Сеттер Р. В.

ИЗУЧАЕМ Java на примерах и задачах

Полный набор сведений о синтаксисе и концепции языка Java

Последовательное изложение

Лучший выбор для студентов и для самостоятельного изучения

НиТ издательство

Наука и Техника

Санкт-Петербург
2016


Java™
ORACLE®

Сеттер Р. В.

ИЗУЧАЕМ JAVA НА ПРИМЕРАХ И ЗАДАЧАХ

Наука и Техника

Санкт-Петербург
2016

Сеттер Р. В.

ИЗУЧАЕМ JAVA НА ПРИМЕРАХ И ЗАДАЧАХ. — СПб.: Наука и Техника, 2016. — 240 с., ил.

Серия «На примерах и задачах»

Эта книга является превосходным базовым учебным пособием для изучения языка программирования Java с нуля. По своей сути Java — популярная современная платформа, позволяющая писать программы, работающие почти на всех мыслимых и немыслимых операционных системах и практически любом оборудовании.

В книге содержатся рецепты и практические указания по решению задач, часто встречающихся при программировании на языке Java. Большинство авторов книг в своих трудах рассматривают теоретические основы языка и уделяют основное внимание базовому синтаксису языка, не рассматривая при этом практическую сторону его применения. Эта же книга старается восполнить недостаток практического материала, содержит множество примеров с комментариями, которые вы сможете использовать в качестве основы своих программных решений, изучения Java.

Материал книги излагается последовательно и сопровождается большим количеством наглядных примеров, разноплановых практических задач и детальным разбором их решений.

Код примеров и дополнительные материалы размещены на сайте www.nit.com.ru.

Контактные телефоны издательства:
(812) 412 70 25, (812) 412 70 26, (044) 516 38 66

Официальный сайт: www.nit.com.ru

Содержание

ГЛАВА 1.

ЧТО ТАКОЕ JAVA?

1.1. История Java

Возникновение языка Java приходится на драматичное время, когда бурно развивался интернет и началась так называемая «война браузеров». Первоначально он предназначался для программирования бытовой техники, например холодильников, микроволновых печей, стиральных машин, электронных записных книжек, видеомагнитофонов и т. п.

Программное обеспечение для бытовой электроники должно удовлетворять определенным требованиям. Прежде всего оно должно быть совместимо с новыми микросхемами, на которых будут строиться последние образцы бытовой техники сразу же после их выпуска. При этом необходимо учитывать, что изготовители очень часто меняют микросхемы: добавляют новые функции, тем самым расширяя спектр устройств, в которых они могут применяться. Программное обеспечение также должно отвечать высоким требованиям надежности, так как поломка бытовой техники обычно связана с большими материальными затратами со стороны производителя. Все это — «отправные точки», с учетом которых шла разработка Java.

Язык Java своим возникновением обязан Sun Microsystems Inc., в дочерней компании которой в начале 90-х гг. началась его разработка в рамках проекта с кодовым названием Green. Фирма Sun известна на рынке аппаратного и программного обеспечения такими продуктами, как процессоры на RISC-архитектуре и собственная операционная система SOLARIS из семейства UNIX.

Над разработкой программного обеспечения для бытовой техники трудилась группа под руководством Джеймса Гослинга, нынешнего вице-президента Sun. Разработчики быстро убедились, что существовавшие в то время языки программирования, в том числе широко распространенный C/C++, для этой цели не годятся, потому что написанные на них программы требуется перекомпилировать для каждого нового процессора. Кроме того, эти языки настолько сложны, что не позволяют быстро писать надежные программы. Поэтому программисты приняли решение для начала создать новый язык программирования — небольшой, надежный и, главное, не зависящий от платформы. Этот язык назвали Oak, и возможности его были впервые продемонстрированы на наладонном компьютере Star 7, управлявшем интерактивным телевизионным приемником. Однако внедрить язык Oak на прибыльные рынки интерактивного телевидения и мобильных телефонов тогда фирме Sun не удалось.

В 1993 году большое влияние на развитие нового языка оказало массовое распространение интернета. Специалисты Sun быстро осознали, что открылось обширнейшее поле для применения программного обеспечения, ра-

ботающего на компьютере любого типа, подключенном к всемирной сети. Впоследствии эта концепция программирования стала известна под лозунгом Сеть — это компьютер».

Появление новой области применения ускорило разработку нового языка, о выпуске первой версии которого вместе с соответствующими библиотеками API было объявлено в 1995 году. Язык пришлось переименовать, так как слово «oak» (англ. «дуб») невозможно было зарегистрировать в качестве торговой марки. По легенде, новое название — Java — обязано своим происхождением многочисленным чашкам кофе, выпитого Джеймсом Гослингом и его коллегами во время работы.

Первым приложением, целиком написанным на Java, стал веб-браузер HotRunner, переименованный позже в HotJava. Новый браузер, в отличие от изделий Netscape и Microsoft, участвовавших в «войне браузеров», был универсальным. Одновременно Sun начала предоставлять лицензии на использование технологии Java (Java 1.1) другим производителям компьютеров и операционных систем: IBM, SGI, Novell, Hewlett Packard (HP), Microsoft.

Производитель обязан был придерживаться спецификации, а за это он получал право украшать свою продукцию знаменитым логотипом с изображением чашки горячего кофе. Всеобщее соблюдение технологии Java привело к реализации маркетингового лозунга, который прославил Java на весь мир: Write once, run anywhere, то есть «Написано однажды, работает везде».

В января 2010 г. корпорация Sun со всеми технологиями (включая язык Java) вошла в состав корпорации Oracle.

1.2. Технологии и версии Java

Внутри Java существуют несколько основных семейств технологий[1]:

- **Java SE** — Java Standard Edition, основное издание Java, содержит компиляторы, API, Java Runtime Environment; подходит для создания пользовательских приложений, в первую очередь — для настольных систем.
- **Java EE** — Java Enterprise Edition, представляет собой набор спецификаций для создания программного обеспечения уровня предприятия.
- **Java ME** — Java Micro Edition, создана для использования в устройствах, ограниченных по вычислительной мощности, например, в мобильных телефонах, КПК, встроенных системах;

- **JavaFX** — технология, являющаяся следующим шагом в эволюции Java как Rich Client Platform; предназначена для создания графических интерфейсов корпоративных приложений и бизнеса.
- **Java Card** — технология предоставляет безопасную среду для приложений, работающих на смарт-картах и других устройствах с очень ограниченным объёмом памяти и возможностями обработки.

Последней версией является Java 8, релиз которой состоялся 19 марта 2014 года.

Список нововведений:

- Полноценная поддержка лямбда-выражений.
- Ключевое слово default в интерфейсах для поддержки функциональности по умолчанию.
- Ссылки на методы.
- Функциональные интерфейсы (предикаты, поставщики и т. д.)
- Потоки (stream) для работы с коллекциями
- Новое API для работы с датами

1.3. Отличительные особенности Java

Java — это простой, объектно-ориентированный, сетевой, интерпретируемый, надежный, безопасный, независимый от архитектуры, переносимый, высокопроизводительный, многопоточный и динамический язык.

Основные возможности [1]:

- автоматическое управление памятью;
- расширенные возможности обработки исключительных ситуаций;
- богатый набор средств фильтрации ввода-вывода;
- набор стандартных коллекций: массив, список, стек и т. п.;
- наличие простых средств создания сетевых приложений (в том числе с использованием протокола RMI);
- наличие классов, позволяющих выполнять HTTP-запросы и обрабатывать ответы;
- встроенные в язык средства создания многопоточных приложений, которые потом были портированы на многие языки (например, python);

- унифицированный доступ к базам данных:
- на уровне отдельных SQL-запросов — на основе JDBC, SQLJ;
- на уровне концепции объектов, обладающих способностью к хранению в базе данных — на основе Java Data Objects (англ.) и Java Persistence API;
- поддержка обобщений (начиная с версии 1.5);
- поддержка лямбд, замыканий, встроенные возможности функционального программирования (с 1.8);
- параллельное выполнение программ

На первый взгляд это не определение, а неумеренные похвалы. Однако каждое из перечисленных прилагательных точно описывает одно из ключевых свойств языка Java. Рассмотрим их значение подробнее.

Простота

Одной из целей, которые ставились при разработке Java, было создание такого языка программирования, который можно изучить легко и быстро.

Поэтому получившийся язык содержит минимум языковых конструкций, которые к тому же восходят к другим широко известным языкам, что облегчает программистам переход на Java.

Из языка исключены элементы, которые портят стиль программирования (например, команда безусловного перехода goto) или усложняют код (заголовочные файлы и препроцессор). Java не поддерживает структур, объединений (union), перегрузки операторов и множественного наследования, а также такого опаснейшего средства C++, как прямая адресация памяти через указатели.

Таким образом, программисту остается намного меньше возможностей внести в код ошибку, чем если бы он писал на традиционных C/C++ или Паскале.

Объектная ориентированность

Java с самого начала разрабатывался как объектно-ориентированный язык. Программист на таком языке может сосредоточиться на конкретных данных своей задачи, представив их как взаимодействующие объекты и определив классы, описывающие состояние и поведение этих объектов. Классы организованы иерархически от наиболее общих до самых специализированных. Пакеты классов для решения стандартных задач поставляются

вместе с языком Java, и разработчик может использовать эти классы в своих приложениях

Поддержка сетевых приложений

Чтобы лозунг сеть — это компьютер» соответствовал действительности, язык Java должен позволять проектировать приложения, работающие в сети, точно так же, как если бы они предназначались для работы на изолированном компьютере. Это требование создателями Java выполнено: язык поддерживает различные уровни сетевого подключения, а открывать файл на удаленном компьютере средствами Java не труднее, чем манипулировать с набором данных, хранящимся локально. Программист на Java имеет в распоряжении также механизм сокетов.

Интерпретируемость

Исходные коды (тексты программ) Java переводятся компилятором не в машинный код, а в так называемый байт-код. Инструкции в байт-коде выполняет виртуальная машина Java. Так достигается переносимость программы на Java: один и тот же байт-код одинаково работает на любой платформе, где установлена нужная версия виртуальной машины.

Интерпретация программы обычно происходит значительно медленнее, чем выполнение машинного кода, но в случае Java благодаря передовым технологиям разница в скорости не слишком велика.

Надежность

Первоначальным назначением языка Java была помощь в написании программ для бытовой техники, поэтому требование надежности программ предъявлялось к нему с самого начала. Java делает невозможными множество типичных ошибок программирования. Так, Java — строго типизированный язык, то есть тип любой переменой и любого выражения должен быть известен на момент компиляции. Как следствие, не допускается использование необъявленных переменных, отсутствует также автоматическое приведение типов. Все ошибки такого рода выявляются уже на стадии трансляции исходного текста в байт-код.

Java поддерживает механизм исключений, в результате чего код программы становится более удобопонятным, чем на других языках.

Преимуществом языка Java является и наличие администратора памяти, или «сборщика мусора» (garbage collector), который автоматически объединяет свободные области памяти, удаляет неиспользуемые объекты, предотвращая появление «дыр» в памяти.

