показано, как все это выглядит в окне **Command Prompt** операционной системы MS Windows 2000.

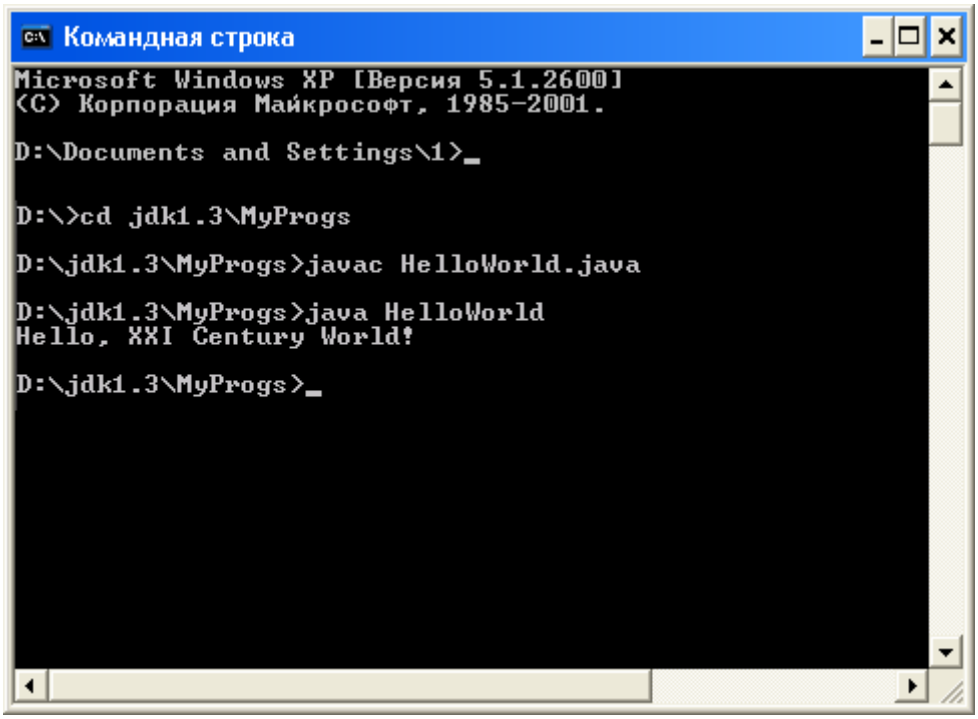

Рис. 1.1. Окно Command Prompt

При работе в интегрированной среде все эти действия вызываются выбором соответствующих пунктов меню или "горячими" клавишами – единых правил здесь нет.

Комментарии

В текст программы можно вставить комментарии, которые компилятор не будет учитывать. Они очень полезны для пояснений по ходу программы. В период отладки можно выключать из действий один или несколько операторов, пометив их символами комментария, как говорят программисты, "закомментарив" их. Комментарии вводятся таким образом:

- за двумя наклонными чертами подряд //, без пробела между ними, начинается комментарий, продолжающийся до конца строки;
- за наклонной чертой и звездочкой /* начинается комментарий, который может занимать несколько строк, до звездочки и наклонной черты */ (без пробелов между этими знаками).

Комментарии очень удобны для чтения и понимания кода, они превращают программу в документ, описывающий ее действия. Программу с хорошими комментариями называют**самодокументированной**. Поэтому в Java введены комментарии третьего типа, а в состав JDK – программа javadoc, извлекающая эти комментарии в отдельные файлы формата HTML и создающая гиперссылки между ними: за наклонной чертой и двумя звездочками подряд, без пробелов, **/**** начинается комментарий, который может занимать несколько строк до звездочки (одной) и наклонной черты ***/** и обрабатываться программой javadoc. В такой комментарий можно вставить указания программе javadoc, которые начинаются с символа @.

Именно так создается документация к JDK.

Добавим комментарии к нашему примеру (листинг 1.2).

Листинг 1.2. Первая программа с комментариями.

```
/**
* Разъяснение содержания и особенностей программы…
* @author Имя Фамилия (автора)
* @version 1.0 (это версия программы)
*/
class HelloWorld{ // HelloWorld – это только имя
// Следующий метод начинает выполнение программы
public static void main(String[] args){ // args не используются
/* Следующий метод просто выводит свой аргумент
 на экран дисплея */
System.out.println("Hello, 21st Century World!");
// Следующий вызов закомментирован,
// метод не будет выполняться
// System.out.println("Farewell, 20th Century!");
}
}
```

Звездочки в начале строк не имеют никакого значения, они написаны просто для выделения комментария. Пример, конечно, перегружен пояснениями (это плохой стиль), здесь просто показаны разные формы комментариев.

Константы

В языке Java можно записывать константы разных типов в разных видах. Перечислим их.

Целые

Целые константы можно записывать в трех системах счисления:

- в десятичной форме: +5, -7, 12345678;
- в восьмеричной форме, начиная с нуля: 027, -0326, 0777; в записи таких констант недопустимы цифры 8 и 9;

Замечание

Число, начинающееся с нуля, записано в восьмеричной форме, а не в десятичной.

- в шестнадцатеричной форме, начиная с нуля и латинской буквы x или X: 0xff0a, 0xFC2D, 0x45a8, 0X77FF; здесь строчные и прописные буквы не различаются.

Целые константы хранятся в формате типа **int** (см. ниже).

В конце целой константы можно записать букву прописную L или строчную l, тогда константа будет сохраняться в длинном формате типа **long** (см. ниже): +25L, -0371, OxffL, OXDFDF1.

Совет

Не используйте при записи длинных целых констант строчную латинскую букву l, ее легко спутать с единицей.

Действительные

Действительные константы записываются только в десятичной системе счисления в двух формах:

- с фиксированной точкой: 37.25, -128.678967, +27.035;
- с плавающей точкой: 2.5e34, -0.345e-25, 37.2E+4; можно писать строчную или прописную латинскую букву E; пробелы и скобки недопустимы.

В конце действительной константы можно поставить букву F или f, тогда константа будет сохраняться в формате типа **float** (см. ниже): 3.5f, -45.67F, 4.7e-5f. Можно приписать и букву D (или d): 0.045D, -456.77889d, означающую тип **double**, но это излишне, поскольку действительные константы и так хранятся в формате типа double.

Символы

Для записи одиночных символов используются следующие формы.

- Печатные символы можно записать в апострофах: ' a ', ' N ', '? '.
- Управляющие символы записываются в апострофах с обратной наклонной чертой:
  - **' \n '** – символ перевода строки newline с кодом ASCII 10;
  - **' \r '** – символ возврата каретки CR с кодом 13;
  - **' \f '** – символ перевода страницы FF с кодом 12;
  - **' \b '** – символ возврата на шаг BS с кодом 8;
  - **' \t '** – символ горизонтальной табуляции HT с кодом 9;
  - **' \\ '** – обратная наклонная черта;
  - **' \" '** – кавычка;
  - **' \' '** – апостроф.
- Код любого символа с десятичной кодировкой от 0 до 255 можно задать, записав его не более чем тремя цифрами в восьмеричной системе счисления в апострофах после обратной наклонной черты: ' \123 ' – буква S, ' \346 ' – буква Ж в кодировке CP1251. Не рекомендуется использовать эту форму записи для печатных и управляющих символов, перечисленных в предыдущем пункте, поскольку компилятор сразу же переведет восьмеричную запись в указанную выше форму. Наибольший код ' \377 ' – десятичное число 255.
- Код любого символа в кодировке Unicode набирается в апострофах после обратной наклонной черты и латинской буквы **u** ровно четырьмя шестнадцатеричными цифрами: ' \u0053 ' – буква S, ' \u0416 ' – буква Ж.

Символы хранятся в формате типа **char** (см. ниже).

Примечание

Прописные русские буквы в кодировке Unicode занимают диапазон от ' \u0410 ' – заглавная буква А, до ' \u042F ' – заглавная Я, строчные буквы от ' \u0430 ' – а, до ' \044F ' – я.

В какой бы форме ни записывались символы, компилятор переводит их в Unicode, включая и исходный текст программы.

Замечание

Компилятор и исполняющая система Java работают только с кодировкой Unicode.

Строки

Строки символов заключаются в кавычки. Управляющие символы и коды записываются в строках точно так же, с обратной наклонной чертой, но, разумеется, без апострофов, и оказывают то же действие. Строки могут располагаться только на одной строке исходного кода, нельзя открывающую кавычку поставить на одной строке, а закрывающую – на следующей.

Вот некоторые примеры:

```
"Это строка\nc переносом"
"\"Спартак\" - Чемпион!"
```

Замечание

Строки символов нельзя начинать на одной строке исходного кода, а заканчивать на другой.

Для строковых констант определена операция сцеплений, обозначаемая плюсом.

```
" Сцепление " + "строк" дает в результате строку "Сцепление строк".
```

Чтобы записать длинную строку в виде одной строковой константы, надо после закрывающей кавычки на первой и следующих строках поставить плюс **+**; тогда компилятор соберет две (или более) строки в одну строковую константу, например:

```
"Одна строковая константа, записанная "+
"на двух строках исходного текста"
```

Тот, кто попытается выводить символы в кодировке Unicode, например, слово "Россия":

```
System.out.println("\u0429\u043e\u0441\u0441\u0438\u044f");
```

…должен знать, что Windows 95/98/ME вообще не работает с Unicode, а Windows NT/2000 использует для вывода в окно **Command Prompt** шрифт Terminal, в котором русские буквы, расположены в начальных кодах Unicode, почему-то в кодировке CP866, и разбросаны по другим сегментам Unicode.

Не все шрифты Unicode содержат начертания (**glyphs**) всех символов, поэтому будьте осторожны при выводе строк в кодировке Unicode.

Совет

Используйте Unicode напрямую только в крайних случаях.

Имена

Имена (names) переменных, классов, методов и других объектов могут быть простыми (общее название – **идентификаторы** (idenifiers)) и **составными** (qualified names). Идентификаторы в Java составляются из так называемых **букв Java** (Java letters) и арабских цифр 0-9, причем первым символом идентификатора не может быть цифра. (Действительно, как понять запись 2e3: как число 2000.0 или как имя переменной?) В число букв Java обязательно входят прописные и строчные латинские буквы, знак доллара **$** и знак подчеркивания _, а так же символы национальных алфавитов.

Замечание

Не указывайте в именах знак доллара. Компилятор Java использует его для записи имен вложенных классов.

Вот примеры правильных идентификаторов:

```
al my_var var3_5 _var veryLongVarName
aName theName a2Vh36kBnMt456dX
```

В именах лучше не использовать строчную букву **l**, которую легко спутать с единицей, и букву **o**, которую легко принять за нуль.

Не забывайте о рекомендациях "Code Conventions".

В классе Character, входящем в состав Java API, есть два метода, проверяющие, пригоден ли данный символ для использования в идентификаторе: **isJavaidentifierStart()**, проверяющий, является ли символ буквой Java, и **isJavaldentifierPart()**, выясняющий, является ли символ – буквой или цифрой.

Служебные слова Java, такие как **class**, **void**, **static**, зарезервированы, их нельзя использовать в качестве идентификаторов своих объектов.

Составное имя (qualified name) – это несколько идентификаторов, разделенных точками, без пробелов, например, уже встречавшееся нам имя System.out.println.

Примитивные типы данных и операции

Все типы исходных данных, встроенные в язык Java, делятся на две группы:**примитивные типы** (primitive types) и **ссылочные типы** (reference types).

Ссылочные типы делятся на **массивы** (arrays), **классы** (classes) и **интерфейсы**(interfaces).

Примитивных типов всего восемь. Их можно разделить на **логический** (иногда говорят**булев)** тип boolean и **числовые** (numeric).

К числовым типам относятся **целые** (integral *[Название "integral" не является устоявшимся термином. Так названа категория целых типов данных в книге The Java Language Specification, Second Edition. James Gosling, Bill Joy, Guy Steele, Gilad Bracha (см. введение). – Ред.]*) и **вещественные** (floating-point) типы.

Целых типов пять: **byte**, **short**, **int**, **long**, **char**.

Символы можно использовать везде, где используется тип int, поэтому JLS причисляет их к целым типам. Например, их можно использовать в арифметических вычислениях, скажем, можно написать 2 + 'ж', к двойке будет прибавляться кодировка Unicode '\u04i6' буквы 'ж'. В десятичной форме это число 1046 и в результате сложения получим 1048.

Напомним, что в записи 2 + "Ж" плюс понимается как сцепление строк, двойка будет преобразована в строку, в результате получится строка "2ж".

Вещественных типов два: **float** и **double**.

На рис. 1.2 показана иерархия типов данных Java.

Поскольку по имени переменной невозможно определить ее тип, все переменные обязательно должны быть описаны перед их использованием. Описание заключается в том, что записывается имя типа, затем, через пробел, список имен переменных, разделенных запятой. Для всех или некоторых переменных можно указать начальные значения после знака равенства, которыми могут служить любые константные выражения того же типа. Описание каждого типа завершается точкой с запятой. В программе может быть сколько угодно описаний каждого типа.

Замечание для специалистов *Java – язык со строгой типизацией (strongly typed language)*.

Разберем каждый тип подробнее.

Логический тип. Логические операции.

Значения логического типа boolean возникают в результате различных сравнений, вроде 2 > 3, и используются, главным образом, в условных операторах и операторах циклов. Логических значении всего два: **true** (истина) и **false** (ложь). Это служебные слова Java. Описание переменных этого типа выглядит так:

```
boolean b = true, bb = false, bool2;
```

Над логическими данными можно выполнять операции присваивания, например, bool2 = true, в том числе и составные с логическими операциями; сравнение на равенство b == bb и на неравенство b!= bb, а также логические операции.

Логические операции

Логические операции:

- **отрицание** (NOT) **!** (обозначается восклицательным знаком);
- **конъюнкция** (AND) **&** (амперсанд);
- **дизъюнкция** (OR) **|** (вертикальная черта);
- **исключающее ИЛИ** (XOR) **^** (каре).

Они выполняются над логическими данными, их результатом будет тоже логическое значение true или false. Про них можно ничего не знать, кроме того, что представлено в табл. 1.1.

Таблица 1.1. Логические операции.

b1	b2	!b1	b1&b2	b1 | b2	b1^b2
true	true	false	true	true	false
true	false	false	false	true	true
false	true	true	false	true	true
false	false	true	false	false	false

Словами эти правила можно выразить так:

- отрицание меняет значение истинности;
- конъюнкция истинна, только если оба операнда истинны;
- дизъюнкция ложна, только если оба операнда ложны;
- исключающее ИЛИ истинно, только если значения операндов различны.

Замечание

Если бы Шекспир был программистом, фразу "To be or not to be" он написал бы так: 2b \|! 2b.

Кроме перечисленных четырех логических операций есть еще две логические операции сокращенного вычисления:

- **сокращенная конъюнкция** (conditional-AND) **&&**;
- **сокращенная дизъюнкция** (conditional-OR) **||**.

Удвоенные знаки амперсанда и вертикальной черты следует записывать без пробелов.

Правый операнд сокращенных операций вычисляется только в том случае, если от него зависит результат операции, т. е. если левый операнд конъюнкции имеет значение true, или левый операнд дизъюнкции имеет значение false.

Это правило очень удобно и ловко используется, например, можно записывать выражения (n!= 0) && (m/n > 0.001) или (n == 0) || (m/n > 0.001) не опасаясь деления на нуль.

Замечание

Практически всегда в Java используются именно сокращенные логические операции.

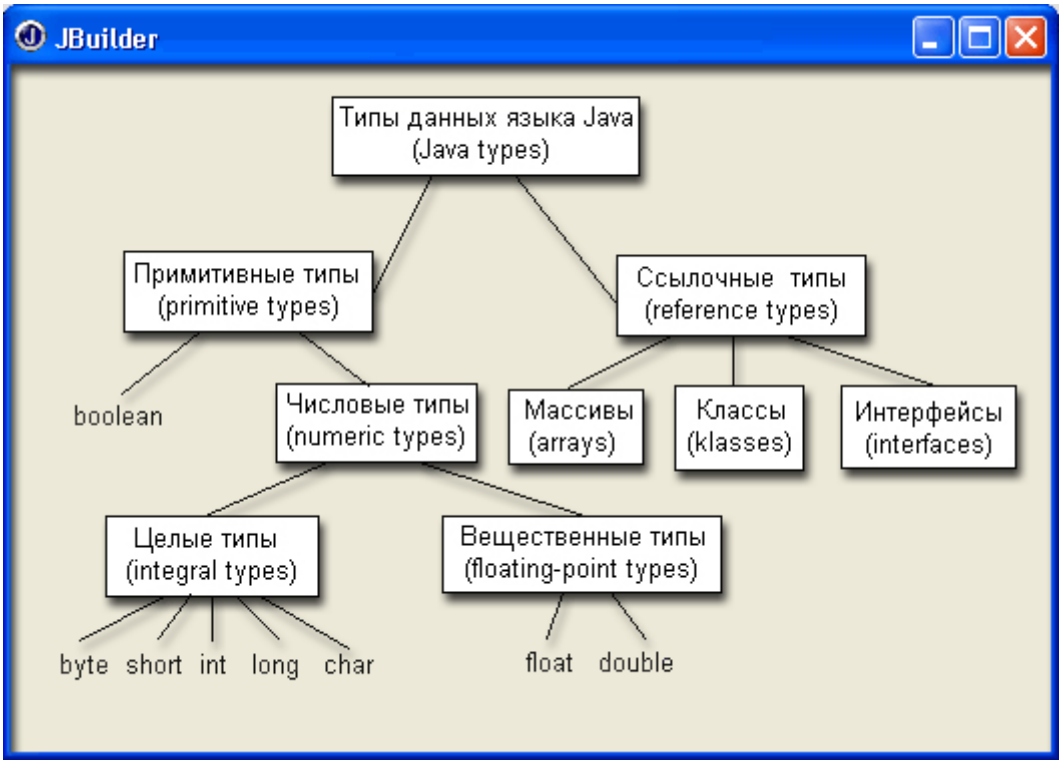

Рис. 1.2. Типы данных языка Java

Значения логического типа boolean возникают в результате различных сравнений, вроде 2 > 3, и используются, главным образом, в условных операторах и операторах циклов. Логических значении всего два: **true** (истина) и **false** (ложь). Это служебные слова Java. Описание переменных этого типа выглядит так:

```
boolean b = true, bb = false, bool2;
```

Над логическими данными можно выполнять операции присваивания, например, bool2 = true, в том числе и составные с логическими операциями; сравнение на равенство b == bb и на неравенство b!= bb, а также логические операции.

Логические операции

Логические операции:

- **отрицание** (NOT) **!** (обозначается восклицательным знаком);
- **конъюнкция** (AND) **&** (амперсанд);
- **дизъюнкция** (OR) **|** (вертикальная черта);
- **исключающее ИЛИ** (XOR) **^** (каре).

Они выполняются над логическими данными, их результатом будет тоже логическое значение true или false. Про них можно ничего не знать, кроме того, что представлено в табл. 1.1.

Таблица 1.1. Логические операции.

b1	b2	!b1	b1&b2	b1 | b2	b1^b2
true	true	false	true	true	false
true	false	false	false	true	true
false	true	true	false	true	true
false	false	true	false	false	false

Словами эти правила можно выразить так:

- отрицание меняет значение истинности;
- конъюнкция истинна, только если оба операнда истинны;
- дизъюнкция ложна, только если оба операнда ложны;
- исключающее ИЛИ истинно, только если значения операндов различны.

Замечание

Если бы Шекспир был программистом, фразу "To be or not to be" он написал бы так: 2b |! 2b.

Кроме перечисленных четырех логических операций есть еще две логические операции сокращенного вычисления:

- **сокращенная конъюнкция** (conditional-AND) **&&**;
- **сокращенная дизъюнкция** (conditional-OR) **||**.

Удвоенные знаки амперсанда и вертикальной черты следует записывать без пробелов.

Правый операнд сокращенных операций вычисляется только в том случае, если от него зависит результат операции, т. е. если левый операнд конъюнкции имеет значение true, или левый операнд дизъюнкции имеет значение false.

Это правило очень удобно и ловко используется, например, можно записывать выражения (n!= 0) && (m/n > 0.001) или (n == 0) || (m/n > 0.001) не опасаясь деления на нуль.

Замечание

Практически всегда в Java используются именно сокращенные логические операции.

Целые типы

Спецификация языка Java, **JLS**, определяет разрядность (количество байтов, выделяемых для хранения значений типа в оперативной памяти) и диапазон значений каждого типа. Для целых типов они приведены в табл. 1.2.

Таблица 1.2. Целые типы.

Тип	Разрядность (байт)	Диапазон
byte	1	от -128 до 127
short	2	от -32768 до 32767
int	4	от -2147483648 до 2147483647
long	8	от -9223372036854775808 до 9223372036854775807
char	2	от '\u0000' до '\uFFFF', в десятичной форме от 0 до 65535

Впрочем, для Java разрядность не столь важна, на некоторых компьютерах она может отличаться от указанной в таблице, а вот диапазон значений должен выдерживаться неукоснительно.

Хотя тип char занимает два байта, в арифметических вычислениях он участвует как тип int, ему выделяется 4 байта, два старших байта заполняются нулями.

Примеры определения переменных целых типов:

```
byte b1 = 50, b2 = -99, b3;
short det = 0, ind = 1;
```

```
int i = -100, j = 100, k = 9999;
long big = 50, veryBig = 2147483648L;
char c1 = 'A', c2 = '?', newLine = '\n';
```

Целые типы хранятся в двоичном виде с дополнительным кодом. Последнее означает, что для отрицательных чисел хранится не их двоичное представление, а **дополнительный код** этого двоичного представления.

Дополнительный же код получается так: в двоичном представлении все нули меняются на единицы, а единицы на нули, после чего к результату прибавляется единица, разумеется, в двоичной арифметике.

Например, значение 50 переменной b1, определенной выше, будет храниться в одном байте с содержимым 00110010, а значение -99 переменной b2 – в байте с содержимым, которое вычисляем так: число 99 переводим в двоичную форму, получая 01100011, меняем единицы и нули, получая 10011100, и прибавляем единицу, получив окончательно байт с содержимым 10011101.

Смысл всех этих сложностей в том, что сложение числа с его дополнительным кодом в двоичной арифметике даст в результате нуль, старший бит просто теряется. Это означает, что в такой странной арифметике дополнительный код числа является противоположным к нему числом, числом с обратным знаком. А это, в свою очередь, означает, что вместо того, чтобы вычесть из числа A число B, можно к A прибавить дополнительный код числа B. Таким образом, операция вычитания исключается из набора машинных операций.