Безопасность

Язык Java содержит в себе механизмы обеспечения безопасности, которые защищают от вредоносного кода (вирусов или червей), представляющего опасность для файловой системы. Все механизмы безопасности основаны на предположении, что верить нельзя никому и ничему. В Java не поддерживается работа с указателями: программист не может зайти «за кулисы» и перенаправить указатели на другую область памяти. Компилятор не принимает решений о размещении данных в памяти, а программист не может при объявлении класса указать место класса в памяти. Важное отличие Java от других языков — верификация байт-кода, полученного из сети.

Любая программа может навредить, но разработчиками языка Java сделано все для того, чтобы это предотвратить.

Независимость от архитектуры

Программа на языке Java не компилируется прямо в машинный код процессора, а переводится в байт-код. Главным преимуществом такого способа является возможность запуска приложения в разных операционных системах и на разных аппаратных платформах без необходимости перекомпиляции для каждой новой платформы. Поэтому с помощью Java легко удовлетворить требованиям написания приложений, которые будут работать совершенно одинаково под MS Windows, Apple Mac, Linux и клонах UNIX.

Переносимость

Архитектурная независимость и переносимость — это не одно и то же, несмотря на то, что они тесно связаны друг с другом. Когда речь идет о независимости от операционной системы, подразумевают, что код программы должен быть одинаковым для любой операционной системы, а когда говорят о ее переносимости, имеют в виду возможность выполнения программы на любой платформе. Порой соображения переносимости накладывают существенные ограничения на возможности языка. Так, спецификация языка Java точно указывает размер в байтах и поведение при арифметических операциях для основных (простых) типов данных.

Пределы переносимости определяются стандартом POSIX (Portable Operating System Interface), обязывающим всех производителей POSIX-совместимых систем поддерживать определенный интерфейс взаимодействия приложения с операционной системой.

Производительность

Java — интерпретируемый язык. Несмотря на это, скорость выполнения программ на языке Java может быть выше, чем у обычных программ, потому что многие современные программы большую часть времени ждут действий пользователя или поступления данных из базы либо из сети.

Если же для конкретной программы соображения скорости выполнения приоритетны перед соображениями переносимости, ее можно преобразовать не в байт-код, а непосредственно в машинные коды конкретного процессора. Существует инструмент, позволяющий сделать это не для всего исходного кода программы, а для отдельных методов, которые компилируются в машинные коды по мере надобности: так называемый компилятор Just In Time (точно вовремя).

Разумеется выигрывая в производительности вы теряете в переносимости и надежности программы.

Многопоточность

На сегодняшний день создание многопоточных приложений уже стало более чем нормой. Пользователь такого приложения может одновременно, например, скачивать файл из сети и прослушивать музыку.

Java поддерживает написание многопоточных программ, включая возможность запретить выполнение одного и того же задания двумя потоками одновременно.

Динамичность

Под динамичностью понимается подгрузка классов в память (в том числе из сети) по мере надобности во время выполнения программы. Это сокращает объем используемой оперативной памяти.

Для каждого программного объекта во время выполнения есть возможность узнать, к какому классу он принадлежит.

ГЛАВА 2.

ПЕРВЫЕ ПРОГРАММЫ. ВВЕДЕНИЕ В СИНТАКСИС ЯЗЫКА

2.1. Установка JDK и NetBeans

Прежде, чем написать и скомпилировать свою первую программу, вам необходимо установить среду разработки Java. Еще в предисловии было упомянуто о том, что примеры, приведенные в книге, компилируются при помощи стандартного программного комплекса JDK (Java Development Kit), бесплатно распространяемого фирмой Oracle (рис. 2.1).

Вместе с JDK рекоемндуем установить интегрированную среду разрабоки, с помощьюкоторойвыбудетесоздаватьивкоторыхвыбудетеписатьпрограммы. Интегрированные среды разработки автоматизируют множество рутинных действий программиста. Вместе с языком Java на сайте Oracle можно скачать бесплатную среду разработки NetBeans. Ее мы рекомендуем для начала и именно на ней мы будем тренироваться в рамках данной книги.

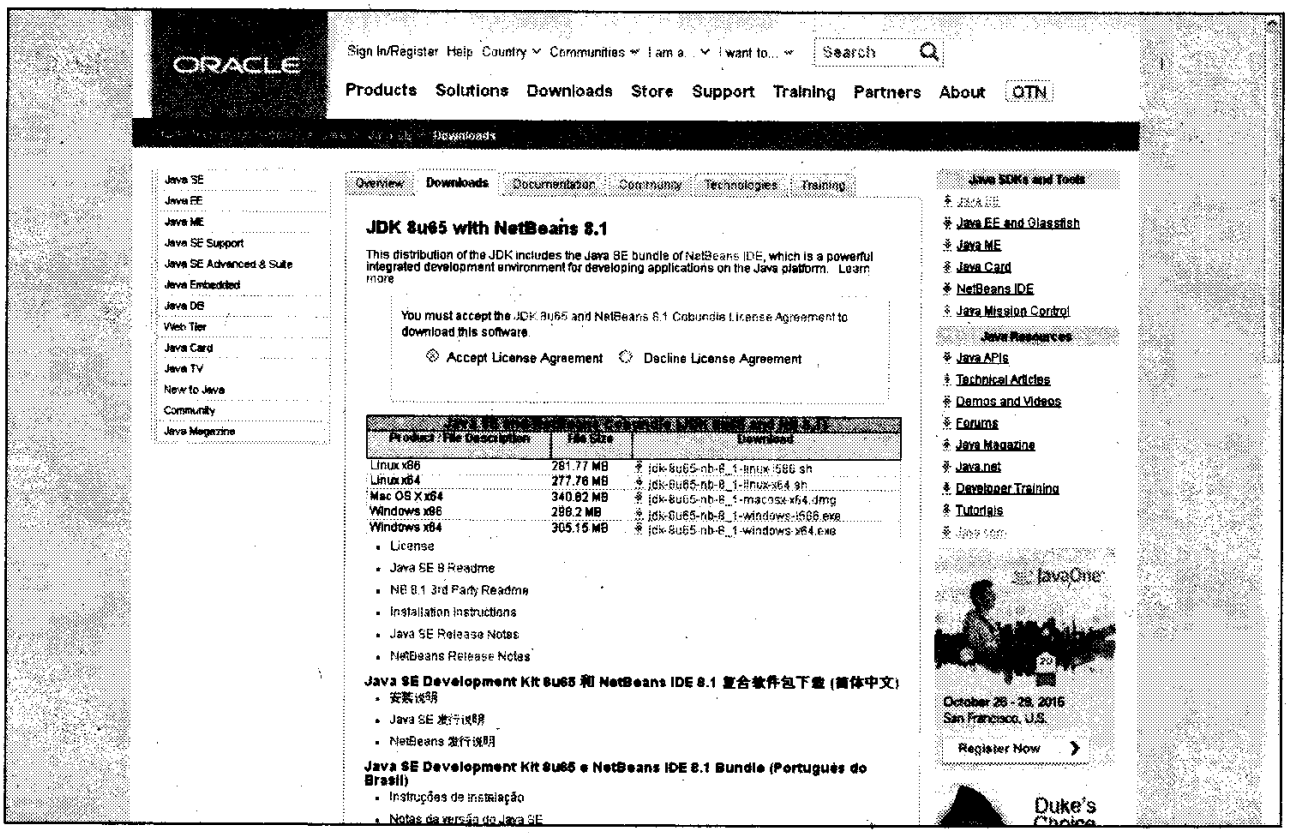


Рис. 2.1. Скачиваем JDK и NetBeans с сайта Oracle.com

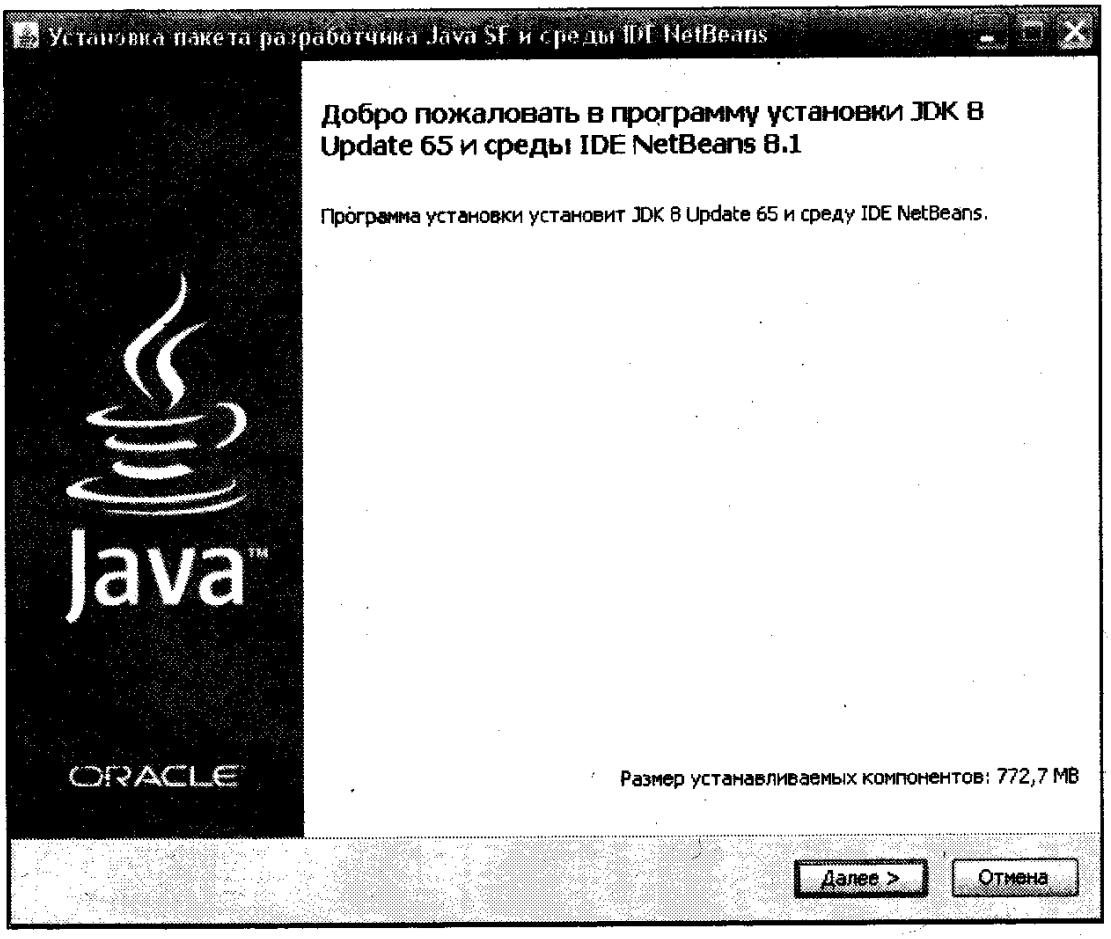


Рис. 2.2. Установка JDK и NetBeans (шаг 1)

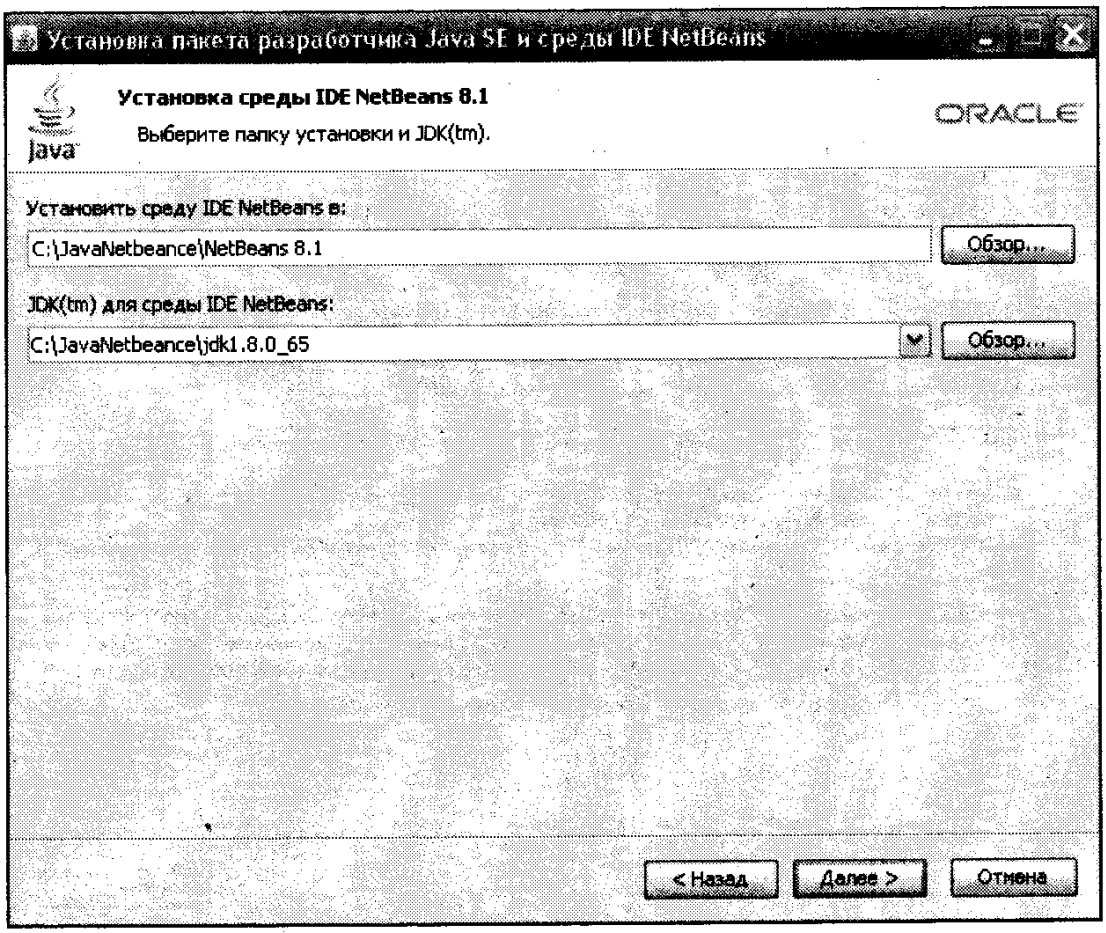


Рис. 2.3. Установка JDK и NetBeans (шаг 2)

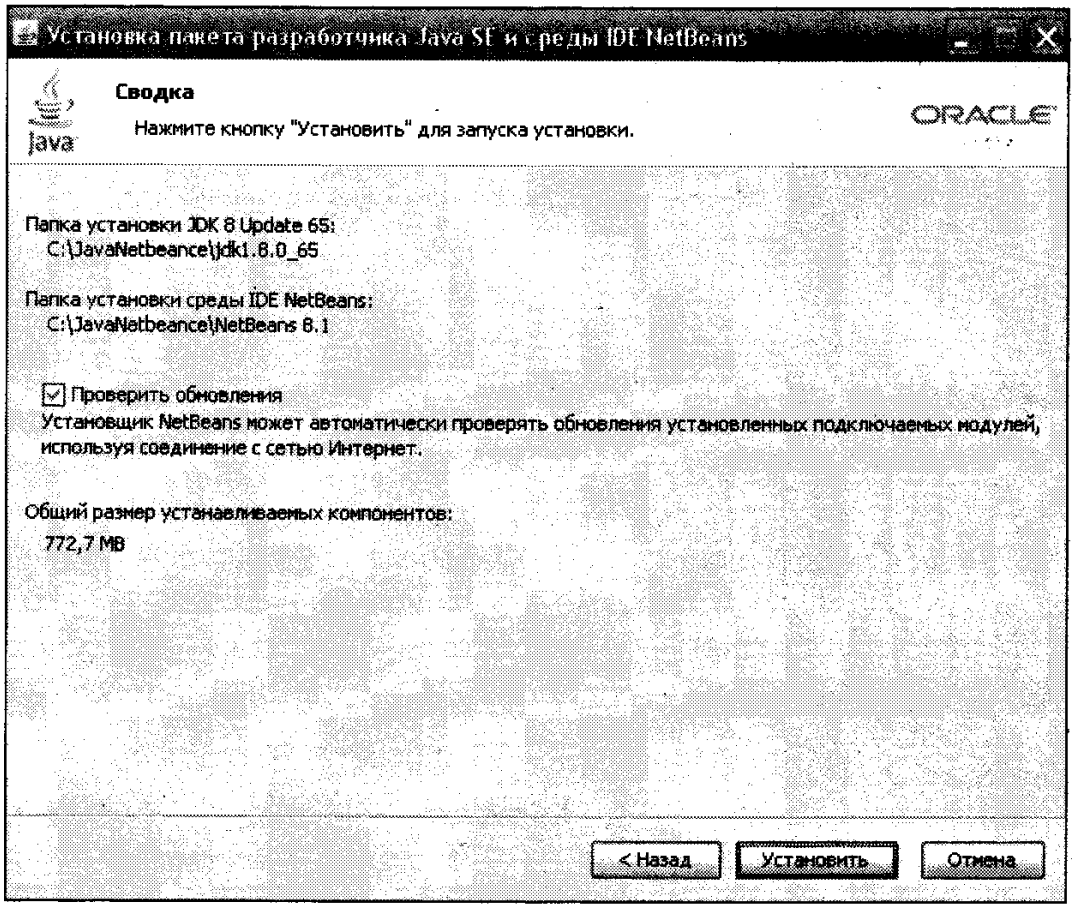


Рис. 2.4. Установка JDK и NetBeans (шаг 3)

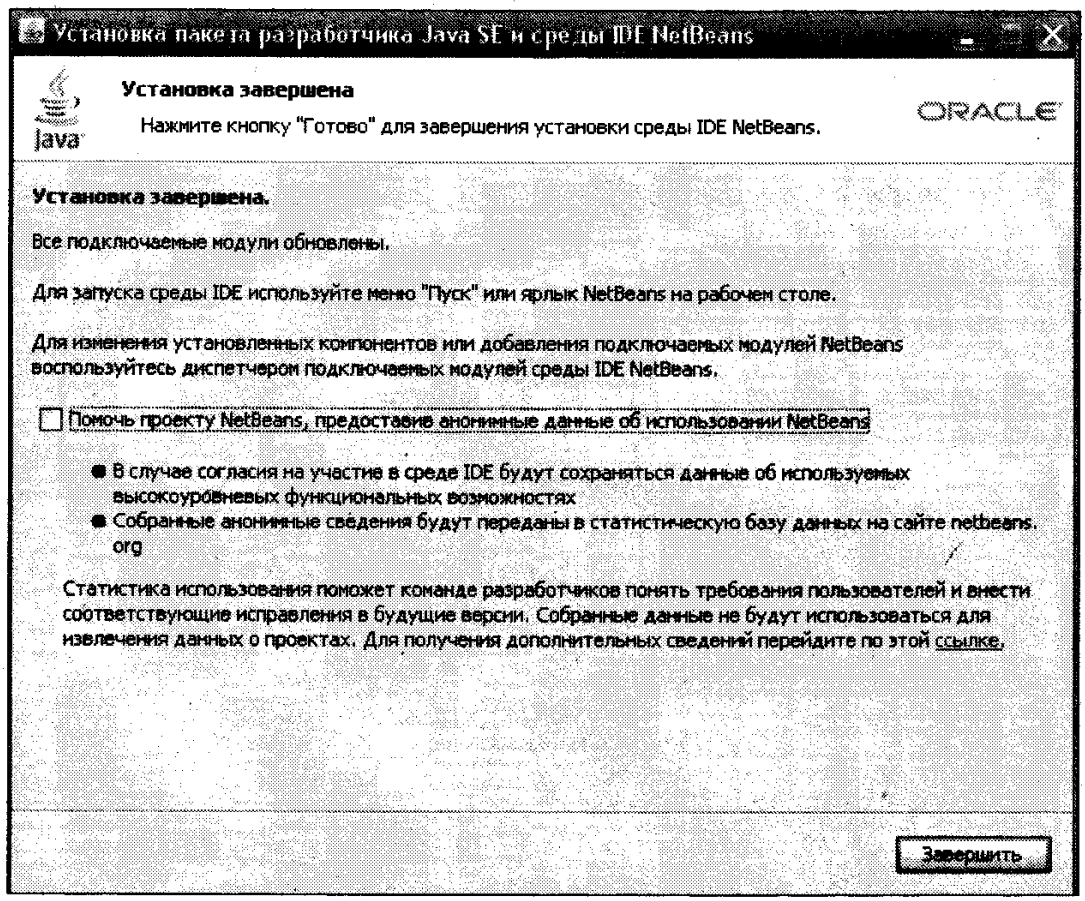


Рис. 2.5. Установка JDK и NetBeans (шаг 4)

2.2. Создаем свою первую программу на Java

После установки запускаем приложение NetBeans. В окне приложения выбираем команду **Файл ▶ Создать проект**, как показано на рис. 1.1.

В появившемся окне **Создать проект** выберите тип создаваемого приложения (рис. 1.3). В зависимости от версии и "комплектации" среды разработки NetBeans окно может выглядеть по-разному, но идея остается неизменной: необходимо указать язык разработки (если соответствующая версия NetBeans допускает использование нескольких языков) и непосредственно тип приложения. В данном случае выбираем в категории языков (список **Категории** в левой части окна) пункт **Java** (что соответствует языку Java), а в правой части окна в списке **Проекты** выбираем пункт **Приложение Java**.

В следующем окне **Новый Приложение Java Application** (рис. 1.4) следует указать название создаваемого проекта — в поле **Имя проекта**. По возможности имя должно быть кратким и информативным. В поле **Расположение проекта** указываем место, где будет сохранен проект. Флажок **Создать главный класс** оставляем установленными. В поле возле опции **Создать главный класс** дублируем название проекта `HelloWorld2` — в этом поле вводится название для главного класса программы. По окончании нажимаем Готово.