Над целыми типами можно производить массу операций. Их набор восходит к языку C, он оказался удобным и кочует из языка в язык почти без изменений. Особенности применения этих операций в языке Java показаны на примерах.

Операции над целыми типами

Все операции, которые производятся над целыми числами, можно разделить на следующие группы.

Арифметические операции

К арифметическим операциям относятся:

- сложение **+** (плюс);
- вычитание **-** (дефис);
- умножение ***** (звездочка);
- деление **/** (наклонная черта – слэш);
- взятие остатка от деления (деление по модулю) **%** (процент);
- инкремент (увеличение на единицу) **++**;
- декремент (уменьшение на единицу) **--**

Между сдвоенными плюсами и минусами нельзя оставлять пробелы. Сложение, вычитание и умножение целых значений выполняются как обычно, а вот деление целых значений в результате дает опять целое (так называемое "**целое деление**"), например, 5/2 даст в результате 2, а не 2.5, а 5/(-3) даст -1. Дробная часть попросту отбрасывается, происходит усечение частного. Это поначалу обескураживает, но потом оказывается удобным для усечения чисел.

Замечание

В Java принято целочисленное деление.

Это странное для математики правило естественно для программирования: если оба операнда имеют один и тот же тип, то и результат имеет тот же тип. Достаточно написать 5/2.0 или 5.0/2 или 5.0/2.0 и получим 2.5 как результат деления вещественных чисел.

Операция **деление по модулю** определяется так: a % b = a -(a / b) * b; например, 5%2 даст в результате 1, а 5% (-3) даст, 2, т.к. 5 = (-3) * (-1) + 2, но (-5)%3 даст -2, поскольку -5 = 3 * (-1) -2.

Операции **инкремент** и **декремент** означают увеличение или уменьшение значения переменной на единицу и применяются только к переменным, но не к константам или выражениям, нельзя написать 5++ или (a + b)++.

Например, после приведенных выше описаний i++ даст -99, а j-- даст 99.

Интересно, что эти операции можно записать и перед переменной: ++i, --j. Разница проявится только в выражениях: при первой форме записи (**постфиксной**) в выражении участвует старое значение переменной и только потом происходит увеличение или уменьшение ее значения. При второй форме записи (**префиксной**) сначала изменится переменная и ее новое значение будет участвовать в выражении.

Например, после приведенных выше описаний, (k++) + 5 даст в результате 10004, а переменная k примет значение 10000. Но в той же исходной ситуации (++k) + 5 даст 10005, а переменная k станет равной 10000.

Приведение типов

Результат арифметической операции имеет тип int, кроме того случая, когда один из операндов типа long. В этом случае результат будет типа long.

Перед выполнением арифметической операции всегда происходит **повышение**(promotion) типов byte, short, char. Они преобразуются в тип int, а может быть, и в тип long, если другой операнд типа long. Операнд типа int повышается до типа long, если другой операнд типа long. Конечно, числовое значение операнда при этом не меняется.

Это правило приводит иногда к неожиданным результатам. Попытка откомпилировать простую программу, представленную в листинге 1.3, приведет к сообщениям компилятора, показанным на рис. 1.3.

Листинг 1.3. Неверное определение переменной.

```
class InvalidDef{
public static void main (String [] args) {
byte b1 = 50, b2 = -99;
short k = b1 + b2; // Неверно! '
System.out.println("k=" + k);
}
}
```

Эти сообщения означают, что в файле InvalidDef.java, в строке 4, обнаружена возможная потеря точности (**possible loss of precision**). Затем приводятся обнаруженный (**found**) и нужный (**required**) типы, выводится строка, в которой обнаружена (а не сделана) ошибка, и отмечается символ, при разборе которого найдена ошибка. Затем указано общее количество обнаруженных (а не сделанных) ошибок (**1 error**).

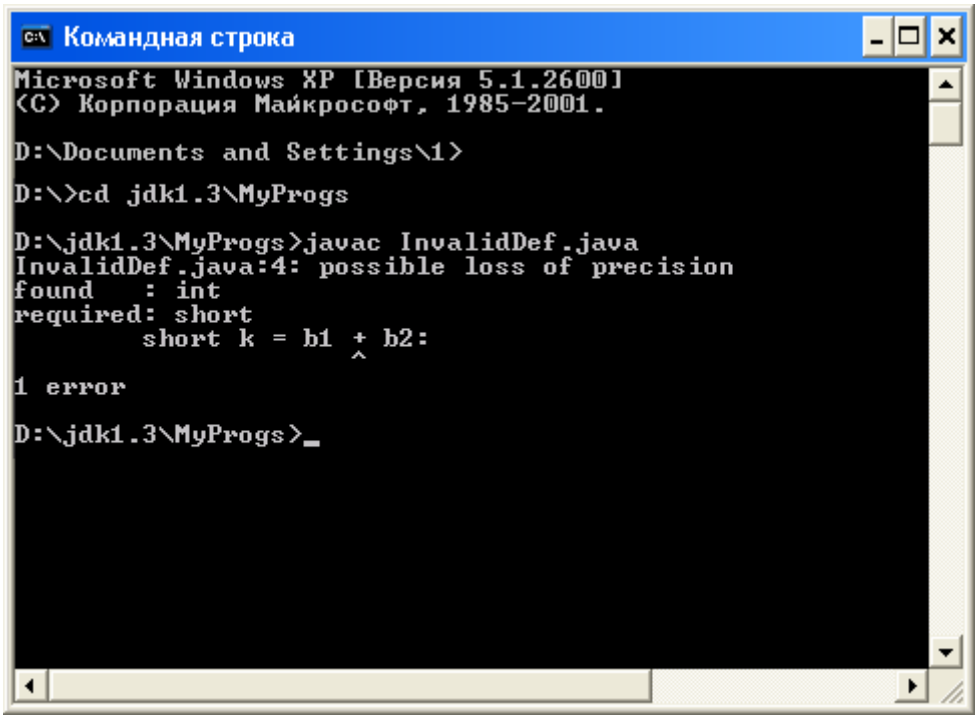

Рис. 1.3. Сообщения компилятора об ошибке

В таких случаях следует выполнить явное приведение типа. В данном случае это будет**сужение** (narrowing) типа int до типа short. Оно осуществляется операцией явного приведения, которая записывается перед приводимым значением в виде имени типа в скобках. Определение:

```
short k = (short)(b1 + b2);
```

…будет верным.

Сужение осуществляется просто отбрасыванием старших битов, что необходимо учитывать для больших значений. Например, определение:

```
byte b = (byte) 300;
```

…даст переменной b значение 44. Действительно, в двоичном представлении числа 300, равном 100101100, отбрасывается старший бит и получается 00101100.

Таким же образом можно произвести и явное **расширение** (widening) типа, если в этом есть необходимость..

Если результат целой операции выходит за диапазон своего типа int или long, то автоматически происходит приведение по модулю, равному длине этого диапазона, и вычисления продолжаются, переполнение никак не отмечается.

Замечание

В языке Java нет целочисленного переполнения.

Операции сравнения

В языке Java шесть обычных операций сравнения целых чисел по величине:

- больше **>**;
- меньше **<**;
- больше или равно **>=**;
- меньше или равно **<=**;
- равно **==**;
- не равно **!=**.

Сдвоенные символы записываются без пробелов, их нельзя переставлять местами, запись **›** будет неверной.

Результат сравнения – логическое значение: true, в результате, например, сравнения 3!= 5; или false, например, в результате сравнения 3 == 5.

Для записи сложных сравнений следует привлекать логические.операции. Например, в вычислениях часто приходится делать проверки вида a < x < b. Подобная запись на языке Java приведет к сообщению об ошибке, поскольку первое сравнение, a < x, даст true или false, а Java не знает, больше это, чем b, или меньше. В данном случае следует написать выражение (a < x) && (x < b), причем здесь скобки можно опустить, написать просто a < x && x < b, но об этом немного позднее.

Побитовые операции

Иногда приходится изменять значения отдельных битов в целых данных. Это выполняется с помощью побитовых (bitwise) операций путем наложения маски. В языке Java есть четыре побитовые операции:

- дополнение (complement) **~** (тильда);
- побитовая конъюнкция (bitwise AND) **&**;
- побитовая дизъюнкция (bitwise OR) **|**;
- побитовое исключающее ИЛИ (bitwise XOR) **^**.

Они выполняются поразрядно, после того как оба операнда будут приведены к одному типу int или long, так же как и для арифметических операций, а значит, и к одной разрядности. Операции над каждой парой битов выполняются согласно табл. 1.3.

Таблица 1.3. Побитовые операции.

n1	n2	~n1	n1 & n2	n1 | n2	n1 ^ n2
1	1	0	1	1	0
1	0	0	0	1	1
0	1	1	0	1	1
0	0	1	0	0	0

В нашем примере b1 == 50, двоичное представление 00110010, b2 == -99, двоичное представление 10011101. Перед операцией происходит повышение до типа int. Получаем представления из 32-х разрядов для b1-0…00110010, для b2-1…l0011101. В результате побитовых операций получаем:

- ~b2 == 98, двоичное представление 0…01100010;
- b1 & b2 == 16, двоичное представление 0…00010000;
- b1 | b2 == – 65, двоичное представление 1…10111111;
- b1 ^ b2 == – 81, двоичное представление 1…10101111.

Двоичное представление каждого результата занимает 32 бита.

Заметьте, что дополнение ~x всегда эквивалентно (-x)-1.

Сдвиги

В языке Java есть три операции сдвига двоичных разрядов:

- сдвиг влево **<<**;
- сдвиг вправо **>>**;
- беззнаковый сдвиг вправо **>>>**.

Эти операции своеобразны тем, что левый и правый операнды в них имеют разный смысл. Слева стоит значение целого типа, а правая часть показывает, на сколько двоичных разрядов сдвигается значение, стоящее в левой части.

Например, операция b1<< 2 сдвинет влево на 2 разряда предварительно повышенное значение 0…00110010 переменной b1, что даст в результате 0…011001000, десятичное 200. Освободившиеся справа разряды заполняются нулями, левые разряды, находящиеся за 32-м битом, теряются.

Операция b2 << 2 сдвинет повышенное значение 1…10011101 на два разряда влево. В результате получим 1…1001110100, десятичное значение -396.

Заметьте, что сдвиг влево на n разрядов эквивалентен умножению числа на 2 в степени n.

Операция b1 >> 2даст в результате 0…00001100, десятичное 12, а b2 >> 2 – результат 1..11100111, десятичное -25, т. е. слева распространяется старший бит, правые биты теряются. Это так называемый **арифметический сдвиг**.

Операция беззнакового сдвига во всех случаях ставит слева на освободившиеся места нули, осуществляя **логический сдвиг. Но** вследствие предварительного повышения это имеет эффект только для нескольких старших разрядов отрицательных чисел. Так, b2 >>> 2 имеет результатом 001…100111, десятичное число 1 073 741 799.

Если же мы хотим получить логический сдвиг исходного значения loomoi переменной b2, т. е., 0…00100111, надо предварительно наложить на b2 **маску**, обнулив старшие биты:

```
(b2 & 0XFF) >>> 2.
```

Замечание

Будьте осторожны при использовании сдвигов вправо.

Вещественные типы

Вещественных типов в Java два: **float** и **double**. Они характеризуются разрядностью, диапазоном значений и точностью представления, отвечающим стандарту IEEE 754-1985 с некоторыми изменениями. К обычным вещественным числам добавляются еще три значения:

1. Положительная бесконечность, выражаемая константой **POSITIVE_INFINITY** и возникающая при переполнении положительного значения, например, в результате операции умножения 3.0*6e307.
2. Отрицательная бесконечность **NEGATIVE_INFINITY**.
3. "Не число", записываемое константой **NaN** (Not a Number) и возникающее при делении вещественного числа на нуль или умножении нуля на бесконечность.

В **главе 4** мы поговорим о них подробнее.

Кроме того, стандарт различает положительный и отрицательный нуль, возникающий при делении на бесконечность соответствующего знака, хотя сравнение 0.0 == -0.0 дает true.

Операции с бесконечностями выполняются по обычным математическим правилам.

Во всем остальном вещественные типы – это обычные, вещественные значения, к которым применимы все арифметические операции и сравнения, перечисленные для целых типов. Характеристики вещественных типов приведены в табл. 1.4.

Знатокам ***C/C++***

В языке Java взятие остатка от деления %, инкремент ++ и декремент -- применяются и к вещественным типам.

Таблица 1.4. Вещественные типы.

Тип	Разрядность	Диапазон	Точность
float	4	3.4e-38 < |x| < 3.4e38	7-8 цифр
double	8	1.7e-308<|x|<1.7e308	17 цифр

Примеры определения вещественных типов:

```
float x = 0.001, y = -34.789;
double 21 = -16.2305, z2;
```

Поскольку к вещественным типам применимы все арифметические операции и сравнения, целые и вещественные значения можно смешивать в операциях. При этом правило приведения типов дополняется такими условиями:

- если в операции один операнд имеет тип double, то и другой приводится к типу double;
- если один операнд имеет тип float, то и другой приводится к типу float;
- в противном случае действует правило приведения целых значений.

Операции присваивания. Условная операция.

Простоя операция присваивания (simple assignment operator) записывается знаком равенства **=**, слева от которого стоит переменная, а справа выражение, совместимое с типом переменной:

```
x = 3.5, y = 2 * (x -0.567) / (x + 2), b = x < y, bb = x >= y && b.
```

Операция присваивания действует так: выражение, стоящее после знака равенства, вычисляется и приводится к типу переменной, стоящей слева от знака равенства. Результатом операции будет приведенное значение правой части.

Операция присваивания имеет еще одно, побочное, действие: переменная, стоящая слева, получает приведенное значение правой части, старое ее значение теряется.

В операции присваивания левая и правая части неравноправны, нельзя написать 3.5 = x. После операции x = y изменится переменная x, став равной y, а после y = x изменится y.

Кроме простой операции присваивания есть еще 11 **составных** операций присваивания (**compound assignment operators**):

```
+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=; >>>=.
```

Символы записываются без пробелов, нельзя переставлять их местами.

Все составные операции присваивания действуют по одной схеме:

```
x op= a эквивалентно x = (тип x), т. е. (x op a).
```

Напомним, что переменная ind типа short определена у нас со значением 1. Присваивание ind +=7.8 даст в результате число 8, то же значение получит и переменная ind. Эта операция эквивалентна простой операции присваивания ind = (short)(ind + 7.8).

Перед присваиванием, при необходимости, автоматически производится приведение типа. Поэтому:

```
byte b = 1;
b = b + 10; // Ошибка!
b += 10; // Правильно!
```

Перед сложением b + 50 происходит повышение b до типа int, результат сложения тоже будет типа int и, в первом случае, не может быть присвоен переменной b без явного приведения типа. Во втором случае перед присваиванием произойдет сужение результата сложения до типа byte.

Условная операция

Эта своеобразная операция имеет три операнда. Вначале записывается произвольное логическое выражение, т. е. имеющее в результате true или false, затем знак вопроса, потом два произвольных выражения, разделенных двоеточием, например:

```
x < 0? 0: x
x > y? x -y: x + y
```

Условная операция выполняется так. Сначала вычисляется логическое выражение. Если получилось значение true, то вычисляется первое выражение после вопросительного знака? и его значение будет результатом всей операции. Последнее выражение при этом не вычисляется. Если же получилось значение false, то вычисляется только последнее выражение, его значение будет результатом операции.

Это позволяет написать n == 0? да: m / n не опасаясь деления на нуль. Условная операция поначалу кажется странной, но она очень удобна для записи небольших разветвлений.

Выражения

Из констант и переменных, операций над ними, вызовов методов и скобок составляются**выражения** (expressions). Разумеется, все элементы выражения должны быть совместимы, нельзя написать, например, 2 + true. При вычислении выражения выполняются четыре правила:

1. 1. Операции одного приоритета вычисляются слева направо: x + y + z вычисляется как (x + y) + z. Исключение: операции присваивания вычисляются справа налево: x = y = z вычисляется как x = (y = z).
2. 2. Левый операнд вычисляется раньше правого.
3. 3. Операнды полностью вычисляются перед выполнением операции.
4. 4. Перед выполнением составной операции присваивания значение левой части сохраняется для использования в правой части.

Следующие примеры показывает особенности применения первых трех правил. Пусть:

```
int a = 3, b = 5;
```

Тогда результатом выражения b + (b = 3) будет число 8; но результатом выражения (b = 3) + b будет число 6. Выражение b += (b = 3) даст в результате 8, потому что вычисляется как первое из приведенных выше выражений.

Знатокам ***C/C++***

Большинство компиляторов языка C++ во всех этих случаях вычислят значение 8.

Четвертое правило можно продемонстрировать так. При тех же определениях а и ь в результате вычисления выражения b += a += b += 7 получим 20. Хотя операции присваивания выполняются справа налево и после первой, правой, операции значение ь становится равным 12, но в последнем, левом, присваивании участвует старое значение ь, равное 5. А в результате двух последовательных вычислений a += b += 7; b += a; получим 27, поскольку во втором выражении участвует уже новое значение переменной b, равное 12.

Знатокам ***C/C++***

Большинство компиляторов C++ в обоих случаях вычислят 27.

Выражения могут иметь сложный и запутанный вид. В таких случаях возникает вопрос о приоритете операций, о том, какие операции будут выполнены в первую очередь. Естественно, умножение и деление производится раньше сложения и вычитания. Остальные правила перечислены в следующем разделе.

Порядок вычисления выражения всегда можно отрегулировать скобками, их можно оставить сколько угодно. Но здесь важно соблюдать "золотую середину". При большом количестве скобок снижается наглядность выражения и легко ошибиться в расстановке скобок. Если выражение со скобками корректно, то компилятор может отследить только парность скобок, но не правильность их расстановки.

Приоритет операций

Операции перечислены в порядке убывания приоритета. Операции на одной строке имеют одинаковый приоритет.

1. Постфиксные операции **++** и **--**.

2. Префиксные операции **++** и **--**, дополнение **~** и отрицание **!**.
3. Приведение типа (тип).
4. Умножение *****, деление **/** и взятие остатка **%**.
5. Сложение **+** и вычитание **-**.
6. Сдвиги **<<**, **>>**, **>>>**.
7. Сравнения **>**, **<**, **>=**, **<=**.
8. Сравнения **==**, **!=**.
9. Побитовая конъюнкция **&**.
10. Побитовое исключающее ИЛИ **^**.
11. Побитовая дизъюнкция **|**.
12. Конъюнкция **&&**.
13. Дизъюнкция **||**.
14. Условная операция **?:**.
15. Присваивания **=**, **+=**, **-=**, ***=**, **/=**, **%=**, **&=**, **^=**, **|=**, **<<**, **>>**, **>>>**.

Здесь перечислены не все операции языка Java, список будет дополняться по мере изучения новых операций.

Знатокам ***C/C++***

*В Java нет операции "запятая", но **список выражений** используется в операторе цикла for*.

Операторы. Блок. Операторы присваивания.

Как вы знаете, любой алгоритм, предназначенный для выполнения на компьютере, можно разработать, используя только линейные вычисления, разветвления и циклы.

Записать его можно в разных формах: в виде блок-схемы, на псевдокоде, на обычном языке, как мы записываем кулинарные рецепты, или как-нибудь еще "алгоритмы".,-.

Всякий язык программирования должен иметь средства записи алгоритмов. Они называются **операторами** (statements) языка. Минимальный набор операторов должен содержать оператор для записи линейных вычислений, условный оператор для записи разветвлении и оператор цикла.

Обычно состав операторов языка программирования шире: для удобства записи алгоритмов в язык включаются несколько операторов цикла, оператор варианта, операторы перехода, операторы описания объектов.

Набор операторов языка Java включает:

- операторы описания переменных и других объектов (они были рассмотрены выше);
- операторы-выражения;
- операторы присваивания;
- условный оператор **if**;
- три оператора цикла **while**, **do-while**, **for**;
- оператор варианта **switch**;
- Операторы перехода **break**, **continue** и **return**;
- блок **{}**;
- пустой оператор – просто точка с запятой.

Здесь приведен не весь набор операторов Java, он будет дополняться по мере изучения языка.

Замечание
В языке Java нет оператора goto.

Всякий оператор завершается точкой с запятой.

Можно поставить точку с запятой в конце любого выражения, и оно станет оператором (**expression statement**). Но смысл это имеет только для операций присваивания, инкремента и декремента и вызовов методов. В остальных случаях это бесполезно, потому что вычисленное значение выражения потеряется.

Знатокам Pascal
Точка с запятой в Java не разделяет операторы, а является частью оператора.

Линейное выполнение алгоритма обеспечивается последовательной записью операторов. Переход со строки на строку в исходном тексте не имеет никакого значения для компилятора, он осуществляется только для наглядности и читаемости текста.

Блок

Блок заключает в себе нуль или несколько операторов с целью использовать их как один оператор в тех местах, где по правилам языка можно записать только один оператор. Например:

```
{x = 5; y =?;}.
```

Можно записать и пустой блок, просто пару фигурных скобок **{}**.

Блоки операторов часто используются для ограничения области действия переменных и просто для улучшения читаемости текста программы.

Операторы присваивания

Точка с запятой в конце любой операции присваивания превращает ее в оператор присваивания. Побочное действие операции – присваивание – становится в операторе основным.

Разница между операцией и оператором присваивания носит лишь теоретический характер. Присваивание чаще используется как оператор, а не операция.

Условный оператор

Условный оператор (**if-then-else statement**) в языке Java записывается так:

```
if (логВыр) оператор1 else оператор2
```

…и действует следующим образом. Сначала вычисляется логическое выражение**логвыр**. Если результат true, то действует **оператор1** и на этом действие условного оператора завершается, **оператор2** не действует, далее будет выполняться следующий за**if** оператор. Если результат false, то действует **оператор2**, при этом оператор1 вообще не выполняется.

Условный оператор может быть сокращенным (**if-then statement**):

```
if (логВыр) оператор1
```

…и в случае false не выполняется ничего.

Синтаксис языка не позволяет записывать несколько операторов ни в ветви then, ни в ветви else. При необходимости составляется блок операторов в фигурных скобках. Соглашения "Code Conventions" рекомендуют всегда использовать фигурные скобки и размещать оператор на нескольких строках с отступами, как в следующем примере:

```
if (a < x) {
x = a + b; } else {
x = a -b;
}
```

Это облегчает добавление операторов в каждую ветвь при изменении алгоритма. Мы не будем строго следовать этому правилу, чтобы не увеличивать объем книги.

Очень часто одним из операторов является снова условный оператор, например:

```
if (n == 0}{
sign = 0;
} else if (n < 0){
sign = -1;
} else {
sign = 1;
}
```

При этом может возникнуть такая ситуация ("dangling else"):

```
int ind = 5, x = 100;
if (ind >= 10) if (ind <= 20) x = 0; else x = 1;
```

Сохранит переменная x значение 100 или станет равной 1? Здесь необходимо волевое решение, и общее для большинства языков, в. том числе и Java, правило таково: ветвь else относится к ближайшему слева условию if, не имеющему своей ветви else. Поэтому в нашем примере переменная x останется равной 100.

Изменить этот порядок можно с помощью блока:

```
if (ind > 10) {if (ind < 20) x = 0; else x = 1;}
```

Вообще не стоит увлекаться сложными вложенными условными операторами. Проверки условий занимают много времени. По возможности лучше использовать логические операции, например, в нашем примере можно написать:

```
if (ind >= 10 && ind <= 20) x = 0; else x = 1;
```

В листинге 1.4 вычисляются корни квадратного уравнения $ax^2 + bx + c = 0$ для любых коэффициентов, в том числе и нулевых.

Листинг 1.4. Вычисление корней квадратного уравнения.

```
class QuadraticEquation{
public static void main(String[] args){
double a = 0.5, b = -2.7, c = 3.5, d, eps=le-8;
```

```
if (Math.abs(a) < eps)
if (Math.abs(b) < eps)
if (Math.abs(c) < eps) // Все коэффициенты равны нулю
System.out.println("Решение – любое число");
else
System.out.println("Решений нет");
else
System.out.println("xl = x2 = " +(-c / b));
else { // Коэффициенты не равны нулю
if((d = b**b -4*a*c)< 0.0){ // Комплексные корни
d = 0.5 * Math.sqrt(-d) / a;
a = - 0.5 * b/ a;
System.out.println("xl = " +a+ " +i " +d+
",x2 = " +a+ " -i " +d);
} else {
// Вещественные корни
d =0.5 * Math.sqrt(d) / a;
a = -0.5 * b / a;
System.out.println("x1 = " + (a + d) + ", x2 = " +(a -d));
}
}
)
}
```

В этой программе использованы методы вычисления модуля abs() и кратного корня sqrt () вещественного числа из встроенного в Java API класса Math. Поскольку все вычисления с вещественными числами производятся приближенно, мы считаем, что коэффициент уравнения равен нулю, если его модуль меньше 0.00000001. Обратите внимание на то, как в методе println () используется сцепление строк, и на то, как операция присваивания при вычислении дискриминанта вложена в логическое выражение.

Продвинутым" пользователям

Вам уже хочется вводить коэффициенты a, b и c прямо с клавиатуры? Пожалуйста, используйте метод System, in .read (byte [ ] bt), но учтите, что этот метод записывает вводимые цифры в массив байтов bt в кодировке ASCII, в каждый байт по одной цифре. Массив байтов затем надо преобразовать в вещественное число, например, методом Double(new String(bt)).doubleValue0. Непонятно? Но это еще не все, нужно обработать исключительные ситуации, которые могут возникнуть при вводе (см. главу 18).

Операторы цикла

Основной оператор цикла – оператор while – выглядит так:

```
while (логВыр) оператор
```

Вначале вычисляется логическое выражение **логВыр;** если его значение true, то выполняется оператор, образующий цикл. Затеем снова вычисляется **лог-выр** и действует оператор, и так до тех пор, пока не получится значение false. Если **логВыр** изначально равняется false, то **оператор** не будет выполнен ни разу. Предварительная проверка обеспечивает безопасность выполнения цикла, позволяет избежать переполнения, деления на нуль и других неприятностей. Поэтому оператор while является основным, а в некоторых языках и единственным оператором цикла.

Оператор в цикле может быть и пустым, например, следующий фрагмент кода:

```
int i = 0;
double s = 0.0;
while ((s += 1.0 / ++i) < 10);
```

…вычисляет количество i сложений, которые необходимо сделать, чтобы гармоническая сумма s достигла значения 10. Такой стиль характерен для языка C. Не стоит им увлекаться, чтобы не превратить текст программы в шифровку, на которую вы сами через пару недель будете смотреть с недоумением.

Можно организовать и бесконечный цикл:

```
while (true) оператор
```

Конечно, из такого цикла следует предусмотреть какой-то выход, например, оператором break, как в листинге 1.5. В противном случае программа зациклится, и вам придется прекращать ее выполнение "комбинацией из трех пальцев" **CTRL + ALT + Del** в MS Windows 95/98/ME, комбинацией **CTRL + C** в UNIX или через Task Manager в Windows NT/2000.

Если в цикл надо включить несколько операторов, то следует образовать блок операторов **{}.**

Второй оператор цикла – оператор do-while – имеет вид:

```
do оператор while (логВыр)
```

Здесь сначала выполняется оператор, а потом происходит вычисление логического выражения логвыр. Цикл выполняется, пока логвыр остается равным true.

Знатокам Pascal

В цикле do-while проверяется условие продолжения, а не окончания цикла.

Существенное различие между этими двумя операторами цикла только в том, что в цикле do-while оператор обязательно выполнится хотя бы один раз.

Например, пусть задана какая-то функция f(x), имеющая на отрезке,[o; b] ровно один корень. В листинге 1.5 приведена программа, вычисляющая этот корень приближенно методом деления пополам (бисекции, дихотомий).

Листинг 1.5. Нахождение корня нелинейного уравнения методом бисекции.

```
class Bisection{
static double f(double x){
return x*x*x -3*x*x +3; // Или что-то другое
}
public static void main(String!] args){
double a = 0.0, b = 1.5, c, y, eps = le-8;
do{
c = 0.5 *(a + b); y = f(c);
if (Math.abs(y) < eps) break;
// Корень найден. Выходим из цикла
// Если на концах отрезка [a; c]
// функция имеет разные знаки:
if (f (a) * y < 0.0) b = c;
// Значит, корень здесь. Переносим точку b в точку c
//В противном случае:
else a = c;
// Переносим точку a в точку c
// Продолжаем, пока отрезок [a; b] не станет мал
} while (Math, abs (b-a) >= eps);
System.out.println("x = " +c+ ", f(" +c+ ") = " +y);
}
```

```
}
```

Класс Bisection сложнее предыдущих примеров: в нем кроме метода main () есть еще метод вычисления функции f(x). Здесь метод f () очень прост: он вычисляет значение многочлена и возвращает его в качестве значения функции, причем все это выполняется одним оператором:

```
return выражение
```

В методе main () появился еще один новый оператор break, который просто прекращает выполнение цикла, если мы по счастливой случайности наткнулись на приближенное значение корня. Внимательный читатель заметил и появление модификатора static в объявлении метода f(). Он необходим потому, что метод f () вызывается из статического метода main ().

Третий оператор цикла – оператор for – выглядит так:

```
for (списокВыр; логВыр; слисокВыр2) оператор
```

Перед выполнением цикла вычисляется список выражений **списокВыр1**. Это нуль или несколько выражений, перечисленных через запятую. Они вычисляются слева направо, и в следующем выражении уже можно использовать результат предыдущего выражения. Как правило, здесь задаются начальные значения переменным цикла.

Затем вычисляется логическое выражение логвьр. Если оно истинно, true, то действует оператор, потом вычисляются слева направо выражения из списка выражений **списокВыр2**. Далее снова проверяется **логвыр**. Если оно истинно, то выполняется оператор и**списокВыр2** и т. д. Как только логВыр станет равным false, выполнение цикла заканчивается.

Короче говоря, выполняется последовательность операторов:

```
списокВыр1; while (логВыр){
оператор
слисокВыр2; }
```

…с тем исключением, что, если оператором в цикле является оператор **continue**, то слисоквыр2 все-таки выполняется.

Вместо **списокВыр1** может стоять одно определение переменных обязательно с начальным значением. Такие переменные известны только в пределах этого цикла.

Любая часть оператора for может отсутствовать: цикл может быть пустым, выражения в заголовке тоже, при этом точки с запятой сохраняются. Можно задать бесконечный цикл:

```
for (;;) оператор
```

В этом случае в теле цикла следует предусмотреть какой-нибудь выход.

Хотя в операторе for заложены большие возможности, используется он, главным образом, для перечислений, когда их число известно, например, фрагмент кода:

```
int s=0;
for (int k = 1; k <= N; k++) s += k * k;
// Здесь переменная k уже неизвестна
```

…вычисляет сумму квадратов первых N натуральных чисел.

Оператор continue и метки. Оператор break.

Оператор **continue** используется только в операторах цикла. Он имеет две формы. Первая форма состоит только из слова continue и осуществляет немедленный переход к следующей итерации цикла.

В очередном фрагменте кода оператор **continue** позволяет обойти деление на нуль:

```
for (int i = 0; i < N; i++){
if (i '== j) continue;
s += 1.0 / (i -j);
}
```

Вторая форма содержит метку:

```
continue метка
```

Метка записывается, как все идентификаторы, из букв Java, цифр и знака подчеркивания, но не требует никакого описания. Метка ставится перед оператором или открывающей фигурной скобкой и отделяется от них двоеточием. Так получается**помеченный оператор** или **помеченный блок.**

Знатокам Pascal

Метка не требует описания и не может начинаться с цифры.

Вторая форма используется только в случае нескольких вложенных циклов для немедленного перехода к очередной итерации одного из объемлющих циклов, а именно, помеченного цикла.

Оператор break

Оператор **break** используется в операторах цикла и операторе варианта для немедленного выхода из этих конструкций.

Оператор **break метка** применяется внутри помеченных операторов цикла, оператора варианта или помеченного блока для немедленного выхода за эти операторы. Следующая схема поясняет эту конструкцию.

```
M1: { // Внешний блок
M2: { // Вложенный блок – второй уровень
M3: { // Третий уровень вложенности…
if (что-то случилось) break M2;
// Если true, то здесь ничего не выполняется
}
// Здесь тоже ничего не выполняется
}
// Сюда передается управление
}
```

Поначалу сбивает с толку то обстоятельство, что метка ставится перед блоком или оператором, а управление передается за этот блок или оператор. Поэтому не стоит увлекаться оператором **break** с **меткой**.

Оператор варианта

Оператор варианта **switch** организует разветвление по нескольким направлениям. Каждая ветвь отмечается константой или константным выражением какого-либо целого типа (кроме long) и выбирается, если значение определенного выражения совпадет с этой константой. Вся конструкция выглядит так.

```
switch (целВыр){
case констВыр1: оператор1
case констВыр2: оператор2
…..
case констВырN: операторN
default: операторDef
}
```

Стоящее в скобках выражение **целвыр** может быть типа byte, short, int, char, но не long. Целые числа или целочисленные выражения, составленные из констант, **констВыр** тоже не должны иметь тип long.

Оператор варианта выполняется так. Все константные выражения вычисляются заранее, на этапе компиляции, и должны иметь отличные друг от друга значения. Сначала вычисляется целочисленное выражение целВыр. Если оно совпадает с одной из констант, то выполняется оператор, отмеченный этой константой. Затем выполняются ("fall through labels") все следующие операторы, включая и **операторDef**, и работа оператора варианта заканчивается.

Если же ни одна константа не равна значению выражения, то выполняется **операторDef**и все следующие за ним операторы. Поэтому ветвь default должна записываться последней. Ветвь default может отсутствовать, тогда в этой ситуации оператор варианта вообще ничего не делает.

Таким образом, константы в вариантах case играют роль только меток, точек входа в оператор варианта, а далее выполняются все оставшиеся операторы в порядке их записи.

Знатокам Pascal

После выполнения одного варианта оператор switch продолжает выполнять все оставшиеся варианты.

Чаще всего необходимо "пройти" только одну ветвь операторов. В таком случае используется оператор **break**, сразу же прекращающий выполнение оператора switch. Может понадобиться выполнить один и тот же оператор в разных ветвях case. В этом случае ставим несколько меток case подряд. Вот простой пример.

```
switch(dayOfWeek){
case 1: case 2: case 3: case 4:case 5:
System.out.println("Week-day");, break;
case 6: case 7:
System.out.println("Week-end"); break;
default:
System.out.printlnt"Unknown day");
}
```

Замечание

Не забывайте завершать варианты оператором break.

Массивы

Как всегда в программировании **массив** – это совокупность переменных одного типа, хранящихся в смежных ячейках оперативной памяти.

Массивы в языке Java относятся к ссылочным типам и описываются своеобразно, но характерно для ссылочных типов. Описание производится в три этапа.

Первый этап – **объявление** (declaration). На этом этапе определяется только переменная типа **ссылка** (reference) **на массив**, содержащая тип массива. Для этого записывается имя типа элементов массива, квадратными скобками указывается, что объявляется ссылка на массив, а не простая переменная, и перечисляются имена переменных типа ссылка, например:

```
double[] a, b;
```

Здесь определены две переменные – ссылки a и b на массивы типа double. Можно поставить квадратные скобки и непосредственно после имени. Это удобно делать среди определений обычных переменных:

```
int I = 0, ar[], k = -1;
```

Здесь определены две переменные целого типа i и k, и объявлена ссылка на целочисленный массив ar.

Второй этап – **определение** (installation). На этом этапе указывается количество элементов массива, называемое его **длиной**, выделяется место для массива в оперативной памяти, переменная-ссылка получает адрес массива. Все эти действия производятся еще одной операцией языка Java – операцией new **тип**, выделяющей участок в оперативной памяти для объекта указанного в операции типа и возвращающей в качестве результата адрес этого участка. Например:

```
a = new double[5];
b = new double[100];
ar = new int[50];
```

Индексы массивов всегда начинаются с о. Массив a состоит из пяти переменных a[0], a[1],, a[4]. Элемента a[5] в массиве нет. Индексы можно задавать любыми целочисленными выражениями, кроме типа long, например:

```
a[i+j], a[i%5], a[++i].
```

Исполняющая система Java следит за тем, чтобы значения этих выражений не выходили за границы длины массива.

Третий этап – **инициализация** (initialization). На этом этапе элементы массива получают начальные значения. Например:

```
a[0] = 0.01; a[1] = -3.4; a[2] = 2:.89; a[3] = 4.5; a[4] = -6.7;
for (int i = 0; i < 100; i++) b[i] = 1.0 /i;
for (int i = 0; i < 50; i++) ar[i] = 2 * i + 1;
```

Первые два этапа можно совместить:

```
double[] a = new double[5], b = new double[100];
int i = 0, ar[] = new int[50], k = -1;
```

Можно сразу задать и начальные значения, записав их в фигурных скобках через запятую в виде констант или константных выражений. При этом даже необязательно указывать количество элементов массива, оно будет равно количеству начальных значений:

```
double[] a = {0.01, -3.4, 2.89, 4.5, -6.7};
```

Можно совместить второй и третий этап:

```
a = new double[] {0.1, 0.2, -0.3, 0.45, -0.02};
```

Можно даже создать безымянный массив, сразу же используя результат операции **new**, например, так:

```
System.out.println(new char[] {'H', 'e', '1', '1', '0'});
```

Ссылка на массив не является частью описанного массива, ее можно перебросить на другой массив того же типа операцией присваивания. Например, после присваивания a = b обе ссылки a и b указывают на один и тот же массив из 100 вещественных переменных типа double и содержат один и тот же адрес.

Ссылка может присвоить "пустое" значение null, не указывающее ни на какой адрес оперативной памяти:

```
ar = null;
```

После этого массив, на который указывала данная ссылка, теряется, если на него не было других ссылок.

Кроме простой операции присваивания, со ссылками можно производить еще только сравнения на равенство, например, a = b, и неравенство, a! = b. При этом сопоставляются адреса, содержащиеся в ссылках, мы можем узнать, не ссылаются ли они на один и тот же массив.

Замечание для специалистов

Массивы в Java всегда определяются динамически, хотя ссылки на них задаются статически.

Кроме ссылки на массив, для каждого массива автоматически определяется целая константа с одним и тем же именем **length**. Она равна длине массива. Для каждого массива имя этой константы уточняется именем массива через точку. Так, после наших определений, константа a.length равна 5, константа b.length равна 100, а ar.length равна 50.

Последний элемент массива a можно записать так: a [a.length -1], предпоследний – a [a.length -2] и т. д. Элементы массива обычно перебираются в цикле вида:

```
double aMin = a[0], aMax = aMin;
for (int i = 1; i < a.length; i++){
if <a[i] < aMin) aMin = a[i];
if (a[i] > aMax) aMax = a[i];
}
double range = aMax – aMin;
```

Здесь вычисляется диапазон значений массива.

Элементы массива – это обыкновенные переменные своего типа, с ними можно производить все операции, допустимые для этого типа:

```
(a[2] + a[4]) / a[0] и т. д.
```

Знатокам C/C++

Массив символов в Java не является строкой, даже если он заканчивается нуль-символом ' \u0000'.

Многомерные массивы

Элементами массивов в Java могут быть снова массивы. Можно объявить:

```
char[] [] c;
```

…что эквивалентно:

```
char c[] c[];
```

…или:

```
char c[][];
```

Затем определяем внешний массив:

```
c = new char[3][];
```

Становится ясно, что c – массив, состоящий из трех элементов-массивов. Теперь определяем его элементы-массивы:

```
c[0] = new char[2];
c[1] = new char[4];
c[2] = new char[3];
```

После этих определений переменная:

```
c.length равна 3,
c[0].length равна 2,
c[1].length равна 4 и
c[2.length равна 3.
```

Наконец, задаем начальные значения:

```
c [0] [0] = 'a',
c[0][1] = 'r',
c[1][0] = 'r',
c[1][1] = 'a',
c[1][2] = 'y' и т.д.
```

Замечание

Двумерный массив в Java не обязан быть прямоугольным.

Описания можно сократить:

```
int[] [] d = new int[3] [4];
```

…а начальные значения задать так:

```
int[][] inds = {{1, 2, 3}, {4, 5, 6}};
```

В листинге 1.6 приведен пример программы, вычисляющей первые 10 строк треугольника Паскаля, заносящей их в треугольный массив и выводящей его элементы на экран. Рис. 1.4 показывает вывод этой программы.

Листинг 1.6. Треугольник Паскаля.

```
class PascalTriangle{
public static final int LINES = 10; // Так определяются константы
public static void main(String[] args) {
int[][] p, = new int [LINES] [];
p[0] = new int[1];
System, out .println (p [0] [0] = 1);
p[l] = new int[2];
p[l][0] = p[l][1] = 1;
System.out.println(p[1][0] + " " + p[l][l]);
for (int i = 2; i < LINES; i++){
p[i] = new int[i+l];
System.out.print((p[i][0] = 1) + " ");
for (int j = 1; j < i; j++)
System.out .print ((p[i] [j] =p[i-l][j-l] – bp[i-l][j]) + " ");
System, out .println (p [ i] [i] = 1)
}
```

```
}
}
```

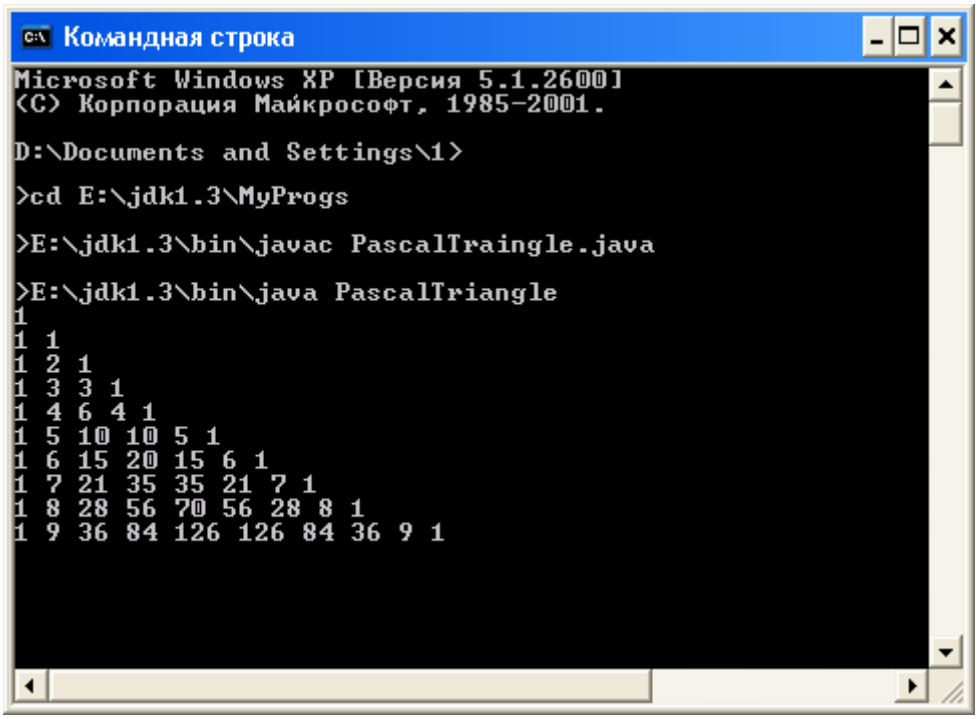

Рис. 1.4. Вывод треугольника Паскаля в окно Gomrriand-Prompt

Заключение

Уф-ф-ф!! Вот вы и одолели базовые конструкции языка. Раз вы добрались до этого места, значит, умеете уже очень много. Вы можете написать программу на Java, отладить ее, устранив ошибки, и выполнить. Вы способны запрограммировать любой не слишком сложный вычислительный алгоритм, обрабатывающий числовые данные.

Теперь можно перейти к вопросам создания сложных производственных программ. Такие программы требуют тщательного планирования. Сделать это помогает объектно-ориентированное программирование, к которому мы теперь переходим.