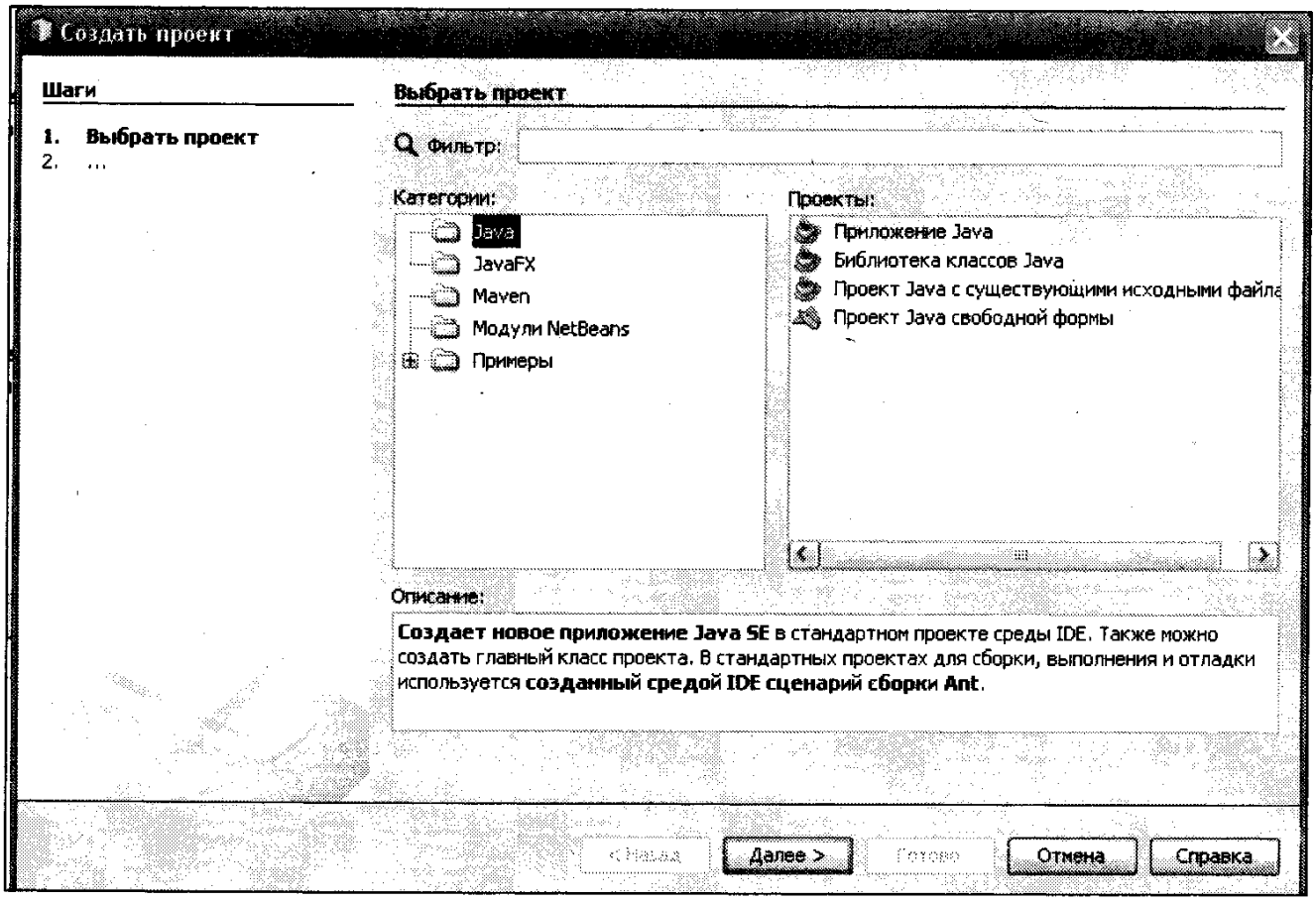


Рис. 2.6. Создание нового проекта (шаг 1)

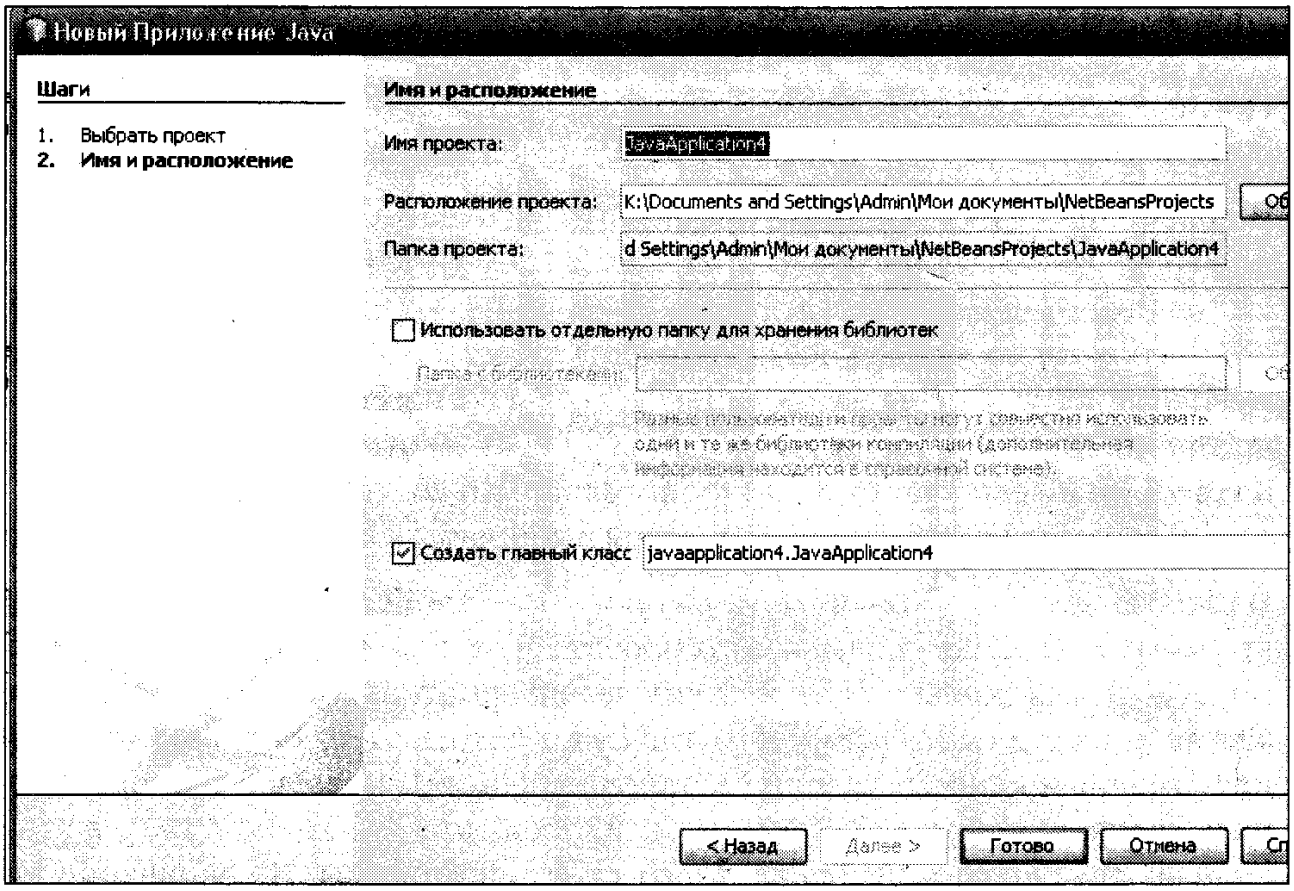


Рис. 2.7. Создание нового проекта (шаг 2)

Создается проект с шаблонным программным кодом, который вы потом можете менять, дополнять и т.д.

Начнем с примера:

Мы рекомендуем вам попробовать написать исходный текст во всех доступных вам средах разработки и редакторах, чтобы определить, какой из них вам больше всего подходит.

Для большинства языков программирования не важно, как называется файл исходного кода, но для компилятора языка Java имя файла существенно. Поэтому мы и указываем имя, под которым нужно сохранить исходный код первой программы: HelloWorld2.java.

```
public class HelloWorld2{
   public static void main(String[] args){
   System.out.println("Всем большой текстовый привет!");
   }
```

Результат представлен на рис 2.8.

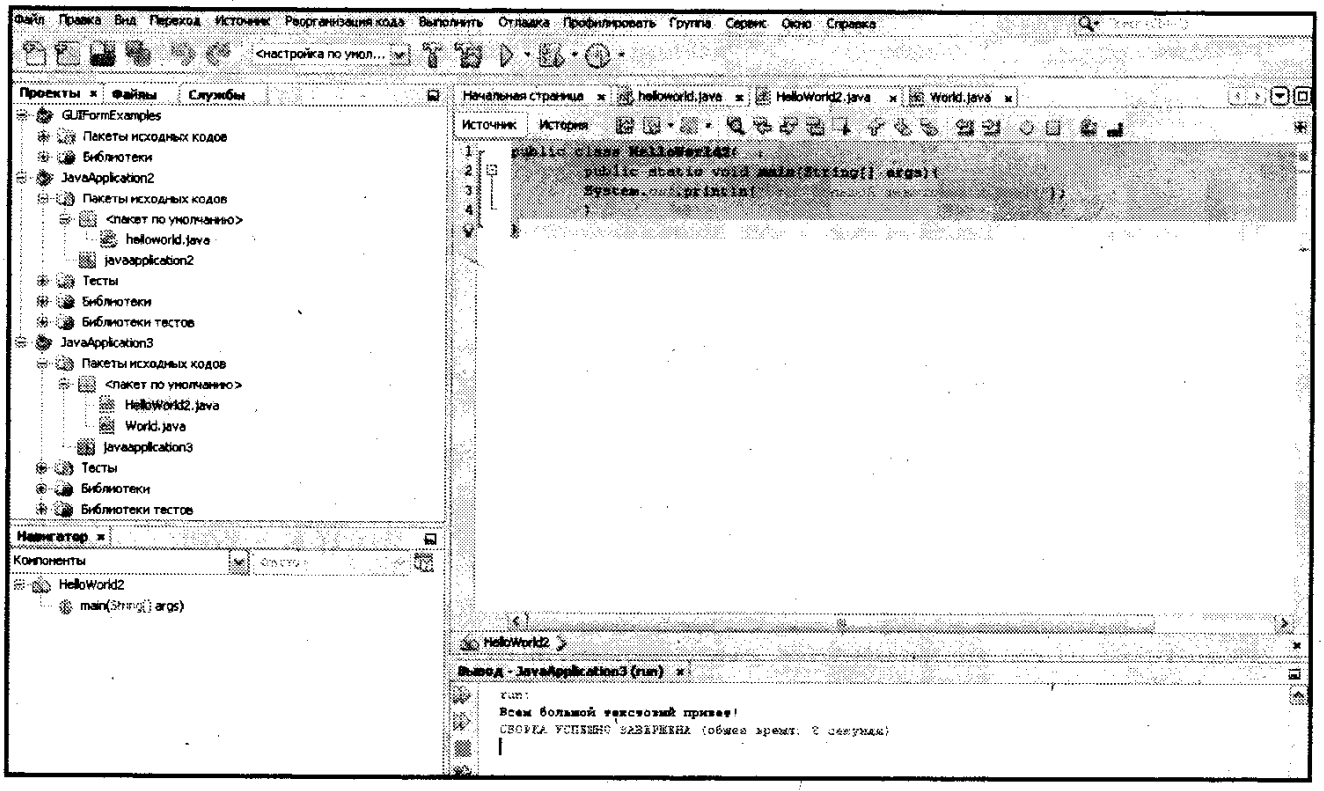


Рис. 2.8. Первая программа (вариант 1)