Объектно-ориентированное программирование в Java

- **Парадигмы программирования**
  Вся полувековая история программирования компьютеров, а может быть, и история всей науки – это попытка совладать со сложностью окружающего мира. Задачи, встающие перед программистами, становятся все более громоздкими, информация, которую надо обработать, растет как снежный ком.
- **Принципы объектно-ориентированного программирования. Абстракция.**
  Объектно-ориентированное программирование развивается уже более двадцати лет. Имеется несколько школ, каждая из которых предлагает свой набор принципов работы с объектами и по-своему излагает эти принципы. Но есть несколько общепринятых понятий. Перечислим их.
- **Иерархия**
  Иерархия объектов давно используете для их классификации. Особенно детально она проработана в биологии. Все знакомы с семействами, родами и видами. Мы можем сделать описание своих домашних животных (pets): кошек (cats), собак (dogs), коров (cows) и прочих следующим образом: | class Pet{ // Здесь описываем общие свойства всех домашних любимцев | Master person;
- **Ответственность**
  В нашем примере рассматривается только взаимодействие в процессе кормления, описываемое методом eat(). В этом методе животное обращается к хозяину, умоляя его применить метод getFood(). | В англоязычной литературе подобное обращение описывается словом message.
- **Модульность. Принцип KISS.**
  Этот принцип утверждает – каждый класс должен составлять отдельный модуль. Члены класса, к которым не планируется обращение извне, должны быть инкапсулированы. | В языке Java инкапсуляция достигается добавлением модификатора private к описанию члена класса. Например: | private int mouseCatched;
- **Как описать класс и подкласс**
  Итак, описание класса начинается со слова class, после которого записывается имя класса. Соглашения "Code Conventions" рекомендуют начинать имя класса с заглавной буквы. | Перед словом class можно записать модификаторы класса (class modifiers). Это одно из слов public, abstract, final, strictfp.
- **Абстрактные методы и классы**
  При описании класса Pet мы не можем задать в методе voice () никакой полезный алгоритм, поскольку у всех животных совершенно разные голоса. | В таких случаях мы записываем только заголовок метода и ставим после закрывающей список параметров скобки точку с запятой.
- **Окончательные члены и классы**
  Пометив метод модификатором final, можно запретить его переопределение в подклассах. Это удобно в целях безопасности. Вы можете быть уверены, что метод выполняет те действия, которые вы задали. Именно так определены математические функции sin(), cos() и прочие в классе Math.
- **Класс Object**
  Если при описании класса мы не указываем никакое расширение, т. е. не пишем слово extends и имя класса за ним, как при описании класса Pet, то Java считает этот класс расширением класса object, и компилятор дописывает это за нас: | class Pet extends Object{… } | Можно записать это расширение и явно.
- **Конструкторы класса**
  Вы уже обратили внимание на то, что в операции new, определяющей экземпляры класса, повторяется имя класса со скобками. Это похоже на обращение к методу, но что за "метод", имя которого полностью совпадает с именем класса? | Такой "метод" называется конструктором класса (class constructor).
- **Операция new**
  Пора подробнее описать операцию с одним операндом, обозначаемую словом new. Она применяется для выделения памяти массивам и объектам. | В первом случае в качестве операнда указывается тип элементов массива и количество его элементов в квадратных скобках, например: | double a[] = new double[100];
- **Статические члены класса**
  Разные экземпляры одного класса имеют совершенно независимые друг от друга поля-, принимающие разные значения. Изменение поля в одном экземпляре никак не влияет на то же поле в другом экземпляре. В каждом экземпляре для таких полей выделяется своя ячейка памяти.
- **Класс Complex**
  Комплексные числа широко используются не только в математике. Они часто применяются в графических преобразованиях, в построении фракталов, не говоря уже о физике и технических дисциплинах. Но класс, описывающий комплексные числа, почему-то не включен в стандартную библиотеку Java.
- **Метод main()**
  Всякая программа, оформленная как приложение (application), должна содержать метод с именем main. Он может быть один на все приложение или содержаться в некоторых классах этого приложения, а может находиться и в каждом классе.
- **Где видны переменные**
  В языке Java нестатические переменные можно объявлять в любом месте кода между операторами. Статические переменные могут быть только полями класса, а значит, не могут объявляться внутри методов и блоков. Какова же область видимости (scope) переменных?
- **Вложенные классы**
  В этой главе уже несколько раз упоминалось, что в теле класса можно сделать описание другого, вложенного (nested) класса. А во вложенном классе можно снова описать вложенный, внутренний (inner) класс и т. д.

- ## Отношения "быть частью" и "являться"

  Теперь у нас появились две различные иерархии классов. Одну иерархию образует наследование классов, другую – вложенность классов. | Определив, какие классы будут написаны в вашей программе, и сколько их будет, подумайте, как спроектировать взаимодействие классов?

Парадигмы программирования

Вся полувековая история программирования компьютеров, а может быть, и история всей науки – это попытка совладать со сложностью окружающего мира. Задачи, встающие перед программистами, становятся все более громоздкими, информация, которую надо обработать, растет как снежный ком. Еще недавно обычными единицами измерения информации были килобайты и мегабайты, а сейчас уже говорят только о гигабайтах и терабайтах.

Как только программисты предлагают более-менее удовлетворительное решение предложенных задач, тут же возникают новые, еще более сложные задачи. Программисты придумывают новые методы, создают новые языки. За полвека появилось несколько сотен языков, предложено множество методов и стилей. Некоторые методы и стили становятся общепринятыми и образуют на некоторое время так называемую **парадигму программирования**.

Первые, даже самые простые программы, написанные в машинных кодах, составляли сотни строк совершенно непонятного текста. Для упрощения и ускорения программирования придумали языки высокого уровня: FORTRAN, Algol и сотни других, возложив рутинные операции по созданию машинного кода на компилятор. Те же программы, переписанные на языках высокого уровня, стали гораздо понятнее и короче. Но жизнь потребовала решения более сложных задач, и программы снова увеличились в размерах, стали необозримыми.

Возникла идея: оформить программу в виде нескольких, по возможности простых, процедур или функций, каждая из которых решает свой определенную задачу. Написать, откомпилировать и отладить небольшую процедуру можно легко и быстро. Затем остается только собрать все процедуры в нужном порядке в одну программу. Кроме того, один раз написанные процедуры можно затем использовать в других программах как строительные кирпичики. **Процедурное программирование** быстро стало парадигмой. Во все языки высокого уровня включили средства написания процедур и функций. Появилось множество библиотек процедур и функций на все случаи жизни.

Встал вопрос о том, как выявить структуру программы, разбить программу на процедуры, какую часть кода выделить в отдельную процедуру, как сделать алгоритм решения задачи простым и наглядным, как удобнее связать процедуры между собой. Опытные программисты предложили свои рекомендации, названные **структурным программированием**. Структурное программирование оказалось удобным и стало парадигмой. Появились языки программирования, например Pascal, на которых удобно писать структурные программы. Более того, на них очень трудно написать неструктурные программы.

Сложность стоящих перед программистами задач проявилась и тут: программу стали содержать сотни процедур, и опять оказались необозримыми. "Кирпичики" стали слишком маленькими. Потребовался новый стиль программирования.

В это же время обнаружилось, что удачная или неудачная структура исходных данных может сильно облегчить или усложнить их обработку. Одни исходные данные удобнее объединить в массив, для других больше подходит структура дерева или стека. Никлаус Вирт даже назвал свою книгу "Алгоритмы + структуры данных = программы".

Возникла идея объединить исходные данные и все процедуры их обработки в один модуль. Эта идея **модульного программирования** быстро завоевала умы и на некоторое время стала парадигмой. Программы составлялись из отдельных модулей, содержащих десяток-другой процедур и функций. Эффективность таких программ тем выше, чем меньше модули зависят друг от друга. Автономность модулей позволяет создавать и библиотеки модулей, чтобы потом использовать их в качестве строительных блоков для программы.

Для того чтобы обеспечить максимальную независимость модулей друг от друга, надо четко отделить процедуры, которые будут вызываться другими модулями,– **открытые**(public) процедуры, от вспомогательных, которые обрабатывают данные, заключенные в этот

модуль, – **закрытых** (private) процедур. Первые перечисляются в отдельной части модуля – **интерфейсе** (interface), вторые участвуют только в **реализации** (implementation) модуля. Данные, занесенные в модуль, тоже делятся на открытые, указанные в интерфейсе и доступные для других модулей, и закрытые, доступные только для процедур того же модуля.

В разных языках программирования это деление производится по-разному. В языке Turbo Pascal модуль специально делится на интерфейс и реализацию в.языке C интерфейс выносится в отдельные "головные" (**header**) файлы. В языке C++, кроме того, для описания интерфейса можно воспользоваться абстрактными классами. В языке Java есть специальная конструкция для описания интерфейсов, которая так и называется – **interface**, но можно написать и абстрактные классы.

Так возникла идея о скрытии, **инкапсуляции** (incapsulation) данных и методов их обработки. Подобные идеи периодически возникают в дизайне бытовой техники. То телевизоры испещряются кнопками и топорщатся ручками и движками на радость любознательному телезрителю, господствует "приборный" стиль, то вдруг все куда-то пропадает, а на панели остаются только кнопка включения и ручка громкости. Любознательный телезритель берется за отвертку.

Инкапсуляция, конечно, производится не для того, чтобы спрятать от другого модуля что-то любопытное. Здесь преследуются две основные цели. Первая – обеспечить безопасность использования модуля, вынести в интерфейс, сделать общедоступными только те методы обработки информации, которые не могут испортить или удалить исходные данные. Вторая цель – уменьшить сложность, скрыв от внешнего мира ненужные детали реализации.

Опять возник вопрос, каким образом разбить программу на модули? Тут кстати оказались методы решения старой задачи программирования – моделирования действий искусственных и природных объектов: роботов, станков с программным управлением, беспилотных самолетов, людей, животных, растений, систем обеспечения жизнедеятельности, систем управления технологическими процессами.

В самом деле, каждый объект – робот, автомобиль, человек – обладает определенными характеристиками. Ими могут служить: вес, рост, максимальная скорость, угол поворота, грузоподъемность, фамилия, возраст. Объект может производить какие-то действия: перемещаться в пространстве, поворачиваться, поднимать, копать, расти или уменьшаться, есть, пить, рождаться и умирать, изменяя свои первоначальные характеристики. Удобно смоделировать объект в виде модуля. Его характеристики будут данными, постоянными или переменными, а действия – процедурами.

Оказалось удобным сделать и обратное – разбить программу на модули так, чтобы она превратилась в совокупность взаимодействующих объектов. Так возникло **объектно-ориентированное программирование** (object-oriented programming), сокращенно ООП (**ООР**) – современная парадигма программирования.

В виде объектов можно представить совсем неожиданные понятия. Например, окно на экране дисплея – это объект, имеющий ширину width и высоту height, расположение на экране, описываемое обычно координатами (x, y) левого верхнего угла окна, а также шрифт, которым в окно выводится текст, скажем, Times New Roman, цвет фона color, несколько кнопок, линейки прокрутки и другие характеристики. Окно может перемещаться по экрану методом **move()**, увеличиваться или уменьшаться в размерах методом **size()**, сворачиваться в ярлык методом **iconify()**, как-то реагировать на действия мыши и нажатия клавиш. Это полноценный объект! Кнопки, полосы прокрутки и прочие элементы окна – это тоже объекты со своими размерами, шрифтами, перемещениями.

Разумеется, считать, что окно само "умеет" выполнять действия, а мы только даем ему поручения: "Свернись, развернись, передвинься", – это несколько неожиданный взгляд на вещи, но ведь сейчас можно подавать команды не только мышью и клавишами, но и голосом!

Идея объектно-ориентированного программирования оказалась очень плодотворной и стала активно развиваться. Выяснилось, что удобно ставить задачу сразу в виде совокупности действующих объектов – возник **объектно-ориентированный анализ**, ООА.

Решили проектировать сложные системы в виде объектов – **появилось объектно-ориентированное проектирование**, ООП (OOD, **object-oriented design**).

Рассмотрим подробнее принципы объектно-ориентированного программирования.

Принципы объектно-ориентированного программирования. Абстракция.

Объектно-ориентированное программирование развивается уже более двадцати лет. Имеется несколько школ, каждая из которых предлагает свой набор принципов работы с объектами и по-своему излагает эти принципы. Но есть несколько общепринятых понятий. Перечислим их.

Абстракция

Описывая поведение какого-либо объекта, например автомобиля, мы строим его модель. Модель, как правило, не может описать объект полностью, реальные объекты слишком сложны. Приходится отбирать только те характеристики объекта, которые важны для решения поставленной перед нами задачи. Для описания грузоперевозок важной характеристикой будет грузоподъемность автомобиля, а для описания автомобильных гонок она не существенна. Но для моделирования гонок обязательно надо описать метод набора скорости данным автомобилем, а для грузоперевозок это не столь важно.

Мы должны **абстрагироваться** от некоторых конкретных деталей объекта. Очень важно выбрать правильную степень абстракции. Слишком высокая степень даст только приблизительное описание объекта, не позволит правильно моделировать его поведение. Слишком низкая степень абстракции сделает модель очень сложной, перегруженной деталями, и потому непригодной.

Например, можно совершенно точно предсказать погоду на завтра в определенном месте, но расчеты по такой модели продлятся трое суток даже на самом мощном компьютере. Зачем нужна модель, опаздывающая на два дня? Ну а точность модели, используемой синоптиками, мы все знаем сами. Зато расчеты по этой модели занимают всего несколько часов.

Описание каждой модели производится в виде одного или нескольких **классов** (classes). Класс можно считать проектом, слепком, чертежом, по которому затем будут создаваться конкретные объекты. Класс содержит описание переменных и констант, характеризующих объект. Они называются **полями класса** (class fields). Процедуры, описывающие поведение объекта, называются **методами класса** (class methods). Внутри класса можно описать и**вложенные классы** (nested classes) и **вложенные интерфейсы**. Поля, методы и вложенные классы первого уровня являются **членами класса** (class members). Разные школы объектно-ориентированного программирования предлагают разные термины, мы используем терминологию, принятую в технологии Java.

Вот набросок описания автомобиля:

```
class Automobile{
int maxVelocity; // Поле, содержащее наибольшую скорость автомобиля
int speed; // Поле, содержащее текущую скорость автомобиля
int weight; // Поле, содержащее вес автомобиля
// Прочие поля…
void moveTo(int x, int y){ // Метод, моделирующий перемещение
// автомобиля. Параметры x и y – не поля
int a = 1; // Локальная переменная – не поле
// Тело метода. Здесь описывается закон
// перемещения автомобиля в точку (x, y)
}
// Прочие методы…
}
```

Знатокам ***Pascal***

В Java нет вложенных процедур и функций, в теле метода нельзя описать другой метод.

После того как описание класса закончено, можно создавать конкретные объекты,**экземпляры** (instances) описанного класса. Создание экземпляров производится в три этапа, подобно описанию массивов. Сначала объявляются ссылки на объекты: записывается имя класса, и через пробел перечисляются экземпляры класса, точнее, ссылки на них.

```
Automobile Lada2110, fordScorpio, oka;
```

Затем операцией new определяются сами объекты, под них выделяется оперативная память, ссылка получает адрес этого участка в качестве своего значения.

```
lada2110 = new Automobile();
fordScorpio = new Automobile();
oka = new Automobile();
```

На третьем этапе происходит инициализация объектов, задаются начальные значения. Этот этап, как правило, совмещается со вторым, именно для этого в операции new повторяется имя класса со скобками Automobile (). Это так называемый **конструктор**(constructor) класса, но о нем поговорим попозже.

Поскольку имена полей, методов и вложенных классов у всех объектов одинаковы, они заданы в описании класса, их надо уточнять именем ссылки на объект:

```
lada2110.maxVelocity = 150;
fordScorpio.maxVelocity = 180;
oka.maxVelocity = 350; // Почему бы и нет?
oka.moveTo(35, 120);
```

Напомним, что текстовая строка в кавычках понимается в Java как объект класса String. Поэтому можно написать:

```
int strlen = "Это объект класса String".length();
```

Объект "строка" выполняет метод **length()**, один из методов своего класса string, подсчитывающий число символов в строке. В результате получаем значение strlen, равное 24. Подобная странная запись встречается в программах на Java на каждом шагу.

Во многих ситуациях строят несколько моделей с разной степенью детализации. Скажем, для конструирования пальто и шубы нужна менее точная модель контуров человеческого тела и его движений, а для конструирования фрака или вечернего платья – уже гораздо более точная. При этом более точная модель, с меньшей степенью абстракции, будет использовать уже имеющиеся методы менее точной модели.

Не кажется ли вам, что класс Automobile сильно перегружен? Действительно, в мире выпущены миллионы автомобилей разных марок и видов. Что между ними общего, кроме четырех колес? Да и колес может быть больше или меньше. Не лучше ли написать отдельные классы для легковых и грузовых автомобилей, для гоночных автомобилей и вездеходов? Как организовать все это множество классов? На этот вопрос объектно-ориентированное программирование отвечает так: надо организовать иерархию классов.

Иерархия

Иерархия объектов давно используете для их классификации. Особенно детально она проработана в биологии. Все знакомы с семействами, родами и видами. Мы можем сделать описание своих домашних животных (pets): кошек (cats), собак (dogs), коров (cows) и прочих следующим образом:

```
class Pet{ // Здесь описываем общие свойства всех домашних любимцев
Master person; // Хозяин животного
int weight, age, eatTimel]; // Вес, возраст, время кормления
int eat(int food, int drink, int time){ // Процесс кормления
// Начальные действия…
if (time == eatTimefi]) person.getFood(food, drink);
// Метод потребления пищи
}
void voice(); // Звуки, издаваемые животным
// Прочее…
}
```

Затем создаем классы, описывающие более конкретные объекты, связывая их с общим классом:

```
class Cat extends Pet{ // Описываются свойства, присущие только кошкам:
int mouseCatched; // число пойманных мышей
void toMouse(); // процесс ловли мышей
// Прочие свойства
}
class Dog extends Pet{ // Свойства собак:
void preserve(); // охранять
}
```

Заметьте, что мы не повторяем общие свойства, описанные в классе Pet. Они наследуются автоматически. Мы можем определить объект класса Dog и использовать в нем все свойства класса Pet так, как будто они описаны в классе Dog:

```
Dog tuzik = new Dog(), sharik = new Dog();
```

После этого определения можно будет написать:

```
tuzik.age = 3;
int p = sharik.eat (30, 10, 12);
```

А классификацию продолжить так:

```
class Pointer extends Dog{… } // Свойства породы Пойнтер
class Setter extends Dog{… } // Свойства сеттеров
```

Заметьте, что на каждом следующем уровне иерархии в класс добавляются новые свойства, но ни одно свойство не пропадает. Поэтому и употребляется слово extends – "расширяет" и говорят, что класс Dog – **расширение** (extension) класса Pet. С другой стороны, количество объектов при этом уменьшается: собак меньше, чем всех домашних животных. Поэтому часто говорят, что класс Dog – **подкласс** (subclass) класса Pet, а класс Pet – **суперкласс** (superclass) или надкласс класса Dog.

Часто используют генеалогическую терминологию: родительский класс, дочерний класс, класс-потомок, класс-предок, возникают племянники и внуки, вся беспокойная семейка вступает в отношения, достойные мексиканского сериала.

В этой терминологии говорят о **наследовании** (inheritance) классов, в нашем примере класс Dog наследует класс Pet.

Мы еще не определили счастливого владельца нашего домашнего зоопарка. Опишем его в классе Master. Делаем набросок:

```
class Master{ // Хозяин животного
String name; // Фамилия, имя
// Другие сведения
void getFood(int food, int drink); // Кормление
// Прочее
}
```

Хозяин и его домашние животные постоянно соприкасаются в жизни. Их взаимодействие выражается глаголами "гулять", "кормить", "охранять", "чистить", "ласкаться", "проситься" и прочими. Для описания взаимодействия объектов применяется третий принцип объектно-ориентированного программирования – обязанность или ответственность.

Ответственность

В нашем примере рассматривается только взаимодействие в процессе кормления, описываемое методом **eat()**. В этом методе животное обращается к хозяину, умоляя его применить метод **getFood()**.

В англоязычной литературе подобное обращение описывается словом message. Это понятие неудачно переведено на русский язык ни к чему не обязывающим словом "**сообщение**". Лучше было бы использовать слово "послание", "поручение" или даже "распоряжение". Но термин "сообщение" устоялся и нам придется его применять.

Почему же не используется словосочетание "вызов метода", ведь говорят: "Вызов процедуры"? Потому что между этими понятиями есть, по крайней мере, три отличия.

- Сообщение идет к конкретному объекту, знающему метод решения задачи, в примере этот объект – текущее значение переменной person. У каждого объекта свое текущее состояние, свои значения полей класса, и это может повлиять на выполнение метода.
- Способ выполнения поручения, содержащегося в сообщении, зависит от объекта, которому оно послано. Один хозяин поставит миску с "Chappi", другой бросит кость, третий выгонит собаку на улицу. Это интересное свойство называется**полиморфизмом** (polymorphism) и будет обсуждаться ниже.
- Обращение к методу произойдет только на этапе выполнения программы, компилятор ничего не знает про метод. Это называется "**поздним связыванием**" в противовес "**раннему связыванию**", при котором процедура присоединяется к программе на этапе компоновки.

Итак, объект sharik, выполняя свой метод **eat ()**, посылает сообщение объекту, ссылка на который содержится в переменной person, с просьбой выдать ему определенное количество еды и питья. Сообщение записано в строке:

```
person.getFood(food, drink).
```

Этим сообщением заключается **контракт** (contract) между объектами, суть которого в том, что объект sharik берет на себя **ответственность** (responsibility) задать правильные параметры в сообщении, а объект – текущее значение person – возлагает на себя**ответственность** применить метод кормления **getFood()**, каким бы он ни был.

Для того чтобы правильно реализовать принцип ответственности, применяется четвертый принцип объектно-ориентированного программирования – **модульность**(modularity).

Модульность. Принцип KISS.

Этот принцип утверждает – каждый класс должен составлять отдельный модуль. Члены класса, к которым не планируется обращение извне, должны быть инкапсулированы.

В языке Java инкапсуляция достигается добавлением модификатора **private** к описанию члена класса. Например:

```
private int mouseCatched;
private String name;
private void preserve();
```

Эти члены классов становятся **закрытыми**, ими могут пользоваться только экземпляры того же самого класса, например, tuzik может дать поручение:

```
sharik.preserve().
```

А если в классе Master мы напишем:

```
private void getFood(int food, int drink);
```

…то метод **getFood()** не будет найден, и несчастный sharik не сможет получить пищу.

В противоположность закрытости мы можем объявить некоторые члены класса**открытыми**, записав вместо слова private модификатор **public**, например:

```
public void getFood(int food, int drink);
```

К таким членам может обратиться любой объект любого класса.

В языке Java словами private, public и protected отмечается каждый член класса в отдельности.