Та же простая программа, но с использованием графической библиотеки приведена ниже. При этом вывод программы осуществляется в виде окна (рис. 2.9).

```
import javax.swing.*;
public class helloworld {
    public static void main (String[] args){
   JOptionPane.showMessageDialog(null, "Привет, читатель книги
Java в примерах и задачах");
    }
}
```

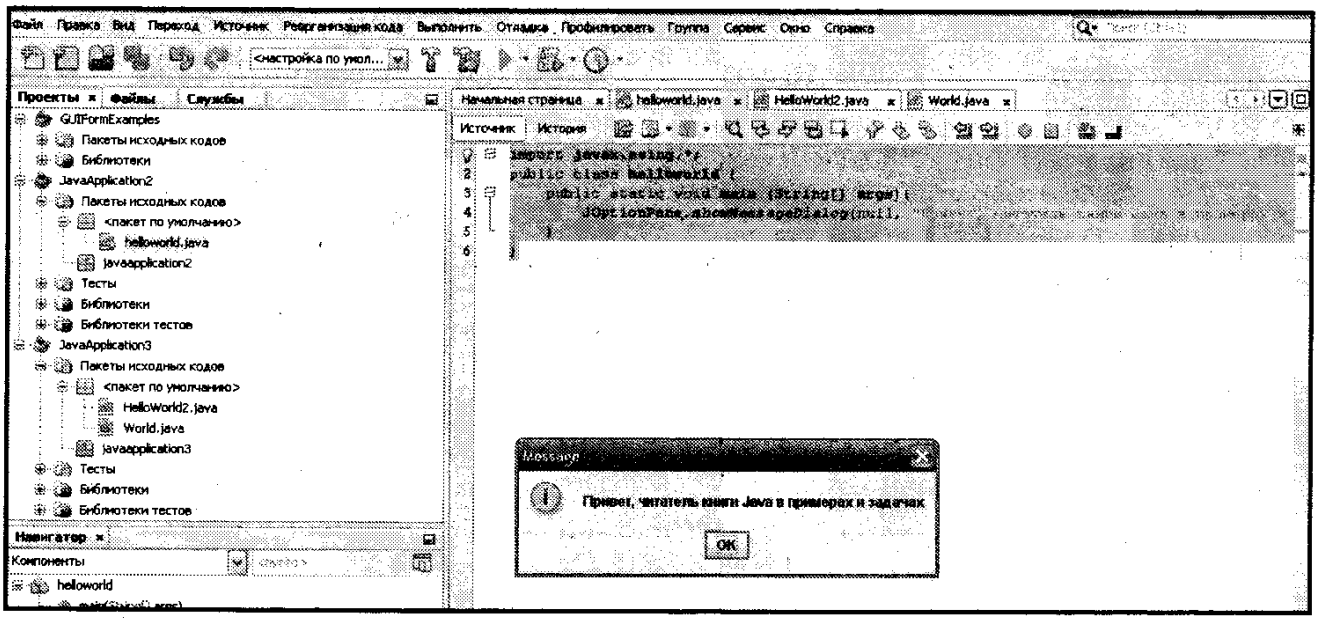


Рис. 2.9. Первая программа (вариант 2)

2.3. Основные конструкции языка

2.3.1. Переменные и присвоение значений

Действие, которое в большинстве программ встречается чаще всего, — это присвоение значений переменным. Любой язык программирования обязательно содержит команду присваивания, а то и несколько. Язык Java — не исключение.

Под переменной понимается именованная область памяти, содержимое которой является значением этой переменной. В ходе работы программы значение переменной может многократно меняться. Способность менять значение — это отличительное свойство переменных.

Напишем пример, демонстрирующий объявление переменных и присвоение им значений (см. листинг 2.2).

Листинг 2.2. Использование переменных

```
// Пример присвоения значений переменным
public class AssignExample {

 public static void main(String[] args) {

   // объявляем две переменные
   int var1;
   int var2;

   // присваиваем им значения
   var1 = 4096;
   var2 = var1 / 4;

   // выводим текущие значения переменных
   System.out.println("var1 = " + var1);
   System.out.println("var2 = var1 / 4 = " + var2);

 } // main()

} // AssignExample class
```

После запуска этой простой программы вы увидите на экране следующий вывод:

```
Y:\>java AssignExample
var1 = 4096
var2 = var1 / 4 = 1024
```

Разберем листинг 2.2 построчно. Первая незакомментированная строка метода `main()` содержит объявление переменной с именем `var1`:

```
int var1;
```

Это переменная целого (`int`) типа. Объявление переменной определенного типа указывает компилятору, сколько памяти он должен зарезервировать для хранения значений этой переменной. В Java существует непреложное правило, согласно которому каждая переменная должна быть объявлена перед первым ее использованием. Подробнее о типах мы поговорим далее.

Примечание

Программисты на C/C++ могли бы возразить, что в данном случае речь идет не об объявлении, а об определении переменной, поскольку под нее выделяется память. Однако в языке Java различия между этими двумя понятиями не существует, и закрепился термин «объявление переменной».

В следующей строке листинга точно так же объявляется переменная целого типа `var2`. Далее переменной `var1` присваивается целочисленная константа:

```
var1 = 4096;
```

Действие присваивания обозначается в языке Java символом «=». Символы или группы символов, которые служат для обозначения различных видов операций, называются операторами. Таким образом, символ «=» — это оператор присваивания. Далее в этой главе мы рассмотрим другие операторы.

```
var2 = var1 / 4;
```

В этой строке переменной `var2` присваивается значение не константы, а выражения. Значение вычисленного арифметического выражения становится значением переменной `var2`, при этом значение переменной `var1` не изменяется.

Арифметические операции в языке Java обозначаются общепринятыми символами (+, -, *, / и т. п.).

```
System.out.println("var1 = " + var1);
```

В этой строке значение переменной выводится на консоль с помощью уже знакомого вам метода `println()`. Новое для вас здесь — подготовка строки к выводу: результирующая строка формируется из текста в кавычках и преобразованного к строковому виду значения переменной `var1`. Оператор «+» здесь служит не для сложения чисел, а для слияния строк.

2.3.2. Управляющие конструкции

Как и в подавляющем большинстве других языков программирования, в Java можно изменить порядок выполнения команд. Для этого служат управляющие конструкции, две основные из которых мы рассмотрим в этом параграфе.

Условный оператор if

Условный оператор служит для ветвления программы. Под ветвлением мы понимаем ситуацию, когда последовательность выполнения команд изменяется в зависимости от выполнения некоторого условия.

В Java предусмотрены несколько форм записи условного оператора. Простейшая из них выглядит так:

```
if (условие) оператор;
```

Условие — это выражение, в результате вычисления которого получается логическое (`boolean`) значение. Логическое выражение может принимать одно из двух значений: истина (`true`) и ложь (`false`). Оператор выполняется только в том случае, если условие принимает значение `true`:

```
var1 = 99;
if (var1 < 100) System.out.println("Условие выполнено");
if (var1 > 100) System.out.println("Условие не
выполнено");
if (99 > 100) System.out.println("Условие не будет
выполнено никогда");
```

В результате выполнения этого фрагмента кода будет напечатана только строка `"Условие выполнено"`. При другом значении переменной `var1` могла бы быть выведена на консоль строка `"Условие не выполнено"`, а последняя строка не будет напечатана ни при каких обстоятельствах.

Для составления логических выражений в Java предусмотрен ряд операторов сравнения:

- `<` меньше;
- `<=` меньше или равно;
- `>` больше;

- >= больше или равно;
- == равно;
- != не равно;

Не путайте оператор присваивания (=) с оператором сравнения «равно» (==) — это очень распространенная ошибка.

Следующий пример демонстрирует применение условного оператора и логических выражений (листинг 2.3). Сохраните текст программы в файле по имени `IfStatementDemo.java`.

Листинг 2.3. Логические выражения

```
// демонстрация команды if
// сохраните текст в файл с именем IfStatementDemo.java
public class IfStatementDemo {

 public static void main(String[] args) {

   int a, b, c, d;

   a = 2;
   b = 3;

   System.out.println("a равно 2, b равно 3");
   if (a < b) System.out.println("a меньше b");
   if (a > b) System.out.println("этот текст вы никогда
                                  не увидите");
   System.out.println("");

   c = a - b; // c будет равно -1
   System.out.println("c равно -1");
   if (c >= 0) System.out.println("c имеет положительное
                                  значение ");
   if (c < 0) System.out.println("c имеет отрицательное
                                  значение");
   System.out.println("");

   d = b - a; // d равно 1
   System.out.println("d равно 1");
   if (d >= 0) System.out.println("d имеет положительное
                                  значение");
   if (d < 0) System.out.println("d имеет отрицательное
                                  значение");
   System.out.println("");
```

```
    if (a + c != b) System.out.println("a плюс c не равняется b");
    if (a + d == b) System.out.println("a плюс d равняется b");

  } // main() method

} // IfStatementDemo class
```

Попробуйте заранее вычислить, что будет выведено на экран в результате работы этой программы, и сравните с ее действительным выводом.

Новой для вас конструкцией в листинге 2.3 является объявление четырех переменных одной строкой:

```
int a, b, c, d;
```

Таким простым способом вы можете одной командой объявить несколько переменных одного и того же типа. Имена переменных должны быть разделены запятыми.

Примечание

Как и в других языках программирования, в Java предусмотрено использование нескольких вариантов условного оператора `if`. Основной из них мы описали здесь, а описание остальных приведено в п. 2.9.

Оператор цикла for

Цикл служит для многократного выполнения некоторой последовательности команд. В языке Java существует несколько разновидностей цикла. Простейший из них — цикл `for`, синтаксис которого выглядит очень знакомо для знающих C/C++:

```
for (инициализация; условие; итерация) оператор;
```

Примечание

Описание других циклов (цикла `while` и цикла `do...while`) приведено в п. 2.9.

Команды инициализации присваивают начальные значения переменным цикла. Условие — это логическое выражение, значение которого вычисляется на каждой итерации цикла. Если это значение равно `true`, то выполняется «`оператор`», а после нее выполняется «`итерация`», меняющая значение переменной цикла. Если значение «`условия`» равно `false`, то выполнение цикла прерывается и выполняется команда, следующая за циклом.

Следующий пример демонстрирует применение цикла `for` (см. листинг 2.4).

Листинг 2.4. Цикл for

```
// демонстрация цикла for
// сохраните исходный код в файл с именем
// ForStatementDemo.java
public class ForStatementDemo {

 public static void main(String[] args) {

   int count;

   for (count = 0; count < 5; count = count + 1)
     System.out.println("Переменная цикла равна " + count);

   System.out.println("Цикл окончен");

 } // main(String[]) method

} // ForStatementDemo class
```

В этом примере переменной цикла служит переменная `count` целого типа. Инициализация цикла присваивает ей значение 0. Текущее значение переменной `count` проверяется в условии: если оно меньше 5, то выполнение цикла продолжается, то есть вызывается метод `println()`, выводящий на консоль текущее значение, после чего значение переменной `count` увеличивается на 1.

В профессионально написанных программах вы чаще встретите сокращенную форму записи цикла `for`. Во-первых, переменная цикла будет объявлена непосредственно в части «инициализация»:

```
for (int count = 0; count < 5; count = count + 1)
```

Любая переменная в языке Java должна быть объявлена до ее использования, но нигде не сказано, что объявление должно стоять в самом начале программы. Переменную можно объявить где угодно, лишь бы это было сделано раньше, чем программа впервые к ней обратится. В том числе прямо в операторе цикла.

Второе изменение вы встретите в части «итерация». В Java, как и в C/C++, не принято писать

```
count = count +1;
```

В этих языках существует оператор инкремента ++, увеличивающий на 1 значение своего аргумента. Использование этого оператора позволяет записать предыдущую команду короче:

```
count++;
```

Обратный к нему оператор декремента уменьшает на единицу значение аргумента:

```
count--;
```

С двумя вышеназванными изменениями запись цикла `for` будет выглядеть так:

```
for (int count = 0; count < 5; count++)
```

Блоки

Блок кода — это ключевое понятие в языке Java, логическая единица текста программы. Блок представляет собой последовательность операторов, заключенную в фигурные скобки:

```
if (a < b) { // начало блока
 v = a + 1;
 w = b + v;
} // конец блока. Точку с запятой ставить не нужно!
```

Блок может быть и пустым, то есть внутри фигурных скобок может не быть ни одного оператора. Вместо любого оператора внутри блока может находиться другой блок — вложенный.

Программист может объединять операторы в блоки для того, чтобы:

- написать несколько операторов там, где по правилам языка полагается только один. Например, в уже рассмотренных операторах ветвления и цикла вместо одного «`оператор;`» можно использовать блок;
- сделать исходный текст более наглядным и понятным;
- ограничить область видимости переменных: переменная, объявленная в блоке, известна только внутри него.

Следующий пример демонстрирует применение блоков.

Листинг 2.5. Блоки кода

```
// пример использования блоков для выполнения нескольких
// операторов как одного
// сохраните исходный код в файл с именем BlockDemo.java
public class BlockDemo {

 public static void main(String[] args) {

   double i, j, d;

   i = 50;
   j = 150;
```

```
    if (i != 0) {

        System.out.println("делитель не равен нулю");
        d = j / i;
        System.out.println("j / i равно " + d);

    } // if (i != 0)

  } // main(String[])

} // BlockDemo class
```

Если условие в операторе `if` выполнено, то выполняется не один оператор, стоящий после `if`, а весь блок последовательно, то есть в результате работы этой программы на консоль будут выведены обе строки:

```
делитель не равен нулю
j / i равно 3.0
```

Обратите внимание, что после фигурной скобки, закрывающей блок, следует комментарий, указывающий, какой именно блок закрывается. Мы настоятельно рекомендуем комментировать так все блоки, чтобы было легче ориентироваться в структуре программы. Вы оцените преимущества такого комментирования, когда начнете писать длинные блоки или блоки, содержащие глубоко вложенные внутренние блоки.

2.3.3. Форматирование текста программы

К форматированию относятся правила употребления пробельных символов и размещения операторов в строке. Как уже сказано, в языке Java каждый оператор заканчивается точкой с запятой. Собственно говоря, то, что заканчивается точкой с запятой, и есть оператор, или логическая единица программы. Точку с запятой можно ставить и после блока, но это не имеет смысла, поскольку конец блока и так отмечен правой фигурной скобкой.

Ни конец строки, ни другой пробельный символ не считается в Java ни разделителем, ни завершителем оператора, поэтому не важно, записывать ли каждый оператор в отдельной строке или несколько в одной. Следующие фрагменты кода эквивалентны:

```
x = y;
y++;
System.out.println(x + " " + y);
```

и

```
x = y; y++; System.out.println(x + " " + y);
```

Можно и даже рекомендуется записывать один оператор в несколько строк. Это значительно облегчает чтение текста программы, а на итоговый байт-код, который генерируется компилятором Java, не влияет:

```
for (int count = 0;
            count < 10;
            count++)
      System.out.println("Переменная цикла равна " + count);
```

Java — язык со свободным форматированием, поэтому вы можете не только разбивать оператор на несколько строк, но и разделять ключевые слова и идентификаторы любым количеством любых пробельных символов. Чаще всего пробельные символы используются для организации отступов, облегчающих понимание структуры программы.

Среди программистов сложились неписаные правила оформления кода, которых мы и будем придерживаться в этой книге. Рекомендуем вам тоже придерживаться их — кроме неудобства чтения текста, несоблюдение правил производит впечатление непрофессионально написанной программы.

Все операторы внутри блока пишутся с одинаковым отступом, блока следующего уровня вложенности — с двойным, и так далее:

```
if (x != 0) {
 // внешний блок
 //
 for (int count = 0; count < 5; count++) {
   // внутренний блок
   //
 } // конец внутреннего блока
} // конец внешнего блока
```

Левую фигурную скобку, открывающую блок, можно ставить и в отдельной строке:

```
if (x != 0)
{
 // блок кода
}
```

Свобода форматирования позволяет использовать пробельные символы (в том числе символ табуляции) и скобки для того, чтобы улучшить читаемость выражений, составляющих программу. Например, следующие записи эквивалентны, но вторая из них явно удобнее для восприятия:

```
x=10/3*(255/z);
x = 10 / 3 * (255/z);
```

Круглые скобки вообще-то предназначены для изменения последовательности вычисления выражения — подвыражения, взятые в скобки, вычисляются первыми — но их рекомендуется использовать и там, где такой необходимости нет. Сравните с точки зрения наглядности два эквивалентных выражения, во втором из которых скобки служат только для визуальной группировки элементов:

```
x = z / 4 - 23 * a + 124;

x = (z/4) - (23*a) + 124;
```

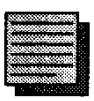
2.4. Типы данных

Тип данных — это характеристика переменной или константы, определяющая, какого рода значение хранится в отведенной для нее области памяти: числовое (целое или вещественное), символьное, логическое, объект какого-либо класса или другое.

Каждый язык программирования предоставляет программисту несколько типов данных для представления различных значений переменных. Типы данных языка во многом определяют его возможности и круг задач, которые на этом языке можно решать. Типы данных и их использование — обширнейшая тема, которой можно было бы посвятить четверть этой книги, но мы ограничимся изложением только самых необходимых сведений.

Напомним, что Java — строго типизированный язык, из чего следует, в числе прочего, что любая переменная перед ее использованием должна быть объявлена с указанием ее типа.

Встроенные типы данных Java делятся на две категории: простые (primitive) и ссылочные (reference). К ссылочным типам относятся массивы, классы, интерфейсы и перечисления. Простые типы данных делятся на числовые (целые и вещественные) и логические, причем символьный тип относится к числовым.



Примечание

Такие естественные для программистов на других языках типы данных, как строковый и дата/время, реализованы в Java как ссылочные: классами `String` и `Date`.

Целочисленные типы

Для хранения целых чисел в Java предусмотрены четыре типа данных (табл. 2.1). Все они хранят целые числа со знаком. Беззнаковых числовых типов, как в C/C++, в языке Java нет.

Характеристики целочисленных типов данных *Таблица 2.1*

Тип	Разрядность	Диапазон значений
byte	8 битов	-128...127
short	16 битов	-32 768...32 767
int	32 бита	-2 147 483 648...2 147 483 647
long	64 бита	-9 223 372 036 854 775 808...9 223 372 036 854 775 807

Тип `int` (integer) используется чаще всего. Обычно именно этот тип имеют переменные цикла, индексы массива и т. п. Тип `long` предназначен для хранения чисел, выходящих за пределы диапазона `int`.

Меньше всего памяти занимает число типа `byte` — один байт. Это единственный тип данных в языке Java, который используется для побайтовой обработки двоичных данных.

Последний тип, `short`, отличается тем, что старший байт двухбайтового числа хранится в памяти первым. Так хранились целые числа на 16-разрядных компьютерах, а в наше 32-разрядное время этот тип данных почти вышел из употребления.

Символьный тип

К целочисленным типам относится и тип `char`, предназначенный для представления одного символа.

К хранению символов национальных алфавитов разработчики Java отнеслись с особым вниманием, потому что приложения, написанные на Java, должны были работать по всему миру. В отличие от большинства других языков, где символ занимает один байт, размер типа `char` в Java составляет **2 байта**. Такой размер необходим для хранения символа по стандарту Unicode.

Unicode — это стандарт кодирования множества национальных алфавитов, видов письма, технических и математических символов, разработанный для того, чтобы разрешить проблемы интернационализации в многоязычной компьютерной среде. Более ранний стандарт ASCII, в котором каждый символ представлен одним байтом, был пригоден только для европейских языков. Таблица ASCII вошла в набор Unicode как его подмножество.

Собственно Unicode — это набор (последовательность) символов, который можно закодировать разными способами. Язык Java использует кодировку UTF-8, которая отводит по одному байту для букв западноевропейских алфавитов, по два байта — для восточноевропейских, а для алфавитов экзотических языков — три и более байтов. При этом строки, содержащие только 7-битные символы из первой половины таблицы ASCII, имеют одинаковое представление как в ASCII, так и в UTF-8.

Переменной типа `char` можно присвоить в качестве значения одиночный символ — как печатный, так и управляющий. Например, переменной `ch` можно присвоить букву `L`:

```
char ch;
ch = 'L';
```

Вывести на консоль значение переменной типа `char` можно с помощью метода `println()`:

```
System.out.println("Значение переменной ch равно " + ch);
```

Поскольку тип данных `char` относится к целочисленным, над ним можно выполнять арифметические операции. Следующий пример демонстрирует применение операторов инкремента `++` и декремента `--`.

Листинг 2.6. Операции инкремента и декремента над переменной символьного типа

```
// сохраните исходный код в файле с именем CharArithDemo.java
public class CharArithDemo {

 public static void main(String[] args) {
   char ch;

   ch = 'L';
   System.out.println("ch равно " + ch);

   ch++;
   System.out.println("значение ch изменилось на " + ch);

   ch = 'Д';
   ch--;
   System.out.println("значение ch снова изменилось
                       и равно " + ch);
 } // main(String[])

} // CharArithDemo class
```

Эта программа выведет следующие строки:

```
ch равно L
значение ch изменилось на M
значение ch снова изменилось и равно Г
```

Важное замечание: символ ('A') и строка из одного символа («A») — это совершенно разные вещи с точки зрения языка Java. Данные символьного и строкового типа хранятся в памяти по-разному.

Типы данных для вещественных чисел

Вещественное число, как известно, состоит из целой и дробной части, разделенных десятичной точкой. Для представления вещественных чисел Java располагает двумя типами данных: `float` и `double`. Первый из них предназначен для хранения чисел с 7-8 значащими цифрами, второй — для чисел с двойной точностью (15-16 значащих цифр).