Принцип модульности предписывает открывать члены класса только в случае необходимости. Вспомните надпись: "Нормальное положение шлагбаума – закрытое".

Если же надо обратиться к полю класса, то рекомендуется включить в класс специальные **методы доступа** (access methods), отдельно для чтения этого поля (**get method**) и для записи в это поле (**set method**). Имена методов доступа рекомендуется начинать со слов get и set, добавляя к этим словам имя поля. Для JavaBeans эти рекомендации возведены в ранг закона.

В нашем примере класса Master методы доступа к полю Name в самом простом виде могут выглядеть так:

```
public String getName(){
return name;
}
public void setName(String newName)
{
name = newName;
}
```

В реальных ситуациях доступ ограничивается разными проверками, особенно в **set-методах**, меняющих значения полей. Можно проверять тип вводимого значения, задавать диапазон значений, сравнивать со списком допустимых значений.

Кроме методов доступа рекомендуется создавать проверочные **is-методы**, возвращающие логическое значение true или false. Например, в класс Master можно включить метод, проверяющий, задано ли имя хозяина:

```
public boolean isEmpty(){
return name == null? true: false;
}
```

…и использовать этот метод для проверки при доступе к полю Name, например:

```
if (master01.isEmpty()) master01.setName("Иванов");
```

Итак, мы оставляем открытыми только методы, необходимые для взаимодействия объектов. При этом удобно спланировать классы так, чтобы зависимость между ними была наименьшей, как принято говорить в теории ООП, было наименьшее **зацепление** (low coupling) между классами. Тогда структура программы сильно упрощается. Кроме того, такие классы удобно использовать как строительные блоки для построения других программ.

Напротив, члены класса должны активно взаимодействовать друг с другом, как говорят, иметь тесную функциональную **связность** (high cohestion). Для этого в класс следует включать все методы, описывающие поведение моделируемого объекта, и только такие методы, ничего лишнего. Одно из правил достижения сильной функциональной связности, введенное Карлом Ли-берхером (**Karl J. Lieberherr**), получило название **закон Деметра**. Закон гласит: "в методе m() класса A следует использовать только методы класса A, методы классов, к которым принадлежат аргументы метода m(), и методы классов, экземпляры которых создаются внутри метода m ().

Объекты, построенные по этим правилам, подобны кораблям, снабженным всем необходимым. Они уходят в автономное плавание, готовые выполнить любое поручение, на которое рассчитана их конструкция.

Будут ли закрытые члены класса доступны его наследникам? Если в классе Pet написано:

```
private Master person;
```

…то можно ли использовать **sharik.person**? Разумеется, нет. Ведь в противном случае каждый, интересующийся закрытыми полями класса A, может расширить его классом B, и просмотреть закрытые поля класса A через экземпляры класса B.

Когда надо разрешить доступ наследникам класса, но нежелательно открывать его всему миру, тогда в Java используется **защищенный** (protected) доступ, отмечаемый модификатором protected, например, объект sharik может обратиться к полю person родительского класса pet, если в классе Pet это поле описано так:

```
protected Master person;
```

Следует сразу сказать, что на доступ к члену класса влияет еще и пакет, в котором находится класс, но об этом поговорим в следующей главе.

Из этого общего схематического описания принципов объектно-ориентированного программирования видно, что язык Java позволяет легко воплощать все эти принципы. Вы уже поняли, как записать класс, его поля и методы, как инкапсулировать члены класса, как сделать расширение класса и какими принципами следует при этом пользоваться. Разберем теперь подробнее правила записи классов и рассмотрим дополнительные их возможности.

Но, говоря о принципах ООП, я не могу удержаться от того, чтобы не напомнить основной принцип всякого программирования.

Принцип KISS

Самый основной, базовый и самый великий принцип программирования – принцип **KISS**– не нуждается в разъяснений и переводе: "Keep It Simple, Stupid!"

Как описать класс и подкласс

Итак, описание класса начинается со слова class, после которого записывается имя класса. Соглашения "Code Conventions" рекомендуют начинать имя класса с заглавной буквы.

Перед словом class можно записать модификаторы класса (**class modifiers**). Это одно из слов **public**, **abstract**, **final**, **strictfp**. Перед именем вложенного класса можно поставить, кроме того, модификаторы **protected**, **private**, **static**. Модификаторы мы будем вводить по мере изучения языка.

Тело класса, в котором в любом порядке перечисляются поля, методы, вложенные классы и интерфейсы, заключается в фигурные скобки.

При описании поля указывается его тип, затем, через пробел, имя и, может быть, начальное значение после знака равенства, которое можно записать константным выражением. Все это уже описано в **главе 1**.

Описание поля может начинаться с одного или нескольких необязательных модификаторов **public**, **protected**, **private**, **static**, **final**, **transient**, **volatile**. Если надо поставить несколько модификаторов, то перечислять их JLS рекомендует в указанном порядке, поскольку некоторые компиляторы требуют определенного порядка записи модификаторов. С модификаторами мы будем знакомиться по мере необходимости.

При описании метода указывается тип возвращаемого им значения или слово void, затем, через пробел, имя метода, потом, в скобках, список параметров. После этого в фигурных скобках расписывается выполняемый метод.

Описание метода может начинаться с модификаторов **public**, **protected**, **private**,**abstract**, **static**, **final**, **synchronized**, **native**, **strictfp**. Мы будем вводить их по необходимости.

В списке параметров через запятую перечисляются тип и имя каждого параметра. Перед типом какого-либо параметра может стоять модификатор **final**. Такой параметр нельзя изменять внутри метода. Список параметров может отсутствовать, но скобки сохраняются.

Перед началом работы метода для каждого параметра выделяется ячейка оперативной памяти, в которую копируется значение параметра, заданное при обращении к методу. Такой способ называется передачей параметров **по значению**.

В листинге 2.1 показано, как можно оформить метод деления пополам для нахождения корня нелинейного уравнения из листинга 1.5.

Листинг 2.1. Нахождение корня нелинейного уравнения методом бисекции.

```
class Bisection2{
private static double final EPS = le-8; // Константа
private double a = 0.0, b = 1.5, root; // Закрытые поля
public double getRoot(){return root;} // Метод доступа
private double f(double x)
{
return x*x*x -3*x*x + 3; // Или что-то другое
}
private void bisect(){ // Параметров нет –
// метод работает с полями экземпляра
double y = 0.0; // Локальная переменная – не поле
do{
root = 0.5 *(a + b); y = f(root);
if (Math.abs(y) < EPS) break;
// Корень найден. Выходим из цикла
// Если на концах отрезка [a; root]
// функция имеет разные знаки:
if (f(a) * y < 0.0} b = root;
// значит, корень здесь
// Переносим точку b в точку root
//В противном случае:
else a = root;
// переносим точку a в точку root
// Продолжаем, пока [a; b] не станет мал
} while(Math.abs(b-a) >= EPS);
}
public static void main(String[] args){
Bisection2 b2 = new Bisection2();
b2.bisect();
System.out.println("x = " +
b2.getRoot() + // Обращаемся к корню через метод доступа
", f() = " +b2.f(b2.getRoot()));
}
}
```

В описании метода **f()** сохранен старый, процедурный стиль: метод получает аргумент, обрабатывает его и возвращает результат. Описание метода bisect о выполнено в духе ООП: метод активен, он сам обращается к полям экземпляра b2 и сам заносит результат в нужное поле. Метод **bisect ()** – это внутренний механизм класса Bisection2, поэтому он закрыт (**private**).

Имя метода, число и типы параметров образуют **сигнатуру** (signature) метода. Компилятор различает методы не по их именам, а по сигнатурам. Это позволяет записывать разные методы с одинаковыми именами, различающиеся числом и/или типами параметров.

Замечание

Тип возвращаемого значения не входит в сигнатуру метода, значит, методы не могут различаться только типом результата их работы.

Например, в классе Automobile мы записали метод **moveTo(int x, int y)**, обозначив пункт назначения его географическими координатами. Можно определить еще метод moveTo (**string destination**) для указания географического названия пункта назначения и обращаться к нему так:

```
oka.moveTo("Москва");
```

Такое дублирование методов называется **перегрузкой** (overloading). Перегрузка методов очень удобна в использовании. Вспомните, в главе 1 мы выводили данные любого типа на экран методом printin() не заботясь о том, данные какого именно типа мы выводим. На самом деле мы использовали разные методы с одним и тем же именем printin, даже не задумываясь об этом. Конечно, все эти методы надо тщательно спланировать и заранее описать в классе. Это и сделано в классе Printstream, где представлено около двадцати методов **print()** и **println()**.

Если же записать метод с тем же именем в подклассе, например:

```
class Truck extends Automobile{
void moveTo(int x, int y){
```

```
// Какие-то действия
}
// Что-то еще
}
```

…то он перекроет метод суперкласса. Определив экземпляр класса Truck, например:

```
Truck gazel = new Truck();
```

…и записав:

```
gazei.moveTo(25, 150),
```

…мы обратимся к методу класса Truck. Произойдет **переопределение** (overriding) метода.

При переопределении права доступа к методу можно только расширить. Открытый метод public должен остаться открытым, защищенный protected может стать открытым.

Можно ли внутри подкласса обратиться к методу суперкласса? Да, можно, если уточнить имя метода, словом super, например:

```
super.moveTo(30, 40).
```

Можно уточнить и имя метода, записанного в этом же классе, словом this, например:

```
this.moveTo (50, 70),
```

…но в данном случае это уже излишне. Таким же образом можно уточнять и совпадающие имена полей, а не только методов.

Данные уточнения подобны тому, как мы говорим про себя "я", а не "Иван Петрович", и говорим "отец", а не "Петр Сидорович".

Переопределение методов приводит к интересным результатам. В классе Pet мы описали метод **voice()**. Переопределим его в подклассах и используем в классе chorus, как показано в листинге 2.2.

Листинг 2.2. Пример полиморфного метода.

```
abstract class Pet{
abstract void voice();
}
class Dog extends Pet{
int k = 10;
void voice(){
System.out.printin("Gav-gav!");
}
}
class Cat extends Pet{
void voice () {
System.out.printin("Miaou!");
}
}
class Cow extends Pet{
void voice(){
System.out.printin("Mu-u-u!");
}
}
public class Chorus(
public static void main(String[] args){
Pet[] singer = new Pet[3];
singer[0] = new Dog();
singer[1] = new Cat();
singer[2] = new Cow();
for (int i = 0; i < singer.length; i++)
singer[i].voice();
}
}
```

На рис. 2.1 показан вывод этой программы. Животные поют своими голосами!

Все дело здесь в определении поля **singer[]**. Хотя массив ссылок singer [] имеет тип Pet, каждый его элемент ссылается на объект своего типа Dog, Cat, cow. При выполнении программы вызывается метод конкретного объекта, а не метод класса, которым определялось имя ссылки. Так в Java реализуется полиморфизм.

Знатокам ***C++***

В языке Java все методы являются виртуальными функциями.

Внимательный читатель заметил в описании класса Pet новое слово abstract. Класс Pet и метод **voice()** являются абстрактными.

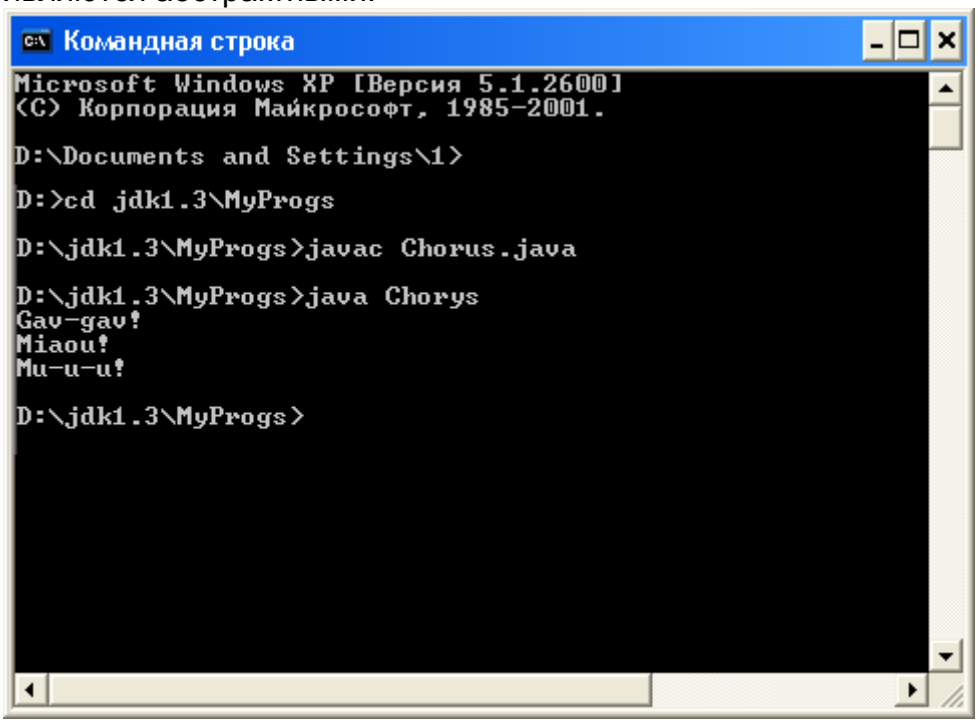

Рис. 2.1. Результат выполнения программы Chorus

Абстрактные методы и классы

При описании класса Pet мы не можем задать в методе **voice ()** никакой полезный алгоритм, поскольку у всех животных совершенно разные голоса.

В таких случаях мы записываем только заголовок метода и ставим после закрывающей список параметров скобки точку с запятой. Этот метод будет **абстрактным** (abstract), что необходимо указать компилятору модификатором **abstract**.

Если класс содержит хоть один абстрактный метод, то создать его экземпляры, а тем более использовать их, не удастся. Такой класс становится **абстрактным**, что обязательно надо указать модификатором **abstract**.

Как же использовать абстрактные классы? Только порождая от них подклассы, в которых переопределены абстрактные методы.

Зачем же нужны абстрактные классы? Не лучше ли сразу написать нужные классы с полностью определенными методами, а не наследовать их от абстрактного класса? Для ответа снова обратимся к листингу 2.2.

Хотя элементы массива singer [] ссылаются на подклассы Dog, Cat, Cow, но все-таки это переменные типа Pet и ссылаться они могут только на поля и методы, описанные в суперклассе Pet. Дополнительные поля подкласса для них недоступны. Попробуйте обратиться, например, к полю k класса Dog, написав singer [0].k. Компилятор "скажет", что он не может реализовать такую ссылку. Поэтому метод, который реализуется в нескольких подклассах, приходится выносить в суперкласс, а если там его нельзя реализовать, то объявить абстрактным. Таким образом, абстрактные классы группируются на вершине иерархии классов.

Кстати, можно задать пустую реализацию метода, просто поставив пару фигурных скобок, ничего не написав между ними, например:

```
void voice(){}
```

Получится полноценный метод. Но это искусственное решение, запутывающее структуру класса.

Замкнуть же иерархию можно окончательными классами.

Окончательные члены и классы

Пометив метод модификатором **final**, можно запретить его переопределение в подклассах. Это удобно в целях безопасности. Вы можете быть уверены, что метод выполняет те действия, которые вы задали. Именно так определены математические функции sin(), cos() и прочие в классе Math. Мы уверены, что метод Math.cos (x) вычисляет именно косинус числа x. Разумеется, такой метод не может быть абстрактным.

Для полной безопасности, поля, обрабатываемые окончательными методами, следует сделать закрытыми (**private**).

Если же пометить модификатором final весь класс, то его вообще нельзя будет расширить. Так определен, например, класс Math:

```
public final class Math{… }
```

Для переменных модификатор final имеет совершенно другой смысл. Если пометить модификатором final описание переменной, то ее значение (а оно должно быть обязательно задано или здесь же, или в блоке инициализации или в конструкторе) нельзя изменить ни в подклассах, ни в самом классе. Переменная превращается в константу. Именно так в языке Java определяются константы:

```
public final int MIN_VALUE = -1, MAX_VALUE = 9999;
```

По соглашению "Code Conventions" константы записываются прописными буквами, слова в них разделяются знаком подчеркивания.

На самой вершине иерархии классов Java стоит класс Object.

Класс Object

Если при описании класса мы не указываем никакое расширение, т. е. не пишем слово extends и имя класса за ним, как при описании класса Pet, то Java считает этот класс расширением класса object, и компилятор дописывает это за нас:

```
class Pet extends Object{… }
```

Можно записать это расширение и явно.

Сам же класс object не является ничьим наследником, от него начинается иерархия любых классов Java. В частности, все массивы – прямые наследники класса object.

Поскольку такой класс может содержать только общие свойства всех классов, в него включено лишь несколько самых общих методов, например, метод **equals()**, сравнивающий данный объект на равенство с объектом, заданным в аргументе, и возвращающий логическое значение. Его можно использовать так:

```
Object objl = new Dog(), obj 2 = new Cat();
if (obj1.equals(obj2))…
```

Оцените объектно-ориентированный дух этой записи: объект obj1 активен, он сам сравнивает себя с другим объектом. Можно, конечно, записать и obj2.equals (obj1), сделав активным объект obj2, с тем же результатом.

Как указывалось в **главе 1**, ссылки можно сравнивать на равенство и неравенство:
```
obj1 == obj2; obj1!= obj 2;
```

В этом случае сопоставляются адреса объектов, мы можем узнать, не указывают ли обе ссылки на один и тот же объект.

Метод **equals()** же сравнивает содержимое объектов в их текущем состоянии, фактически он реализован в классе object как тождество: объект равен только самому себе. Поэтому его часто переопределяют в подклассах, более того, правильно спроектированные, "хорошо воспитанные", классы должны переопределить методы класса object, если их не устраивает стандартная реализация.

Второй метод класса object, который следует переопределять в подклассах, – метод**tostring ()**. Это метод без параметров, который пытается содержимое объекта преобразовать в строку символов и возвращает объект класса string.

К этому методу исполняющая система Java обращается каждый раз, когда требуется представить объект в виде строки, например, в методе **printing**.

Конструкторы класса

Вы уже обратили внимание на то, что в операции **new**, определяющей экземпляры класса, повторяется имя класса со скобками. Это похоже на обращение к методу, но что за "метод", имя которого полностью совпадает с именем класса?

Такой "метод" называется **конструктором класса** (class constructor). Его своеобразие заключается не только в имени. Перечислим особенности конструктора.

- Конструктор имеется в любом классе. Даже если вы его не написали, компилятор Java сам создаст **конструктор по умолчанию** (default constructor), который, впрочем, пуст, он не делает ничего, кроме вызова конструктора суперкласса.
- Конструктор выполняется автоматически при создании экземпляра класса, после распределения памяти и обнуления полей, но до начала использования создаваемого объекта.
- Конструктор не возвращает никакого значения. Поэтому в его описании не пишется даже слово **void**, но можно задать один из трех модификаторов **public**, **protected** или**private**.
- Конструктор не является методом, он даже не считается членом класса. Поэтому его нельзя наследовать или переопределить в подклассе.
- Тело конструктора может начинаться:
    - с вызова одного из конструкторов суперкласса, для этого записывается слово**super()** с параметрами в скобках, если они нужны;
    - с вызова другого конструктора того же класса, для этого записывается слово**this()** с параметрами в скобках, если они нужны.

  Если же **super()** в начале конструктора не указан, то вначале выполняется конструктор суперкласса без аргументов, затем происходит инициализация полей значениями, указанными при их объявлении, а уж потом то, что записано в конструкторе.

Во всем остальном конструктор можно считать обычным методом, в нем разрешается записывать любые операторы, даже оператор **return**, но только пустой, без всякого возвращаемого значения.

В классе может быть несколько конструкторов. Поскольку у них одно и то же имя, совпадающее с именем класса, то они должны отличаться типом и/или количеством параметров.

В наших примерах мы ни разу не рассматривали конструкторы классов, поэтому при создании экземпляров наших классов вызывался конструктор класса **object**.

Операция new

Пора подробнее описать операцию с одним операндом, обозначаемую словом **new**. Она применяется для выделения памяти массивам и объектам.

В первом случае в качестве операнда указывается тип элементов массива и количество его элементов в квадратных скобках, например:

```
double a[] = new double[100];
```

Во втором случае операндом служит конструктор класса. Если конструктора в классе нет, то вызывается конструктор по умолчанию.

Числовые поля класса получают нулевые значения, логические поля – значение **false**, ссылки – значение **null**.

Результатом операции **new** будет ссылка на созданный объект. Эта ссылка может быть присвоена переменной типа ссылка на данный тип:

```
Dog k9 = new Dog ();
```

…но может использоваться и непосредственно:

```
new Dog().voice();
```

Здесь после создания безымянного объекта сразу выполняется его метод **voice()**. Такая странная запись встречается в программах, написанных на Java, на каждом шагу.

Статические члены класса

Разные экземпляры одного класса имеют совершенно независимые друг от друга поля-, принимающие разные значения. Изменение поля в одном экземпляре никак не влияет на то же поле в другом экземпляре. В каждом экземпляре для таких полей выделяется своя ячейка памяти. Поэтому такие поля называются переменными **экземпляра класса** (instance variables) или переменными **объекта**.

Иногда надо определить поле, общее для всего класса, изменение которого в одном экземпляре повлечет изменение того же поля во всех экземплярах. Например, мы хотим в классе Automobile отмечать порядковый заводской номер автомобиля. Такие поля называются **переменными класса** (class variables). Для переменных класса выделяется только одна ячейка памяти, общая для всех экземпляров. Переменные класса образуются в Java модификатором static. В листинге 2.3 мы записываем этот модификатор при определении переменной number.

Листинг 2.3. Статическая переменная.

```
class Automobile {
private static int number;
Automobile(){
number++;
System.out.println("From Automobile constructor:"+
" number = "+number);
}
}
public class AutomobiieTest{
public static void main(String[] args){
Automobile lada2105 = new Automobile(),
fordScorpio = new Automobile(),
oka = new Automobile!);
}
}
```

Получаем результат, показанный на рис. 2.2.