Характеристики вещественных типов данных *Таблица 2.2*

Тип	Разрядность	Диапазон значений
float	32 бита	3.4e-38 ... 3.4e+38
double	64 бита	1.7e-308 ... 1.7e+308

Тип данных `double` используется гораздо чаще, чем `float`. С ним работают все математические функции библиотек Java. Например, метод `sqrt()`, определенный в классе `Math` и вычисляющий квадратный корень своего аргумента, принимает аргумент типа `double` и возвращает значение этого же типа. Следующий пример демонстрирует применение метода `sqrt()` для вычисления длины гипотенузы прямоугольного треугольника по теореме Пифагора.

Листинг 2.7. Пример работы с вещественными числами

```
// вычисление длины гипотенузы по теореме Пифагора
public class HypotDemo {

 public static void main(String[] args) {
   double cathetus1, cathetus2, hypot;

   cathetus1 = 3; // длина первого катета
   cathetus2 = 4; // длина второго катета

   hypot = Math.sqrt((cathetus1 * cathetus1) +
                     (cathetus2 * cathetus2));
```

```
    System.out.println("длина гипотенузы равна " + hypot);
  } // main(String[])
} // class HypotDemo
```

Логический тип

Переменная типа `boolean` (логического) может принимать всего два значения: `true` (истина) и `false` (ложь). Для наглядности приведем небольшой пример:

Листинг 2.8. Демонстрация переменных логического типа

```
// пример использования типа данных boolean
// сохраните в файл BoolDemo.java
public class BoolDemo {
 public static void main(String[] args) {
   boolean b;

   b = false;
   System.out.println("b равно " + b);
   b = true;
   System.out.println("b равно "+ b);

   // переменная логического типа может стоять
     // в условии оператора if
   if (b) System.out.println("Как вы думаете, увидите
                              ли вы эту строку?");

   // операторы сравнения возвращают логическое значение
   System.out.println("Выражение 10 > 9 имеет значение "
                      + (10 > 9));
 } // main(String[] args)

} // BoolDemo class
```

Когда вы запустите программу BoolDemo, вы можете заметить, что метод `println()` выводит значение логического типа в виде строки `true` или `false`. Из листинга видно также, что при условном операторе не обязательно должно стоять логическое выражение — с таким же успехом интерпретатор Java может проверять значение переменной типа `boolean`. Благодаря этому отпадает необходимость писать громоздкие конструкции вроде:

```
if (b == true) System.out.println("true");
   /*хотя это тоже допустимо*/
```

В последней команде вывода метод `println()` печатает значение не переменной, а логического выражения `(10 > 9)`, возвращающего значение `true`. Забегая вперед, скажем, что выражение в данном случае должно быть взято в скобки, потому что оператор конкатенации строк «+» имеет более высокий приоритет, чем оператор сравнения «>».

2.5. Литералы и константы

Что такое «литерал» и что такое «константа»

Литерал — это постоянное значение, записанное в программе в читаемой форме. Например, число 100 — это литерал, одиночный символ и текстовая строка тоже могут быть литералами и т. д.

Литерал отличается от константы и переменной тем, что ему не может быть присвоено значение. На самом деле, в своей программе вы в качестве значений констант и переменных обычно используете литералы. Например:

```
Counter = 1;
Textstr = "Произвольная строка";
ch = '?';
A = 4e-1;
```

Здесь `Counter`, `Textstr`, `S`, `A` — это переменные, а `1`, `"Произвольная строка"`, `'?'`, `4e-1` — литералы.

Литерал может иметь значение любого простого типа данных. Тип литерала определяется формой его записи. Например, `'?'` — это литерал символьного типа, а «?» — строкового; 10 — типа `int`, а 10.0 — типа `double`. Вещественные числа записываются либо с десятичной точкой, либо в так называемой научной (экспоненциальной) нотации с указанием степени числа 10: `4e-1` — то же самое, что 0.4.

Для того, чтобы заставить компилятор считать литерал, опознаваемый им как данные одного типа, данными другого типа, служат модификаторы типа. Символ `L` (заглавный или строчный) превращает литерал типа `int` в литерал типа `long`:

```
int n1 = 24;
long n2 = 24L;
```

Вещественные литералы имеют по умолчанию тип данных `double`. Чтобы его поменять на `float`, добавьте к присваиваемому значению символ `F`:

```
float f1 = 12.34F;
```

Целочисленные литералы распознаются как тип `int`, но их можно присваивать также переменным других целых типов: `short`, `byte` и `char` — при условии, что значение литерала не выходит за пределы диапазона для соответствующего типа, а переменной типа `long` — без всяких условий.

Шестнадцатеричные и восьмеричные литералы

Если вы уже когда-нибудь писали программы, то знаете, что нередко приходится работать с числами, которые удобнее представлять не в десятичной записи. Язык Java позволяет без труда использовать восьмеричную и шестнадцатеричную формы записи:

```
hex = 0xFF // в десятичной системе соответствует числу 255
oct = 011 // в десятичной системе соответствует числу 9
```

Если запись числа начинается с нуля, то число понимается как восьмеричное; если в префиксе «0x», то — как шестнадцатеричное.

Специальные символы (Esc-последовательности)

Как уже сказано, переменной типа `char` можно присваивать в качестве значения не только алфавитно-цифровой символ, но и такой, который нельзя ввести с клавиатуры — например, символы, посылаемые клавишами «Tab» или «Backspace». Для того, чтобы использовать такие символы в литералах, служат так называемые Esc-последовательности, то есть символ обратного слэша, за которым следует код вводимого символа (табл. 2.3).

Esc-последовательности в языке Java *Таблица 2.3*

Последовательность	Значение
\'	апостроф (single quote)
\"	кавычка (double quote)
\\	обратный слэш (backslash)
\r	возврат каретки (carriage return)
\n	перевод строки (new line)
\f	прогон страницы (form feed)
\t	табуляция (horizontal tab)
\b	забой (backspace)
\ddd	код символа в восьмеричной записи (ddd — восьмеричные цифры)
\uhhhh	код символа в шестнадцатеричной записи (hhhh — шестнадцатеричные цифры)

Примеры:

```
char ch;
ch = '\t'; // знак табуляции
ch = '\''; // символ апострофа
```

Строковые литералы

Распространенной разновидностью литерала в языке Java является строка — последовательность символов, заключенная в кавычки:

```
System.out.println("Это строковый литерал");
```

А что делать, если внутри самой строки встречается кавычка? Не будет ли она воспринята как символ завершения строки? Будет, если ее не экранировать с помощью Esc-последовательности:

```
System.out.println("O\'Key"); // строка с апострофом
System.out.println("Строка с \"кавычками\"");
```

Часто строковые литералы используются для указания пути к некоторому файлу. В ОС Windows символом-разделителем пути служит обратный слэш. Его тоже необходимо экранировать с помощью Esc-последовательности:

```
String filename = "c:\\project\\helloworld\\HelloWorld.java";
```

Примечание

В ОС семейства Unix для разделения пути служит прямой слэш (/). Интерпретатор Java позволяет использовать этот символ и на платформе Windows. Рекомендуется для единообразия всегда писать пути через прямой слэш. Экранировать этот символ не нужно.

Следующий пример показывает, как использовать Esc-последовательности для представления специальных символов внутри строки.

Листинг 2.9. Символы форматирования текста

```
public class StrDemo {
 public static void main(String[] args) {

   // для перевода каретки (начала новой строки)
    // служит код \n
   System.out.println("Первая строка\nВторая строка");

   // знак табуляции помогает оформить выводимый текст
    // в колонки. Он вводится с помощью кода \t
   System.out.println("A\tB\tC");
   System.out.println("D\tE\tF");

 } // main(String[]) method
} // StrDemo class
```

2.6. Переменные

С понятием переменной вы уже встречались в начале главы. Мы уже как минимум два раза обращались к этому вопросу. Поэтому будем рассматривать только то, о чем еще не говорилось, сначала кратко суммируя уже полученную информацию.

Итак, вы можете объявить переменную любого из типов, имеющихся у вас в распоряжении. Объявление переменной создает «экземпляр» этого типа, то есть выделяет область памяти соответствующего размера для хранения значений переменной. Нельзя присваивать переменной значение, выходящее за пределы диапазона, заданного для ее типа, то есть невозможно, например, поместить вещественное значение в переменную типа `boolean`. Невозможно также на ходу изменить тип уже существующей переменной.

Любая переменная должна быть объявлена до ее первого использования. Причина этого требования — строгий контроль типов в языке Java: при первом обращении к переменной компилятор проверяет, в соответствии ли с объявленным типом она используется, и если нет, то сигнализирует о синтаксической ошибке.

Динамическая инициализация переменной

Вы уже знаете, как инициализировать переменную, то есть присваивать ей начальное значение. Вы знаете также, что можно совместить инициализацию с объявлением и что можно объявлять сразу несколько переменных одного типа:

```
int a, b;
a = 10;
b = 2;
int c = 12;
```

В приведенных примерах переменной в качестве значения присваивается константа или константное выражение. В Java существует и другая возможность: инициализировать переменную значением динамического выражения, в частности — значением другой переменной, если оно ей уже присвоено.

Продемонстрируем динамическую инициализацию на примере вычисления объема цилиндра.

Листинг 2.10. Динамическая инициализация переменной

```
public class CylinderVolume {
 public static void main(String[] args) {

   // эти переменные инициализируются константами
   double radius = 4; // радиус цилиндра
   double height = 5; // высота цилиндра

   // переменная volume инициализируется динамически
   // вычисленным значением выражения
   double volume = 3.1416 * radius * radius * height;
   System.out.println("Объем цилиндра равен " + volume);

 } // main(String[]) method
} // CylinderVolume class
```

Область действия переменных

До сих пор мы всегда объявляли переменные в начале метода `main()`. Вы уже знаете, что объявлять переменную можно где угодно в пределах блока — группы команд, заключенных между левой и правой фигурными скобками.

Как уже сказано, блок ограничивает область действия переменных. В этом параграфе мы поясним, что это значит.

Область действия влияет на переменные двояким образом. Во-первых, она ограничивает видимость переменной из других частей программы. Во-вторых, она определяет срок, в течение которого переменная существует и может быть использована.

В большинстве языков программирования области действия бывают локальными и глобальными. Хотя в Java тоже можно использовать такое деление, это будет не точно. Области действия в Java определяются в терминах классов и методов. Сейчас мы рассмотрим только области действия переменных в рамках метода. О доступности классов мы будем говорить позже, излагая основы объектно-ориентированного программирования.

Общее правило таково: переменная, объявленная внутри блока, доступна из программного кода, находящегося в этом блоке и вложенных в него блоках любого уровня вложенности, и недоступна из внешних по отношению к нему блоков. Поясним это на примере (см. листинг 2.11).