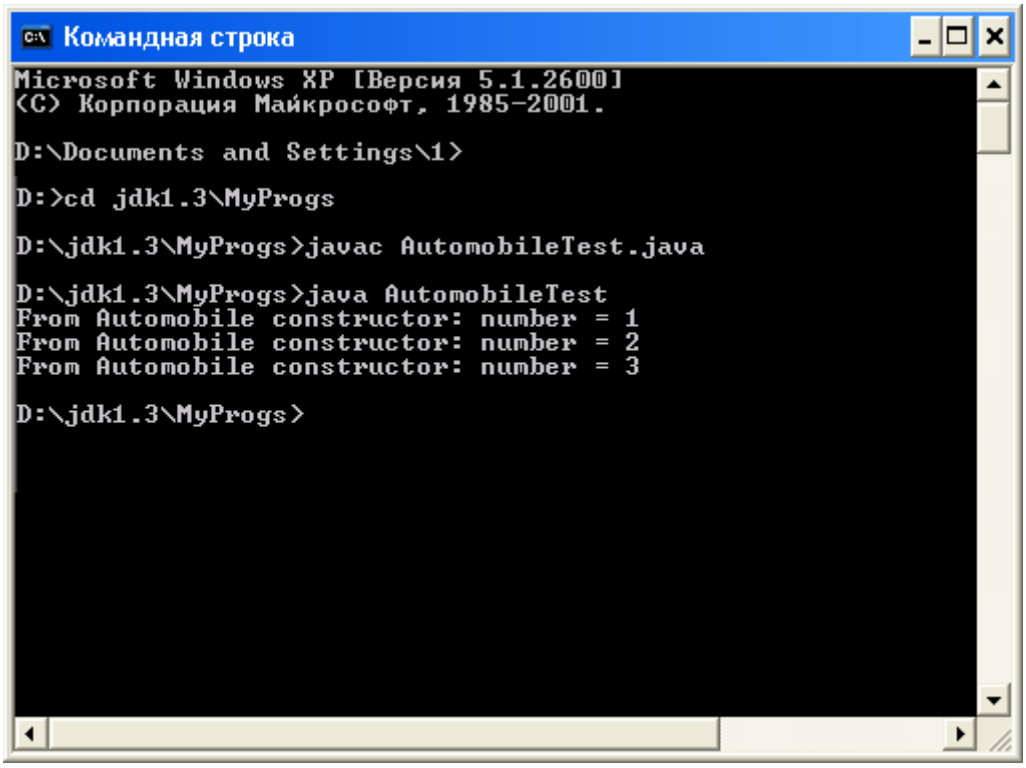

Рис. 2.2. Изменение статической переменной

Интересно, что к статическим переменным можно обращаться с именем класса, Automobile.number, а не только с именем экземпляра, lada2105.number, причем это можно делать, даже если не создан ни один экземпляр класса.

Для работы с такими **статическими переменными** обычно создаются **статические методы**, помеченные модификатором static. Для методов слово static имеет совсем другой смысл. Исполняющая система Java всегда создает в памяти только одну копию машинного кода метода, разделяемую всеми экземплярами, независимо от того, статический это метод или нет.

Основная особенность статических методов – они выполняются сразу во всех экземплярах класса. Более того, они могут выполняться, даже если не создан ни один экземпляр класса. Достаточно уточнить имя метода именем класса (а не именем объекта), чтобы метод мог работать. Именно так мы пользовались методами класса Math, не создавая его экземпляры, а просто записывая Math.abs(x), Math.sqrt(x). Точно так же мы использовали метод System, out .println(). Да и методом main() мы пользуемся, вообще не создавая никаких объектов.

Поэтому статические методы называются **методами класса** (class methods), в отличие от нестатических методов, называемых **методами экземпляра** (instance methods).

Отсюда вытекают другие особенности статических методов:

- в статическом методе нельзя использовать ссылки **this** и **super**;
- в статическом методе нельзя прямо, не создавая экземпляров, ссылаться на нестатические поля и методы;
- статические методы не могут быть абстрактными;
- статические методы переопределяются в подклассах только как статические.

Именно поэтому в листинге 1.5 мы пометили метод **f()** модификатором **static**. Но в листинге 2.1 мы работали с экземпляром b2 класса Bisection2, и нам не потребовалось объявлять метод **f()** статическим.

Статические переменные инициализируются еще до начала работы конструктора, но при инициализации можно использовать только константные выражения. Если же инициализация требует сложных вычислений, например, циклов для задания значений элементам

статических массивов или обращений к методам, то эти вычисления заключают в блок, помеченный словом **static**, который тоже будет выполнен до запуска конструктора:

```
static int[] a = new a[10];
static {
for(int k = 0; k < a.length; k++)
a[k] = k * k;
}
```

Операторы, заключенные в такой блок, выполняются только один раз, при первой загрузке класса, а не при создании каждого экземпляра.

Здесь внимательный читатель, наверное, поймал меня: "А говорил, что все действия выполняются только с помощью методов!" Каюсь: блоки статической инициализации, и блоки инициализации экземпляра записываются вне всяких методов и выполняются до начала выполнения не то что метода, но даже конструктора.

Класс Complex

Комплексные числа широко используются не только в математике. Они часто применяются в графических преобразованиях, в построении фракталов, не говоря уже о физике и технических дисциплинах. Но класс, описывающий комплексные числа, почему-то не включен в стандартную библиотеку Java. Восполним этот пробел.

Листинг 2.4 длинный, но просмотрите его внимательно, при обучении языку программирования очень полезно чтение программ на этом языке. Более того, только программы и стоит читать, пояснения автора лишь мешают вникнуть в смысл действий (шутка).

Листинг 2.4. Класс Complex.

```
class Complex {
private static final double EPS = le-12; // Точность вычислений
private double re, im; // Действительная и мнимая часть
// Четыре конструктора
Complex(double re, double im) {
this, re = re; this.im = im;
}
Complex(double re){this(re, 0.0); }
Complex(){this(0.0, 0.0); }
Complex(Complex z){this(z.re, z.im); }
// Методы доступа
public double getRe(){return re;}
public double getlmf){return im;}
public Complex getZ(){return new Complex(re, im);}
public void setRe(double re){this.re = re;}
public void setlm(double im){this.im = im;}
public void setZ(Complex z){re = z.re; im = z.im;}
// Модуль и аргумент комплексного числа
public double mod(){return Math.sqrt(re * re + im * im);}
public double arg()(return Math.atan2(re, im);}
// Проверка: действительное число?
public boolean isReal(){return Math.abs(im) < EPS;}
public void pr(){ // Вывод на экран
System.out.println(re + (im < 0.0? "": '"+") + im + "i");
}
// Переопределение методов класса Object
public boolean equals(Complex z){
return Math.abs(re – 'z.re) < EPS &&
Math.abs(im – z.im) < EPS;
}
public String toString(){
return "Complex: " + re + " " + im;
}
```

```
// Методы, реализующие операции +=, -=, *=, /=
public void add(Complex z){re += z.re; im += z.im;}
public void sub(Complex z){re -= z.re; im -= z.im;}
public void mul(Complex z){
double t = re * z.re – im * z .im;
im = re * z.im + im * z.re;
re = t;
}
public void div(Complex z){
double m = z.mod();
double t = re * z.re – im * z.im;
im = (im * z.re – re * z.im) / m;
re = t / m;
}
// Методы, реализующие операции +, -, *, /
public Complex plus(Complex z){
return new Complex(re + z.re, im + z im);
}
public Complex minus(Complex z){
return new Complex(re – z.re, im – z.im);
}
public Complex asterisk(Complex z){
return new Complex(
re * z.re – im * z.im, re * z.im + im * z re);
}
public Complex slash(Complex z){
double m = z.mod();
return new Complex(
(re * z.re – im * z.im) / m, (im * z.re – re * z.im) / m);
}
}
// Проверим работу класса Complex
public class ComplexTest{
public static void main(Stringf] args){
Complex z1 = new Complex(),
z2 = new Complex(1.5),
z3 = new Complex(3.6, -2.2),
z4 = new Complex(z3);
System.out.printlnf); // Оставляем пустую строку
System.out.print("z1 = "); z1.pr();
System.out.print("z2 = "); z2.pr();
System.out.print("z3 = "); z3.pr();
System.out.print ("z4 = "}; z4.pr();
System.out.println(z4); // Работает метод toString()
z2.add(z3);
System.out.print("z2 + z3 = "}; z2.pr();
z2.div(z3);
System.out.print("z2 / z3 = "); z2.pr();
z2 = z2.plus(z2);
System.out.print("z2 + z2 = "); z2.pr();
z3 = z2.slash(zl);
System.out.print("z2 / zl = "); z3.pr();
}
}
```

На рис. 2.3 показан вывод этой программы.

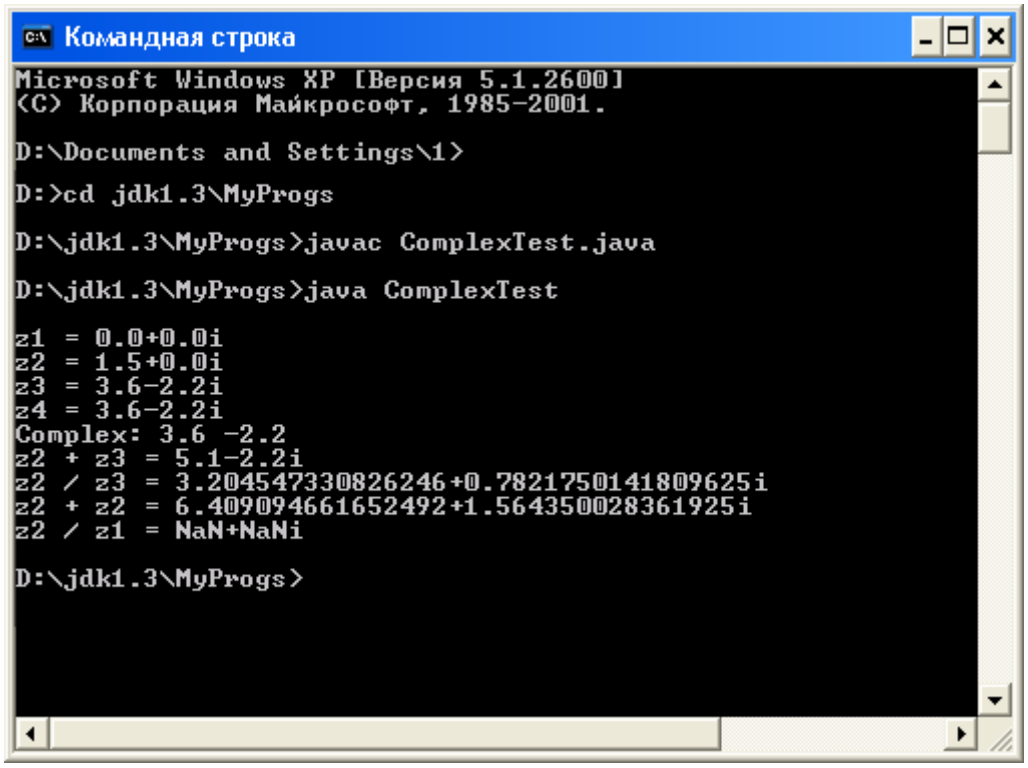

Рис. 2.3. Вывод программы ComplexTest

Метод main()

Всякая программа, оформленная как **приложение** (application), должна содержать метод с именем main. Он может быть один на все приложение или содержаться в некоторых классах этого приложения, а может находиться и в каждом классе.

Метод main() записывается как обычный метод, может содержать любые описания и действия, но он обязательно должен быть открытым (**public**), статическим (**static**), не иметь возвращаемого значения (**void**). Его аргументом обязательно должен быть массив строк (**string[]**). По традиции этот массив называют args, хотя имя может быть любым.

Эти особенности возникают из-за того, что метод **main()** вызывается автоматически исполняющей системой Java в самом начале выполнения приложения. При вызове интерпретатора java указывается класс, где записан метод main(), с которого надо начать выполнение. Поскольку классов с методом main() может быть несколько, можно построить приложение с дополнительными точками входа, начиная выполнение приложения в разных ситуациях из различных классов.

Часто метод main() заносят в каждый класс с целью отладки. В этом случае в метод main() включают тесты для проверки работы всех методов класса.

При вызове интерпретатора java можно передать в метод main() несколько параметров, которые интерпретатор заносит в массив строк. Эти параметры перечисляются в строке вызова java через пробел сразу после имени класса. Если же параметр содержит пробелы, надо заключить его в кавычки. Кавычки не будут включены в параметр, это только ограничители.

Все это легко понять на примере листинга 2.5, в котором записана программа, просто выводящая параметры, передаваемые в метод **main()** при запуске.

Листинг 2.5. Передача параметров в метод main().

```
class Echo {
public static void main(String[] args){
for (int i = 0; i < args.length; i++)
System.out.println("args[" + i +"]="+ args[i]);
}
}
```

На рис. 2.4 показаны результаты работы этой программы с разными вариантами задания параметров.

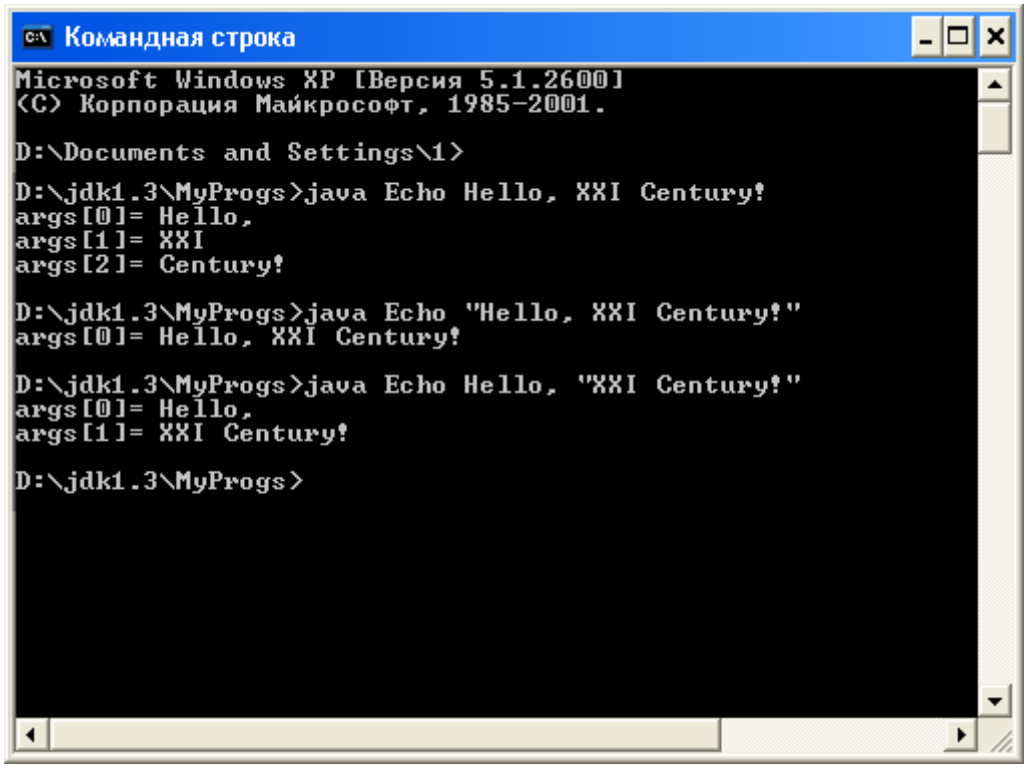

Рис. 2.4. Вывод параметров командной строки

Как видите, имя класса не входит в число параметров. Оно и так известно в методе**main()**.

Знатокам C/C++

Поскольку в Java имя файла всегда совпадает с именем класса, содержащего метод main(), оно не заносится в args[0]. Вместо argc используется args .length. Доступ к переменным среды разрешен не всегда и осуществляется другим способом. Некоторые значения можно просмотреть так:

```
System.getProperties().list(System.out);
```

Где видны переменные

В языке Java нестатические переменные можно объявлять в любом месте кода между операторами. Статические переменные могут быть только полями класса, а значит, не могут объявляться внутри методов и блоков. Какова же **область видимости** (scope) переменных? Из каких методов мы можем обратиться к той или иной переменной? В каких операторах использовать? Рассмотрим на примере листинга 2.6 разные случаи объявления переменных.

Листинг 2.6. Видимость и инициализация переменных.

```
class ManyVariables{
static int x = 9, y; // Статические переменные – поля класса
// Они известны во всех методах и блоках класса
// Переменная y получает значение 0
static{ // Блок инициализации статических переменных
// Выполняется один раз при первой загрузке класса после
// инициализаций в объявлениях переменных
x = 99; // Оператор выполняется вне всякого метода!
}
int a = 1, p; // Нестатические переменные – поля экземпляра
// Известны во всех методах и блоках класса, в которых они
//не перекрыты другими переменными с тем же именем
// Переменная p получает значение 0
{ // Блок инициализации экземпляра
// Выполняется при создании, каждого экземпляра после
// инициализаций при объявлениях переменных
```

```
p = 999; // Оператор выполняется вне всякого метода!
}
static void f(int b){ // Параметр метода b – локальная
// переменная, известна только внутри метода
int a = 2; // Это вторая переменная с тем же именем "a"
// Она известна только внутри метода f() и
// здесь перекрывает первую "a"
int c; // Локальная переменная, известна только в методе f()
//Не получает никакого начального значения
//и должна быть определена перед применением
{ int c = 555; // Ошибка! Попытка повторного объявления
int x = 333; // Локальная переменная, известна только в этом блоке
}
// Здесь переменная x уже неизвестна
for (int d = 0; d < 10; d++){
// Переменная цикла d известна только в цикле
int a = 4; // Ошибка!
int e = 5; // Локальная переменная, известна только в цикле for
e++; // Инициализируется при каждом выполнении цикла
System.out.println("e = " + e); // Выводится всегда "e = 6"
}
// Здесь переменные d и e неизвестны
}
public static void main(String!] args){
int a = 9999; // Локальная переменная, известна
// только внутри метода main()
f (a);
}
}
```

Обратите внимание на то, что переменным класса и экземпляра неявно присваиваются нулевые значения. Символы неявно получают значение '\u0000', логические переменные – значение false, ссылки получают неявно значение null.

Локальные же переменные неявно не инициализируются. Им должны либо явно присваиваться значения, либо они обязаны определяться до первого использования. К счастью, компилятор замечает неопределенные локальные переменные и сообщает о них.

Внимание

Поля класса при объявлении обнуляются, локальные переменные автоматически не инициализируются.

В листинге 2.6 появилась еще одна новая конструкция: **блок инициализации экземпляра** (instance initialization). Это просто блок операторов в фигурных скобках, но записывается он вне всякого метода, прямо в теле класса. Этот блок выполняется при создании каждого экземпляра, после инициализации при объявлении переменных, но до выполнения конструктора. Он играет такую же роль, как и static-блок для статических переменных. Зачем же он нужен, ведь все его содержимое можно написать в начале конструктора? В тех случаях, когда конструктор написать нельзя, а именно, в безымянных внутренних классах.

Вложенные классы

В этой главе уже несколько раз упоминалось, что в теле класса можно сделать описание другого, **вложенного** (nested) класса. А во вложенном классе можно снова описать вложенный, **внутренний** (inner) класс и т. д. Эта матрешка кажется вполне естественной, но вы уже поднаторели в написании классов, и у вас возникает масса вопросов.

- Можем ли мы из вложенного класса обратиться к членам внешнего класса? Можем, для того это все и задумывалось.
- А можем ли мы в таком случае определить экземпляр вложенного класса, не определяя экземпляры внешнего класса? Нет, не можем, сначала надо определить хоть один экземпляр внешнего класса, матрешка ведь!

- А если экземпляров внешнего класса несколько, как узнать, с каким экземпляром внешнего класса работает данный экземпляр вложенного класса? Имя экземпляра вложенного класса уточняется именем связанного с ним экземпляра внешнего класса. Более того, при создании вложенного экземпляра операция **new** тоже уточняется именем внешнего экземпляра.
- А…?

Хватит вопросов, давайте разберем все по порядку.

Все вложенные классы можно разделить на вложенные **классы-члены** класса (member classes), описанные вне методов, и вложенные **локальные классы** (local classes), описанные внутри методов и/или блоков. Локальные классы, как и все локальные переменные, не являются членами класса.

Классы-члены могут быть объявлены статическим модификатором **static**. Поведение статических классов-членов ничем не отличается от поведения обычных классов, отличается только обращение к таким классам. Поэтому они называются **вложенными классами верхнего уровня** (nestee tep-level classes), хотя статические классы-члены можно вкладывать друг в друга. В них можно объявлять статические члены. Используются они обычно для того, чтобы сгруппировать вспомогательные классы вместе с основным классом.

Все нестатические вложенные классы называются **внутренними** (inner). В них нельзя объявлять статические члены.

Локальные классы, как и все локальные переменные, известны только в блоке, в котором они определены. Они могут быть **безымянными** (anonymous classes).

В листинге 2.7 рассмотрены все эти случаи.

Листинг 2.7. Вложенные классы.

```
class Nested{
static private int pr; // Переменная pr обьявлена статической
// чтобы к ней был доступ из статических классов A и AB
String s = "Member of Nested";
// Вкладываем статический класс.
static class.A{ // Полное имя этого класса – Nested.A
private int a=pr;
String s = "Member of A";
// Во вложенньм класс A вкладываем еще один статический класс
static class AB{ // Полное имя класса – Nested.A.AB
private int ab=pr;
String s = "Member of AB";
}
}
//В класс Nested вкладываем нестатический класс
class B{ // Полное имя этого класса – Nested.B
private int b=pr;
String s = "Member of B";
// В класс B вкладываем еще один класс
class BC{ // Полное имя класса – Nested.B.BC
private int bc=pr;
String s = "Member of BC";
}
void f(final int i){ // Без слова final переменные i и j
final int j = 99; // нельзя использовать в локальном классе D
class D{ // Локальный класс D известен только внутри f()
private int d=pr;
String s = "Member of D";
void pr(){
// Обратите внимание на то, как различаются
// переменные с одним и тем же именем "s"
System.out.println(s + (i+j)); // "s" эквивалентно "this.s"
System.out.println(B.this.s);
System.out.println(Nested.this.s);
// System.out.println(AB.this.s); // Нет доступа
// System.out.println(A.this.s); // Нет доступа
}
```

```
}
D d = new D(); // Объект определяется тут же, в методе f()
d.pr(); // Объект известен только в методе f()
}
}
void m(){
new Object(){ // Создается объект безымянного класса,
// указывается конструктор его суперкласса
private int e = pr;
void g(){
System.out.println("From g());
}
}.g(); // Тут же выполняется метод только что созданного объекта
}
}
public class NestedClasses{
public static void main(String[] args){
Nested nest = new Nested(); // Последовательно раскрываются
// три матрешки
Nested.A theA = nest.new A(); // Полное имя класса и уточненная
// операция new. Но конструктор только вложенного класса
Nested.A.AB theAB = theA.new AB(); // Те же правила. Операция
// new уточняется только одним именем
Nested.B theB = nest.new B(); // Еще одна матрешка
Nested.B.BC theBC = theB.new BC();
theB.f(999); // Методы вызываются обычным образом
nest.m();
}
}
```

Ну как? Поняли что-нибудь? Если вы все поняли и готовы применять эти конструкции в своих программах, значит вы – выдающийся талант и можете перейти к следующему пункту. Если вы ничего не поняли, значит вы – нормальный человек. Помните принцип KISS и используйте вложенные классы как можно реже.

Для остальных дадим пояснения.

- Как видите, доступ к полям внешнего класса Nested возможен отовсюду, даже к закрытому полю pr. Именно для этого в Java и введены вложенные классы. Остальные конструкции введены вынужденно, для того чтобы увязать концы с концами.
- Язык Java позволяет использовать одни и те же имена в разных областях видимости – пришлось уточнять константу this именем класса: Nested.this, B.this.
- В безымянном классе не может быть конструктора, ведь имя конструктора должно совпадать с именем класса, – пришлось использовать имя суперкласса, в примере это класс object. Вместо конструктора в безымянном классе используется блок инициализации экземпляра.
- Нельзя создать экземпляр вложенного класса, не создав предварительно экземпляр внешнего класса, – пришлось подстраховать это правило уточнением операции new именем экземпляра внешнего класса– nest.new, theA.new, theB.new.
- При определении экземпляра указывается полное имя вложенного класса, но в операции new записывается просто конструктор класса.

Введение вложенных классов сильно усложнило синтаксис и поставило много задач разработчикам языка. Это еще не все. Дотошный читатель уже зарядил новую обойму вопросов.

- Можно ли наследовать вложенные классы? Можно.
- Как из подкласса обратиться к методу суперкласса? Константа super уточняется именем соответствующего суперкласса, подобно константе this.
- А могут ли вложенные классы быть расширениями других классов? Могут.
- А как? KISS!!!

Механизм вложенных классов станет понятнее, если посмотреть, какие файлы с байт-кодами создал компилятор:

- **Nested1D.class** – локальный класс D, вложенный в класс Nested;
- **NestedS1.class** – безымянный класс;
- **Nested$A$AB.class** – класс Nested.A.AB;
- **Nested$A.class** – класс Nested.A;
- **Nested$B$BC.class** – класс Nested.в.вс;
- **NestedSB.class** – класс Nested.в;
- **Nested.class** – внешний класс Nested;
- **NestedClasses.class** – класс с методом main ().

Компилятор разложил матрешки и, как всегда, создал отдельные файлы для каждого класса. При этом, поскольку в идентификаторах недопустимы точки, компилятор заменил их знаками доллара. Для безымянного класса компилятор придумал имя. Локальный класс компилятор пометил номером.

Оказывается, вложенные классы существуют только на уровне исходного кода. Виртуальная машина Java ничего не знает о вложенных классах. Она работает с обычными внешними классами. Для взаимодействия объектов вложенных классов компилятор вставляет в них специальные закрытые поля. Поэтому в локальных классах можно использовать только константы объемлющего метода, т. е. переменные, помеченные словом **final**. Виртуальная машина просто не догадается передавать изменяющиеся значения переменных в локальный класс. Таким образом не имеет смысла помечать вложенные классы **private**, все равно они выходят на самый внешний уровень.

Все эти вопросы можно не брать в голову. Вложенные классы – это прямое нарушение принципа KISS, и в Java используются только в самом простом виде, главным образом, при обработке событий, возникающих при действиях с мышью и клавиатурой.

В каких же случаях создавать вложенные классы? В теории ООП вопрос о создании вложенных классов решается при рассмотрении отношений "быть частью" и "являться".

Отношения "быть частью" и "являться"

Теперь у нас появились две различные иерархии классов. Одну иерархию образует наследование классов, другую – вложенность классов.

Определив, какие классы будут написаны в вашей программе, и сколько их будет, подумайте, как спроектировать взаимодействие классов? Вырастить пышное генеалогическое дерево классов-наследников или расписать матрешку вложенных классов?

Теория ООП советует прежде всего выяснить, в каком отношении находятся ваши классы P и Q – в отношении "класс Q является экземпляром класса P" ("a class Q is a class p") или в отношении "класс Q – часть класса P" ("a class Q has a class P").

Например: "Собака является животным" или "Собака – часть животного"? Ясно, что верно первое отношение "is-a", поэтому мы и определили класс Dog как расширение класса Pet.

Отношение "is-a" – это отношение "обобщение-детализация", отношение большей или меньшей абстракции, и ему соответствует наследование классов.

Отношение "has-a" – это отношение "целое-часть", ему соответствует вложение.

Заключение

После прочтения этой главы вы получили представление о современной парадигме программирования – объектно-ориентированном программировании и реализации этой парадигмы в языке Java. Если вас заинтересовало ООП, обратитесь к специальной литературе.

Не беда, если вы не усвоили сразу принципы ООП. Для выработки "объектного" взгляда на программирование нужны время и практика. Вторая и третья части книги как раз и дадут

вам эту практику. Но сначала необходимо ознакомиться с важными понятиями языка Java – пакетами и интерфейсами.

Пакеты и интерфейсы

- ## Пакет и подпакет
  В стандартную библиотеку Java API входят сотни классов. Каждый программист в ходе работы добавляет к ним десятки своих. Множество классов становится необозримым. Уже давно принять классы объединять в библиотеки. Но библиотеки классов, кроме стандартной, не являются частью языка.
- ## Права доступа к членам класса
  Пришло время подробно разобрать различные ограничения доступа к полям и методам класса. | Рассмотрим большой пример. Пусть имеется пять классов, размещенных в двух пакетах, как показано на рис. 3.1. | Рис. 3.1.
- ## Размещение пакетов по файлам
  То обстоятельство, что class-файлы, содержащие байт-коды классов, должны быть размещены по соответствующим каталогам, накладывает свои особенности на процесс компиляции и выполнения программы. | Обратимся к тому же примеру.
- ## Импорт классов и пакетов
  Внимательный читатель заметил во второй строке листинга 3.2 новый оператор import. Для чего он нужен? | Дело в том, что компилятор будет искать классы только в – одном пакете, именно, в том, что указан в первой строке файла. Для классов из другого пакета надо указывать полные имена.
- ## Java-файлы
  Теперь можно описать структуру исходного файла с текстом программы на языке Java. | В первой строке файла может быть необязательный оператор package. | В следующих строках могут быть необязательные операторы import. | Далее идут описания классов и интерфейсов. | Еще два правила.
- ## Интерфейсы
  Вы уже заметили, что получить расширение можно только от одного класса, каждый класс в или с происходит из неполной семьи, как показано на рис. 3.4, а. Все классы происходят только от "Адама", от класса object.
- ## Design patterns
  В математике давно выработаны общие методы решения типовых задач. Доказательство теоремы начинается со слов: "Проведем доказательство от противного" или: "Докажем это методом математической индукции", и вы сразу представляете себе схему доказательства, его путь становится вам понятен.

Пакет и подпакет

В стандартную библиотеку Java API входят сотни классов. Каждый программист в ходе работы добавляет к ним десятки своих. Множество классов становится необозримым. Уже давно принять классы объединять в библиотеки. Но библиотеки классов, кроме стандартной, не являются частью языка.

Разработчики Java включили в язык дополнительную конструкцию – **пакеты** (packages). Все классы Java распределяются по пакетам. Кроме классов пакеты могут включать в себя интерфейсы и вложенные **подпакеты** (subpackages). Образуется древовидная структура пакетов и подпакедчэв.

Эта структура в точности отображается на структуру файловой системы. Все файлы с расширением class (содержащие байт-коды), образующие пакет, хранятся в одном каталоге файловой системы. Подпакеты собраны в подкаталоги этого каталога.

Каждый пакет образует одно **пространство имен** (namespace). Это означает, что все имена классов, интерфейсов и подпакетов в пакете должны быть уникальны. Имена в разных пакетах могут совпадать, но это будут разные программные единицы. Таким образом, ни один класс, интерфейс или под-пакет не может оказаться сразу в двух пакетах. Если надо использовать два класса с одинаковыми именами из разных пакетов, то имя класса уточняется именем пакета: пакет.класс. Такое уточненное имя называется **полным именем класса** (fully qualified name).

Все эти правила, опять-таки, совпадают с правилами хранения файлов и подкаталогов в каталогах.

Пакетами пользуются еще и для того, чтобы добавить к уже имеющимся правам доступа к членам класса private, protected и public еще один, "пакетный" уровень доступа.

Если член класса не отмечен ни одним из модификаторов **private**, **protected**, **public**, то, по умолчанию, к нему осуществляется **пакетный доступ** (default access), а именно, к такому члену может обратиться любой метод любого класса из того же пакета. Пакеты ограничивают и доступ к классу целиком – если класс не помечен модификатором public, то все его члены, даже открытые, public, не будут видны из других пакетов.

Как же создать пакет и разместить в нем классы и подпакеты?

Чтобы создать пакет надо просто в первой строке Java-файла с исходным кодом записать строку **package имя;**, например:

```
package mypack;
```

Тем самым создается пакет с указанным именем mypack и все классы, записанные в этом файле, попадут в пакет mypack. Повторяя эту строку в начале каждого исходного файла, включаем в пакет новые классы.

Имя подпакета уточняется именем пакета. Чтобы создать подпакет с именем, например, subpack, следует в первой строке исходного файла написать;

```
package mypack.subpack;
```

…и все классы этого файла и всех файлов с такой же первой строкой попадут в подпакет subpack пакета mypack.

Можно создать и подпакет подпакета, написав что-нибудь вроде:

```
package mypack.subpack.sub;
```

…и т. д. сколько угодно раз.

Поскольку строка **package имя;** только одна и это обязательно первая строка файла, каждый класс попадает только в один пакет или подпакет.

Компилятор Java может сам создать каталог с тем же именем mypack, а в нем подкаталог subpack, и разместить в них class-файлы с байт-кодами.

Полные имена классов A, в будут выглядеть так:

```
mypack.A, mypack.subpack.в.
```

Фирма SUN рекомендует записывать имена пакетов строчными буквами, тогда они не будут совпадать с именами классов, которые, по соглашению, начинаются с прописной. Кроме того, фирма SUN советует использовать в качестве имени пакета или подпакета доменное имя своего сайта, записанное в обратном порядке, например:

```
com.sun.developer
```

До сих пор мы ни разу не создавали пакет. Куда же попадали наши файлы с откомпилированными классами?

Компилятор всегда создает для таких классов **безымянный пакет** (unnamed package), которому соответствует текущий каталог (**current working directory**) файловой системы. Вот поэтому у нас class-файл всегда оказывался в том же каталоге, что и соответствующий Java-файл.

Безымянный пакет служит обычно хранилищем небольших пробных или промежуточных классов. Большие проекты лучше хранить в пакетах. Например, библиотека классов Java 2 API хранится в пакетах **java**, **javax**, **org**, **omg**. Пакет Java содержит только подпакеты **applet**,**awt**, **beans**, **io**, **lang**, **math**, **net**, **rmi**, **security**, **sql**, **text**, **util** и ни одного класса. Эти пакеты имеют свои подпакеты, например, пакет создания ГИП и графики java.awt содержит подпакеты **color**, **datatransfer**, **dnd**, **event**, **font**, **geometry**, **im**, **image**, **print**.

Конечно, состав пакетов меняется от версии к версии.

Права доступа к членам класса

Пришло время подробно разобрать различные ограничения доступа к полям и методам класса.

Рассмотрим большой пример. Пусть имеется пять классов, размещенных в двух пакетах, как показано на рис. 3.1.

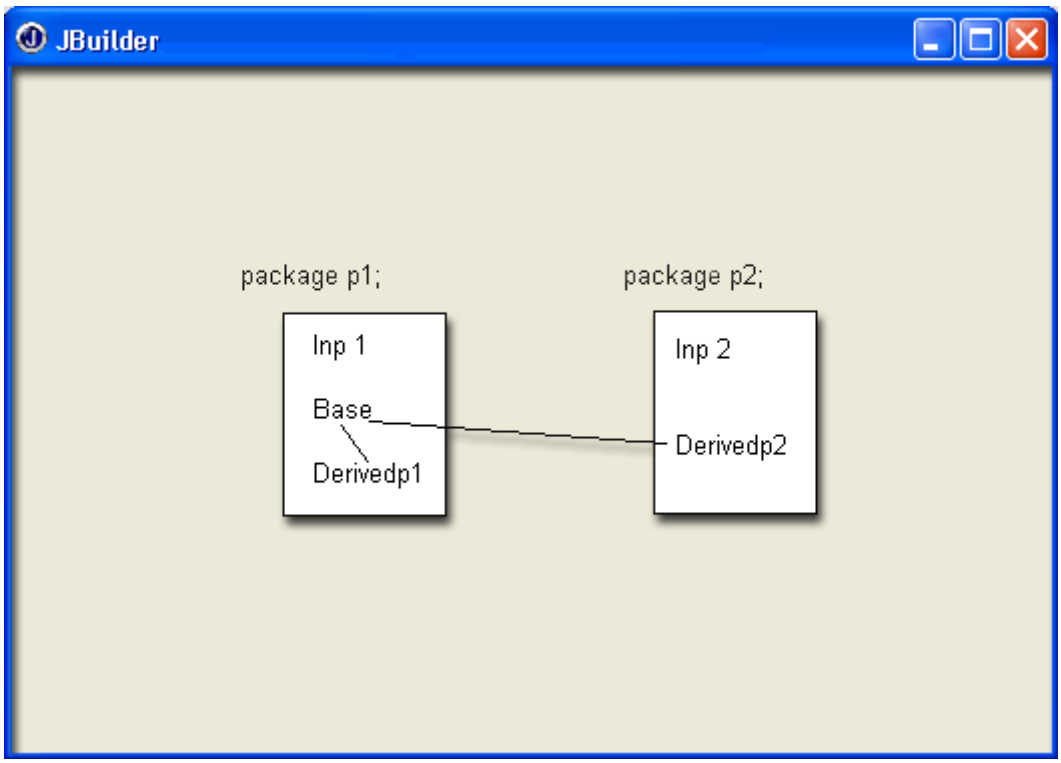

Рис. 3.1. Размещение наших классов по пакетам

В файле Base.java описаны три класса: inpi, Base и класс Derivedpi, расширяющий класс вазе. Эти классы размещены в пакете pi. В классе Base определены переменные всех четырех типов доступа, а в методах f() классов inp1 и Derivedp1 сделана попытка доступа ко всем полям класса вазе. Неудачные попытки отмечены комментариями. В комментариях помещены сообщения компилятора. Листинг 3.1 показывает содержимое этого файла.

Листинг 3.1. Файл Base.java с описанием пакета p1.

```
package p1;
class Inp1{
public void f () {
Base b = new Base();
// b.priv = 1; // "priv has private access in p1.Base"
b.pack = 1;
b.prot = 1;
b.publ = 1;
}
}
public class Base{
private int priv = 0;
int pack = 0;
protected int prot = 0;
public int publ = 0;
}
class Derivedpi extends Base{
public void f(Base a) {
// a.priv = 1; // "priv hds private access in pi.Base"
a.pack = 1;
a.prot = 1;
a.publ = 1;
// priv = 1; // "priv has private access in pi.Base"
pack = 1;
prot = 1;
publ = 1;
}
}
```

Как видно из листинга 3.1, в пакете недоступны только закрытые, **private**, поля другого класса.

В файле Inp2.java описаны два класса: inp2 и класс Derivedp2, расширяющий класс **base**. Эти классы находятся в другом пакете p2. В этих классах тоже сделана попытка обращения к полям класса вазе. Неудачные попытки прокомментированы сообщениями компилятора. Листинг 3.2 показывает содержимое этого файла.

Напомним, что класс вазе должен быть помечен при своем описании в пакете p1 модификатором **public**, иначе из пакета p2 не будет видно ни одного его члена.

Листинг 3.2. Файл Inp2.java с описанием пакета p2.

```
package p2;
import pl.Base;
class Inp2{
public static void main(String[] args){
Base b = new Base();
// b.priv = 1; // "priv has private access in pl.Base"
// b.pack = 1; // "pack is not public in pl.Base;
// cannot be accessed from outside package"
// b.prot = 1; //""prot has protected access in pi.Base"
b.publ = 1;
}
}
class Derivedp2 extends Base{
public void, f (Base a){
// a.priv = 1; // "priv has private access in .p1.Base"
// a.pack = 1; // "pack, is not public in pi.Base; cannot
//be accessed from outside package"
// a.prot = 1; // "prot has protected access in p1.Base"
a.publ = 1;
// priv = 1; // "priv has private access in pi.Base"
// pack = 1; // "pack is not public in pi.Base; cannot
// be accessed from outside package"
prot = 1;
publ = 1;
super.prot = 1;
}
}
```

Здесь, в другом пакете, доступ ограничен в большей степени.

Из независимого класса можно обратиться только к открытым, **public**, полям класса другого пакета. Из подкласса можно обратиться еще и к защищенным, **protected**, полям, но только унаследованным непосредственно, а не через экземпляр суперкласса.

Все указанное относится не только к полям, но и к методам. Подытожим все сказанное в табл. 3.1.

Таблица 3.1. Права доступа к полям и методам класса.

	Класс	Пакет	Пакет и подклассы	Все классы
private	+			
"**package**"	+	+		
protected		+	+	*
public	+	+	+	+

Особенность доступа к protected-полям и методам из чужого пакета отмечена звездочкой.

Размещение пакетов по файлам

То обстоятельство, что class-файлы, содержащие байт-коды классов, должны быть размещены по соответствующим каталогам, накладывает свои особенности на процесс компиляции и выполнения программы.

Обратимся к тому же примеру. Пусть в каталоге D:\jdkl.3\MyProgs\ch3 есть пустой подкаталог classes и два файла – Base.java и Inp2.java, – содержимое которых показано в листингах 3.1 и 3.2. Рис. 3.2 демонстрирует структуру каталогов уже после компиляции.

Мы можем проделать всю работу вручную.

1. В каталоге classes создаем подкаталоги p1 и p2.
2. Переносим файл Base.java в каталог p1 и делаем p1 текущим каталогом.
3. Компилируем Base.java, получая в каталоге p1 три файла: Base.class, Inpl.class, Derivedpl.class.
4. Переносим файл Inp2java в каталог p2.
5. Снова делаем текущим каталог classes.
6. Компилируем второй файл, указывая путь p2\Inp2.java.
7. Запускаем программу java p2.inp2.

Вместо шагов 2 и 3 можно просто создать три class-файла в любом месте, а потом перенести их в каталог p1. В class-файлах не хранится никакая информация о путях к файлам.

Смысл действий 5 и 6 в том, что при компиляции файла Inp2.java компилятор уже должен знать класс p1.Base, а отыскивает он файл с этим классом по пути p1.Base.class, начиная от текущего каталога.

Обратите внимание на то, что в последнем действии 7 надо указывать полное имя класса.

Если использовать ключи (**options**) командной строки компилятора, то можно выполнить всю работу быстрее.

1. Вызываем компилятор с ключом – d путь, указывая параметром путь начальный каталог для пакета:

```
2. javac – d classes Base.java
```

   Компилятор создаст в каталоге classes подкаталог p1 и поместит туда три class-файла.

3. Вызываем компилятор с еще одним ключом – classpath **путь**, указывая параметром путь каталог classes, в котором находится подкаталог с уже откомпилированным пакетом pi:

```
4. javac – classpath classes – d classes Inp2.java
```

   Компилятор, руководствуясь ключом – d, создаст в каталоге classes подкаталог p2 и поместит туда два class-файла, при создании которых он "заглядывал" в каталог pi, руководствуясь ключом – classpath.

5. Делаем текущим каталог classes.
6. Запускаем профамму java p2.inp2.

Для "юниксоидов" все это звучит, как музыка, ну а прочим придется вспомнить MS DOS.

Конечно, если вы используете для работы не компилятор командной строки, а какое-нибудь IDE, то все эти действия будут сделаны без вашего участия.

На рис. 3.2 отображена структура каталогов после компиляции.

На рис. 3.3 показан вывод этих действий в окно **Command Prompt** и содержимое каталогов после компиляции.

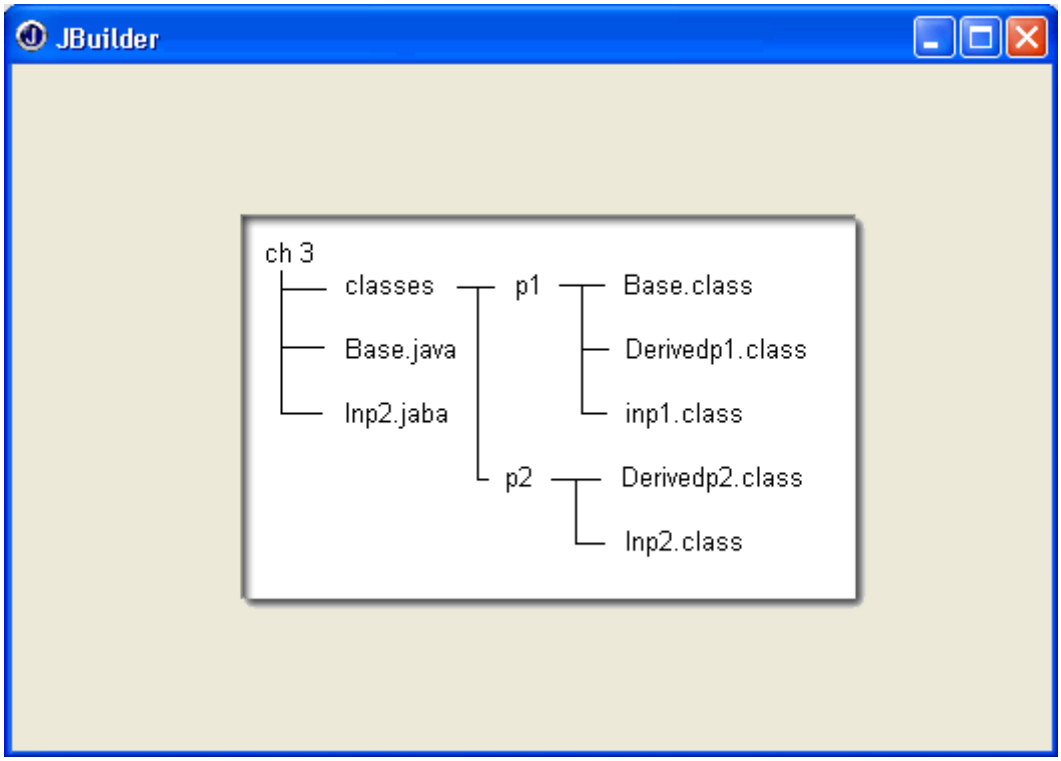

Рис. 3.2. Структура каталогов

```
Командная строка

D:\>cd jdk1.3\MyProgs\ch3

D:\jdk1.3\MyProgs\ch3>javac -d classes Base.java

D:\jdk1.3\MyProgs\ch3>javac -classpath classes -d classes Inp2.java

D:\jdk1.3\MyProgs\ch3>dir /s
 Volume in drive D has no label.
 Volume Serial Number is AC95-B8D2

 Directory of D:\jdk1.3\MyProgs\ch3

18.10.2000  18:05       <DIR>          .
18.10.2000  18:05       <DIR>          ..
18.10.2000  17:44                   604 Base.java
18.10.2000  18:08       <DIR>          classes
18.10.2000  17:48                   911 Inp2.java
               2 File(s)          1 515 bytes

 Directory of D:\jdk1.3\MyProgs\ch3\classes

18.10.2000  18:08       <DIR>          .
18.10.2000  18:08       <DIR>          ..
18.10.2000  18:07       <DIR>          p1
18.10.2000  18:08       <DIR>          p2
               0 File(s)              0 bytes

 Directory of D:\jdk1.3\MyProgs\ch3\classes\p1

18.10.2000  18:07       <DIR>          .
18.10.2000  18:07       <DIR>          ..
18.10.2000  18:11                   329 Base.class
18.10.2000  18:11                   348 Derivedp1.class
18.10.2000  18:11                   340 Inp1.class
               3 File(s)          1 017 bytes

 Directory of D:\jdk1.3\MyProgs\ch3\classes\p2

18.10.2000  18:08       <DIR>          .
18.10.2000  18:08       <DIR>          ..
18.10.2000  18:12                   313 Derivedp2.class
18.10.2000  18:12                   316 Inp2.class
               2 File(s)            629 bytes

     Total Files Listed:
               7 File(s)          3 161 bytes
              11 Dir(s)   3 744 546 816 bytes free

D:\jdk1.3\MyProgs\ch3>cd classes

D:\jdk1.3\Myprogs\ch3\classes>java p2.inp2
```

Рис. 3.3. Протокол компиляции и запуска программы

Импорт классов и пакетов

Внимательный читатель заметил во второй строке листинга 3.2 новый оператор **import**. Для чего он нужен?

Дело в том, что компилятор будет искать классы только в – одном пакете, именно, в том, что указан в первой строке файла. Для классов из другого пакета надо указывать полные имена. В нашем примере они короткие, и мы могли бы писать в листинге 3.2 вместо Base полное имя p1.Base.

Но если полные имена длинные, а используются классы часто, то стучать по клавишам, набирая полные имена, становится утомительно. Вот тут-то мы и пишем операторы import, указывая компилятору полные имена классов.

Правила использования оператора import очень просты: пишется слово import и, через пробел, полное имя класса, завершенное точкой с запятой. Сколько классов надо указать, столько операторов import и пишется.

Это тоже может стать утомительным и тогда используется вторая форма оператора import – указывается имя пакета или подпакета, а вместо короткого имени класса ставится звездочка *. Этой записью компилятору предписывается просмотреть весь пакет. В нашем примере можно было написать:

```
import p1.*;
```

Напомним, что импортировать можно только открытые классы, помеченные модификатором **public**.

Внимательный читатель и тут настороже. Мы ведь пользовались методами классов стандартной библиотеки, не указывая ее пакетов? Да, правильно.

Пакет java.iang просматривается всегда, его необязательно импортировать. Остальные пакеты стандартной библиотеки надо указывать в операторах **import**, либо записывать полные имена классов.

Подчеркнем, что оператор import вводится только для удобства программистов и слово "импортировать" не означает никаких перемещений классов.

Знатокам ***C/C++***

Оператор import не эквивалентен директиве препроцессора include – он не подключает никакие файлы.

Java-файлы

Теперь можно описать структуру исходного файла с текстом программы на языке Java.

- В первой строке файла может быть необязательный оператор **package**.
- В следующих строках могут быть необязательные операторы **import**.
- Далее идут описания классов и интерфейсов.

Еще два правила.

- Среди классов файла может быть только один открытый public -класс.
- Имя файла должно совпадать с именем открытого класса, если последний существует.

Отсюда следует, что, если в проекте есть несколько открытых классов, то они должны находиться в разных файлах.

Соглашение "Code Conventions" рекомендует открытый класс, который, если он имеется в файле, нужно описывать первым.

Интерфейсы

Вы уже заметили, что получить расширение можно только от одного класса, каждый класс в или с происходит из неполной семьи, как показано на рис. 3.4, а. Все классы происходят только от "Адама", от класса **object**. Но часто возникает необходимость породить класс о от двух классов вис, как показано на рис. 3.4, б. Это называется**множественным наследованием** (multiple inheritance). В множественном наследовании нет ничего плохого. Трудности возникают, если классы вис сами порождены от одного класса А, как показано на рис. 3.4* в. Это так называемое "ромбовидное" наследование.

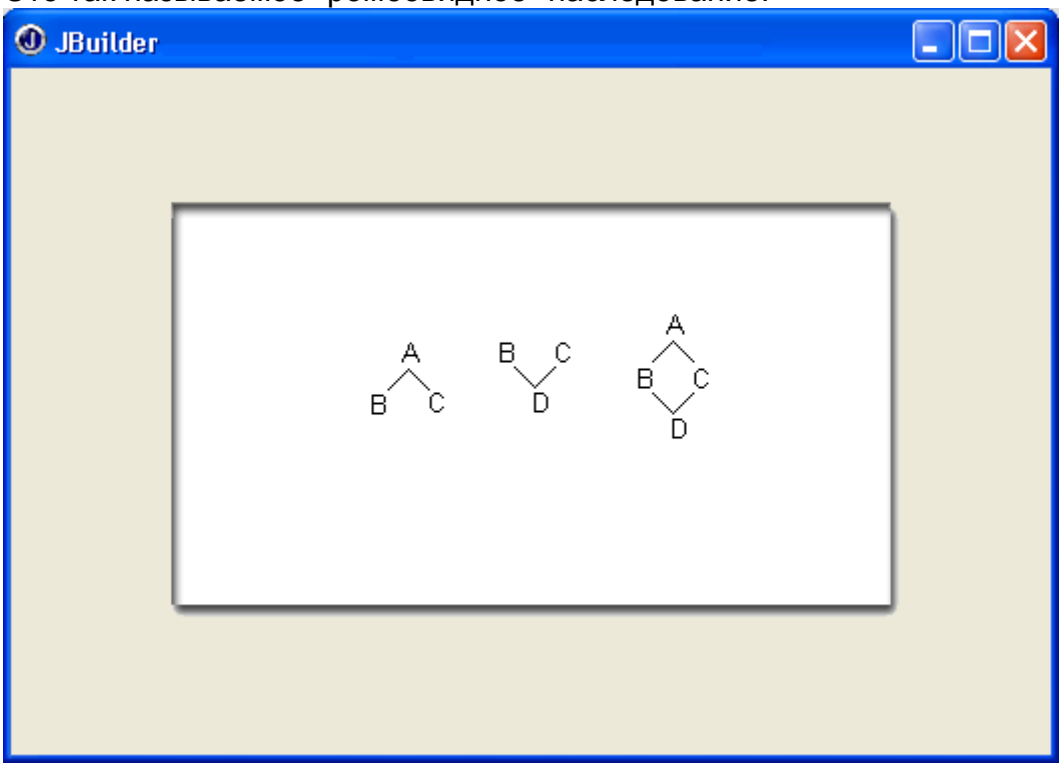

Рис. 3.4. Разные варианты наследования

В самом деле, пусть в классе А определен метод f (), к которому мы обращаемся из некоего метода класса о. Можем мы быть уверены, что метод f о выполняет то, что написано в классе А, т. е. это метод A.f о? Может, он переопределен в классах в и с? Если так, то каким вариантом мы пользуемся: B.f() или c.f()? Конечно, можно определить экземпляры классов и обращаться к методам этих экземпляров, но это совсем другой разговор.

В разных языках программирования этот вопрос решается по-разному, главным образом, уточнением имени метода ft). Но при этом всегда нарушается принцип KISS. Вокруг множественного наследования всегда много споров, есть его ярые приверженцы и столь же ярые противники. Не будем встревать в эти споры, наше дело – наилучшим образом использовать средства языка для решения своих задач.

Создатели языка Java после долгих споров и размышлений поступили радикально – запретили множественное наследование вообще. При расширении класса после слова**extends** можно написать только одно имя суперкласса. С помощью уточнения **super** можно обратиться только к членам непосредственного суперкласса.

Но что делать, если все-таки при порождении надо использовать несколько предков? Например, у нас есть общий класс автомобилей Automobile, от которого можно породить класс грузовиков Truck и класс легковых автомобилей Car. Но вот надо описать пикап Pickup. Этот класс должен наследовать свойства и грузовых, и легковых автомобилей.

В таких случаях используется еще одна конструкция языка Java– интерфейс. Внимательно проанализировав ромбовидное наследование, теоретики ООП выяснили, что проблему создает только реализация методов, а не их описание.

Интерфейс (interface), в отличие от класса, содержит только константы и заголовки методов, без их реализации.

Интерфейсы размещаются в тех же пакетах и подпакетах, что и классы, и компилируются тоже в class-файлы.

Описание интерфейса начинается со слова interface, перед которым может стоять модификатор public, означающий, как и для класса, что интерфейс доступен всюду. Если же модификатора public нет, интерфейс будет виден только в своем пакете.

После слова **interface** записывается имя интерфейса,.потом может;стоять слово **extends**и список интерфейсов-предков через запятую. Таким образом, интерфейсы могут порождаться от интерфейсов, образуя свою, независимую от классов, иерархию, причем в ней допускается множественное наследование интерфейсов. В этой иерархии нет корня, общего предка.

Затем, в фигурных скобках, записываются в любом порядке константы и заголовки методов. Можно сказать, что в интерфейсе все методы абстрактные, но слово **abstract**писать не надо. Константы всегда статические, но слова **static** и **final** указывать не нужно.

Все константы и методы в интерфейсах всегда открыты, не надо даже.указывать модификатор **public**.

Вот какую схему можно предложить для иерархии автомобилей:

```
interface Automobile{… }
interface Car extends Automobile{… }
interface Truck extends Automobile{… }
interface Pickup extends Car, Truck{… }
```

Таким образом, интерфейс – это только набросок, эскиз. В нем указано, что делать, но не указано, как это делать.

Как же использовать интерфейс, если он полностью абстрактен, в нем нет ни одного полного метода?

Использовать нужно не интерфейс, а его **реализацию** (implementation). Реализация интерфейса – это класс, в котором расписываются методы одного или нескольких интерфейсов. В заголовке класса после его имени или после имени его суперкласса, если он есть, записывается слово **implements** и, через запятую, перечисляются имена интерфейсов.

Вот как можно реализовать иерархию автомобилей:

```
interface Automobile{… }
interface Car extends Automobile!… }
class Truck implements Automobile!… }
class Pickup extends Truck implements Car{… }
```

…или так:

```
interface Automobile{… }
interface Car extends Automobile{… }
interface Truck extends Automobile{… }
class Pickup implements Car, Truck{… }
```

Реализация интерфейса может быть неполной, некоторые методы интерфейса расписаны, а другие – нет. Такая реализация – абстрактный класс, его обязательно надо пометить модификатором **abstract**.

Как реализовать в классе pickup метод f(), описанный и в интерфейсе car, и в интерфейсе Truck с одинаковой сигнатурой? Ответ простой – никак. Такую ситуацию нельзя реализовать в классе Pickup. Программу надо спроектировать по-другому.

Итак, интерфейсы позволяют реализовать средствами Java чистое объектно-ориентированное проектирование, не отвлекаясь на вопросы реализации проекта.

Мы можем, приступая к разработке проекта, записать его в виде иерархии интерфейсов, не думая о реализации, а затем построить по этому проекту иерархию классов, учитывая ограничения одиночного наследования и видимости членов классов.

Интересно то, что мы можем создавать ссылки на интерфейсы. Конечно, указывать такая ссылка может только на какую-нибудь реализацию интерфейса. Тем самым мы получаем еще один способ организации полиморфизма.

Листинг 3.3 показывает, как можно собрать с помощью интерфейса хор домашних животных из листинга 2.2.

Листинг 3.3. Использование интерфейса для организации полиморфизма.

```
interface Voice{
void voice();
}
class Dog implements Voice{
public void voice (){
System.out.println("Gav-gav!");
}
}
class Cat implements Voice{
public void voice (){
System.out.println("Miaou!");
}
}
class Cow implements Voice{
public void voice(){
System.out.println("Mu-u-u!");
}
}
public class Chorus{
public static void main(String[] args){
Voiced singer = new Voice[3];
singer[0] = new Dog();
singer[1] = new Cat();
singer[2] = new Cow();
for(int i = 0; i < singer.length; i++)
singer[i].voice();
}
}
```

Здесь используется интерфейс voice вместо абстрактного класса Pet, описанного в листинге 2.2.

Что же лучше использовать: абстрактный класс или интерфейс? На этот вопрос нет однозначного ответа.

Создавая абстрактный класс, вы волей-неволей погружаете его в иерархию классов, связанную условиями одиночного наследования и единым предком – классом **object**. Пользуясь интерфейсами, вы можете свободно проектировать систему, не задумываясь об этих ограничениях.

С другой стороны, в абстрактных классах можно сразу реализовать часть методов. Реализуя же интерфейсы, вы обречены на скучное переопределение всех методов.

Вы, наверное, заметили и еще одно ограничение: все реализации методов интерфейсов должны быть открытыми, **public**, поскольку при переопределении можно лишь расширять доступ, а методы интерфейсов всегда открыты.

Вообще же наличие и классов, и интерфейсов дает разработчику богатые возможности проектирования. В нашем примере, вы можете включить в хор любой класс, просто реализовав в нем интерфейс **voice**.

Наконец, можно использовать интерфейсы просто для определения констант, как показано в листинге 3.4.

Листинг 3.4. Система управления светофором.

```
interface Lights{
int RED = 0;
int YELLOW = 1;
int GREEN = 2;
int ERROR = -1;
}
class Timer implements Lights{
private int delay;
private static int light = RED;
Timer(int sec)(delay = 1000 * sec;}
public int SHIFT(){
int count = (light++) % 3;
try{
switch(count){
case RED: Thread.sleep(delay); break;
case YELLOW: Thread.sleep(delay/3); break;
case GREEN: Thread.sleep(delay/2); break;
}
}catch(Exception e){return ERROR;}
return count;
}
}
class TrafficRegulator{
private static Timer t = new Timer(1);
public static void main(String[] args){
for (int k = -0; k < 10; k++)
switch(t.SHIFT()){
case Lights.RED: System.out.println("Stop!"); break;
case Lights.YELLOW: System.out.println("Wait!"); break;
case Lights.GREEN: System.out.println("Go!"); break;
case Lights.ERROR: System.err.println("Time Error"); break;
default: System.err.println("Unknown light."); return;
}
}
}
```

Здесь, в интерфейсе Lights, определены константы, общие для всего проекта.

Класс Timer реализует этот интерфейс и использует константы напрямую как свои собственные. Метод **SHIFT ()** этого класса подает сигналы переключения светофору с разной задержкой в зависимости от цвета. Задержку осуществляет метод **sleep()** класса Thread из стандартной библиотеки, которому передается время задержки в миллисекундах. Этот метод нуждается в обработке исключений **try{} catch() {}**, о которой мы будем говорить в **главе 16**.

Класс TrafficReguiator не реализует интерфейс Lights и пользуется полными именами Lights.RED и т.д. Это возможно потому, что константы RED, YELLOW и GREEN по умолчанию являются статическими.

Теперь нам известны все средства языка Java, позволяющие проектировать решение поставленной задачи. Заканчивая разговор о проектировании, нельзя не упомянуть о постоянно пополняемой коллекции образцов проектирования (**design patterns**).

Design patterns

В математике давно выработаны общие методы решения типовых задач. Доказательство теоремы начинается со слов: "Проведем доказательство от противного" или: "Докажем это методом математической индукции", и вы сразу представляете себе схему доказательства, его путь становится вам понятен.

Нет ли подобных общих методов в программировании? Есть.

Допустим, вам поручили автоматизировать метеорологическую станцию. Информация от различных датчиков или, другими словами, **контроллеров** температуры, давления, влажности, скорости ветра поступает в цифровом виде в компьютер. Там она обрабатывается: вычисляются усредненные значения по регионам, на основе многодневных наблюдений делается прогноз на завтра, т. е. создается **модель** метеорологической картины местности. Затем прогноз выводится по разным каналам: на экран монитора, самописец, передается по сети. Он представляется в разных **видах**, колонках чисел, графиках, диаграммах.

Естественно спроектировать такую автоматизированную систему из трех частей.

- Первая часть, назовем ее **Контроллером** (controller), принимает сведения от датчиков и преобразует их в какую-то единообразную форму, пригодную для дальнейшей обработки, например, приводит к одному масштабу. При этом для каждого датчика надо написать свой модуль, на вход которого поступают сигналы конкретного устройства, а на выходе образуется унифицированная информация.
- Вторая часть, назовем ее **Моделью** (model), принимает эту унифицированную информацию от Контроллера, ничего не зная о датчике и не интересуясь тем, от какого именно датчика она поступила, и преобразует ее по своим алгоритмам опять-таки к какому-то однообразному виду, например, к последовательности чисел.
- Третья часть системы, **Вид** (view), непосредственно связана с устройствами вывода и преобразует поступившую от Модели последовательность чисел в график, диаграмму или пакет для отправки по сети. Для каждого устройства придется написать свой модуль, учитывающий особенности именно этого устройства.

В чем удобство такой трехзвенной схемы? Она очень гибка. Замена одного датчика приведет к замене только одного модуля в Контроллере, ни Модель, ни Вид этого даже не заметят. Надо представить прогноз в каком-то новом виде, например, для телевидения? Пожалуйста, достаточно написать один модуль и вставить его в Вид. Изменился алгоритм обработки данных? Меняем Модель.

Эта схема разработана еще в 80-х годах прошлого столетия [*То есть XX века. – Ред*.] в языке Smalltalk и получила название MVG (**Model-View-Controller**). Оказалось, что она применима во многих областях, далеких от метеорологии, всюду, где удобно отделить обработку от ввода и вывода информации.

Сбор информации часто организуется так. На экране дисплея открывается поле ввода, в которое вы набиваете сведения, допустим, фамилии в произвольном порядке, а в соседнем поле вывода отображается обработанная информация, например, список фамилий по алфавиту. Будьте уверены, что эта программа организована по схеме МУС. Контроллером служит поле ввода, Видом – поле вывода, а Моделью – метод сортировки фамилий. В третьей части книги мы рассмотрим примеры реализации этой схемы.

К середине 90-х годов накопилось много подобных схем. В них сконцентрирован многолетний опыт тысяч программистов, выражены наилучшие решения типовых задач.

Вот, пожалуй, самая простая из этих схем. Надо написать класс, у которого можно создать только один экземпляр, но этим экземпляром должны пользоваться объекты других классов. Для решения этой задачи предложена схема Singleton, представленная в листинге 3.5.

Листинг 3.5. Схема Singleton.

```
final class Singleton{
private static Singleton s = new Singleton(0);
private int k;
private Singleton(int i){k = i;}
public static Singleton getReference()(return s;}
public int getValue(){return k;}
public void setValue(int i){k = i;}
}
public class SingletonTest {
public static void main(String[] args){
Singleton ref = Singleton.getReference();
System.out.println(ref.getValue());
ref.setValue(ref.getValue() + 5);
System.out.println(ref.getValue());
}
}
```

Класс **singleton** окончательный – его нельзя расширить. Его конструктор закрытый – никакой метод не может создать экземпляр этого класса. Единственный экземпляр s класса**singleton** – статический, он создается внутри класса. Зато любой объект может получить ссылку на экземпляр методом **getReference ()**, Изменить состояние экземпляра s методом**setValue()** или просмотреть его текущее состояние методом **getValue()**.

Это только схема – класс singleton надо еще наполнить полезным содержимым, но идея выражена ясно и полностью.

Схемы проектирования были систематизированы и изложены в книге. Четыре автора этой книги были прозваны "бандой четырех" (**Gang of Four**), а книга, коротко, "GoF". Схемы обработки информации получили название "Design Patterns". Русский термин еще не устоялся. Говорят о "шаблонах", "схемах разработки", "шаблонах проектирования".

В книге GoF описаны 23 шаблона, разбитые на три группы:

1. Шаблоны создания объектов: **Factory**, **Abstract Factory**, **Singleton**, **Builder**, **Prototype**.
2. Шаблоны структуры объектов: **Adapter**, **Bridge**, **Composite**, **Decorator**, **Facade**,**Flyweight**, **Proxy**.
3. Шаблоны поведения объектов: **Chain of Responsibility**, **Command**, **Interpreter**, **Iterator**,**Mediator**, **Memento**, **Observer**, **State**, **Strategy**, **Template**, **Visitor**.

Мы, к сожалению, не можем разобрать подробно **design patterns** в этой книге. Но каждый программист начала XXI века должен их знать. Описание многих разработок начинается словами: "Проект решен на основе шаблона", и структура проекта сразу становится ясна для всякого, знакомого с design patterns.

По ходу книги мы будем указывать, на основе какого шаблона сделана та или иная разработка.

Заключение

Вот мы и закончили первую часть книги. Теперь вы знаете все основные конструкции языка Java, позволяющие спроектировать и реализовать проект любой сложности на основе ООП. Оставшиеся конструкции языка, не менее важные, но реже используемые, отложим до четвертой части. Вторую и третью часть книги посвятим изучению классов и методов, входящих в Core API. Это будет для вас хорошей тренировкой.

Язык Java, как и все современные языки программирования, – это не только синтаксические конструкции, но и богатая библиотека классов. Знание этих классов и умение пользоваться ими как раз и определяет программиста-практика.