Листинг 2.11. Блок как область действия переменных

```
public class ScopeDemo {
 public static void main(String[] args) {

   int x; // переменная x видна во всем методе  main()

   x = 1;
   System.out.println("До вложенного блока: x равно " + x);
   {
     // новый блок создает новую область видимости

     /* переменная y доступна только внутри текущего
       блока, но это не мешает ее использованию вместе
         с переменной x из внешнего блока
     */
     int y = 3;

     //здесь действуют обе переменных, x и y
     System.out.println("Внутренний блок: x равно " +
                          x + ", y равно " + y);
     x = y * 3;
   } // конец вложенного блока

   /* если вы раскомментируете строку "y = 100", то получите
   ** ошибку компиляции: переменная y в текущем блоке не объявлена
   */
   // y = 100;

   // как видите, переменная x видна во всей программе
   System.out.println("После вложенного блока: x равно " + x);

 } // main(String[]) method

} // ScopeDemo class
```

В начале программы мы объявляем переменную `x`, которая благодаря этому доступна из любого места в методе `main()`. Во вложенном блоке мы объявляем переменную `y`, видимую и доступную только изнутри этого блока. После завершения выполнения блока переменная `y` прекращает свое существование: далее в объемлющем блоке она не известна, поэтому выражение `y = 100` является синтаксической ошибкой.

Что происходит, если блок, в котором объявлена переменная, представляет собой тело цикла? Попробуйте догадаться, каков будет результат выполнения следующего фрагмента кода:

```
for (int i = 1, i < 5; i++) {
 int x = 100;
 System.out.println("Получилось " + (x - i));
}
```

Новичок предположил бы, что переменная x инициализируется единожды, при первом прогоне цикла, то есть в первой строке вывода будет число 99, во второй — 97, в третьей — 94 и т. д. На самом же деле при каждом повторе цикла переменная x заново объявляется и инициализируется значением 100, то есть вывод будет выглядеть так:

```
Получилось 99
Получилось 98
Получилось 97
Получилось 96
```

Очевидно, что в программе на языке Java могут быть две переменные с одинаковыми именами, если их области действия не пересекаются. В других языках возможна ситуация, когда переменные с одинаковыми именами объявлены во внешнем и внутреннем блоках, причем на время работы внутреннего блока внешняя переменная для него как бы не существует.

В Java такая ситуация является синтаксической ошибкой, то есть в пересекающихся областях видимости двух переменных с одинаковыми именами быть не может (см. листинг 2.12).

Листинг 2.12. Переменные с одинаковыми именами

```
public class NestVar {
 public static void main(String[] args) {
   int count=1;
   System.out.println("count = " + count);

   {
   // ошибка: переменная count уже объявлена.
   // Закомментируйте следующую строку
   int count=2;
   System.out.println("Первый независимый блок:
                       count = " + count);

   int n=3;
   System.out.println("Первый независимый блок: n = " + n);
   }
   {
   int n=5;
```

```
        System.out.println("Второй независимый блок: n = " + n);
      }

  } // main(String[]) method

} // NestVar class
```

Идентификаторы и ключевые слова

Идентификатор — это имя, присвоенное методу, переменной или другому объекту, созданному пользователем. Идентификаторы состоят из букв (в том числе кириллических), цифр и специальных символов и согласно спецификации подчиняются следующим правилам:

- идентификатор не может совпадать с одним из зарезервированных ключевых слов языка Java;
- идентификатор должен начинаться с буквы, знака подчеркивания (_) или знака доллара ($);
- второй и последующие символы могут быть буквами, цифрами, знаками подчеркивания или доллара;
- заглавные и строчные буквы различаются.

Кроме того, практика программистов выработала следующие неписаные правила, облегчающие чтение программы:

- идентификатор должен отражать назначение объекта (переменной, класса или метода);
- идентификатор не должен совпадать с именем какого-либо стандартного метода (например, `println`);
- в идентификаторах, сформированных из нескольких слов, слова разделяются знаком подчеркивания (например, `line_count`).

Зарезервированные слова языка Java *Таблица 2.4*

abstract	assert	boolean	break	byte	case	catch
char	class	const	continue	default	do	double
else	extends	final	finally	float	for	goto
if	implements	import	instanceof	int	interface	long
native	new	package	private	protected	public	return
short	static	strictfp	super	switch	synchronized	this
throw	throws	transient	try	void	volatile	while

Слова `const` и `goto` относятся к зарезервированным, но в настоящее время они в языке Java не употребляются.

2.7. Операторы

С операторами вы уже несколько раз встречались, а в этом параграфе мы рассмотрим понятие оператора подробнее. **Оператор — это средство языка программирования, с помощью которого записывается алгоритм программы. Обычно это символ, сообщающий компилятору, какую математическую или логическую операцию должен выполнить компьютер.**

В языке Java существует шесть основных групп операторов:

- арифметические;
- логические;
- операторы сравнения;
- побитовые операторы;
- операторы присваивания;
- тернарный оператор.

Арифметические операторы

Кроме уже знакомых вам операторов, реализующих четыре арифметических действия, к арифметическим операторам в языке Java относятся еще три (табл. 2.5).

Арифметические операторы *Таблица 2.5*

Оператор	Назначение
+	сложение
-	вычитание
*	умножение
/	деление
%	остаток от деления
++	инкремент (увеличение значения на единицу)
- -	декремент (уменьшение значения на единицу)

Первые четыре оператора применимы к переменным, литералам и константам любого из числовых типов, включая тип `char`, и результат их применения именно таков, как можно было бы ожидать. Единственная тонкость связана с оператором деления: если и делимое, и делитель имеют целочисленный тип, то оператор «/» выполняет деление с остатком, то есть возвращает целую часть частного. Например, результатом выражения 10 / 3 будет число 3, а результатом деления вещественных чисел 10.0 / 3.0 — вещественное число 3.3333333333333335.

Остаток от деления можно получить с помощью оператора %. Если вернуться к предыдущему примеру, то результатом выражения 10 % 3 будет 1. К вещественным числам деление с остатком тоже применимо. Следующий пример демонстрирует деление целых и вещественных чисел.

Листинг 2.13. Операторы деления

```
public class DivisionDemo {
   public static void main(String[] args) {
       int iresult, iremain; //для деления целых чисел
       double dresult, dremain;
                        // для деления вещественных чисел

       //  деление целых чисел
       iresult = 10 / 3;
       iremain = 10 % 3;

       System.out.print("частное от деления 10 / 3 равно ");
       System.out.println(iresult +
                          ", остаток равен " + iremain);

       // деление вещественных чисел
       dresult = 10.0 / 3.0;
       dremain = 10.0 % 3.0;

       System.out.print("частное от деления 10.0 / 3.0 равно ");
       System.out.println(dresult +
                          ", остаток равен " + dremain);

   } // main(String[]) method

} // DivisionDemo
```

Эта программа должна вывести следующие строки:

```
частное от деления 10 / 3 равно 3, остаток равен 1
частное от деления 10.0 / 3.0 равно 3.3333333333333335,
остаток равен 1.0
```

Операторы инкремента ++ и декремента -- мы уже упоминали, рассматривая оператор цикла `for`. Они применимы к переменной целочисленного типа и служат для увеличения/уменьшения ее значения на единицу. К константе или выражению операторы ++ и -- применять нельзя.

Оба этих оператора могут стоять как до аргумента (префиксная форма), так и после (постфиксная форма). Следующие выражения эквивалентны:

```
x++; y--; // постфиксная форма
++x; --y; // префиксная форма
```

Разница между формами записи операторов ++ и -- проявляется, когда переменная, к которой применен один из этих операторов, участвует в выражении. Возьмем для определенности оператор инкремента. Если

оператор ++ стоит перед операндом, то сначала увеличивается значение операнда, а потом вычисляется значение выражения. Если же оператор ++ стоит после операнда, то сначала вычисляется выражение и только после этого увеличивается значение операнда. Сравните фрагменты кода:

```
int x = 10;
int y = ++x; // новое значение y == 11; новое значение x == 11
```

и

```
int x = 10;
int y = x++; // новое значение y == 10; новое значение x == 11
```

Логические операторы и операторы сравнения

Логические операторы применяются к операндам типа `boolean`. Мы рассматриваем операторы сравнения в этом же параграфе, потому что они возвращают значение логического типа и чаще всего используются вместе с логическими операторами. Сначала рассмотрим логические операторы (табл. 2.6).

Логические операторы *Таблица 2.6*

Оператор	Назначение
!	отрицание (NOT)
&	конъюнкция (AND)
|	дизъюнкция (OR)
^	исключающее ИЛИ (XOR)
&&	сокращенная конъюнкция
||	сокращенная дизъюнкция

Оба операнда логического оператора должны иметь тип `boolean`, и возвращаемое значение имеет тот же тип. В таблице 2.7 приведены результаты выполнения логических операций над всеми возможными значениями операндов (таблица истинности логических операторов). Операторы NOT, AND (и) и OR (или) действуют именно так, как следует из их названия, а оператор XOR возвращает значение `true` тогда и только тогда, когда его значения его операндов различны.

Логические операции *Таблица 2.7*

x	y	x & y	x | y	x ^ y	! x
false	false	false	false	false	true
true	false	false	true	true	false
false	true	false	true	true	true
true	true	true	true	false	false

Сокращенные операторы && и ||, называемые также условными, применяются в условиях команд `if`, `while` или `do` для вычисления значения истинности условия. Эти операторы отличаются от своих аналогов & и | порядком выполнения. При выполнении логической операции вычисляется значение обоих операндов, а для оценки истинности условия вычисление второго операнда может оказаться излишним: например, если первый операнд оператора && имеет значение `false`, то независимо от значения второго операнда результатом операции будет `false`. Точно так же, если первый операнд оператора || имеет значение `true`, то независимо от значения второго результатом операции будет `true`. В обоих случаях в вычислении второго операнда нет никакой необходимости и условные операторы его не вычисляют, поэтому они и называются сокращенными. Таким образом, код получается более эффективным.

Операторы сравнения перечислены в таблице 2.8.

Операторы сравнения *Таблица 2.8*

Оператор	Назначение
==	равно
!=	не равно
>	больше
<	меньше
>=	больше или равно
<=	меньше или равно

Операндами операторов == и != могут быть переменные, константы, литералы и объекты любого типа. Остальные четыре оператора сравнения применимы к объектам только таких типов, значения которых упорядочены. Порядок предусмотрен для всех числовых типов, включая символьный (алфавитный порядок или место в наборе символов), но не, например, для типа `boolean`. Это означает, что из всех операторов сравнения к значениям `true` и `false` применимы только == и !=.

Побитовые операторы

Несмотря на то, что побитовые операции не так часто осуществляются, как арифметические, логические и операции сравнения, тем не менее, бывают случаи, когда их отсутствие могло бы ограничить область применения языка Java — побитовые операции часто используются при решении задач системного программирования. Как следует из названия, они выполняются над отдельными битами операндов.

Побитовые операции могут выполняться над операндами целочисленных типов: `byte`, `char`, `short`, `int` и `long`. Перед выполнением операции оба операнда преобразуются к одному типу: `int` или `long`,

если один из операндов имел тип `long`. Результатом операции является число того же типа.

К операндам типа `boolean`, `float` или `double`, а также к экземплярам классов такие операции не применимы.

Побитовые операторы *Таблица 2.9*

Оператор	Назначение
~	дополнение (NOT)
&	побитовая конъюнкция (AND)
|	побитовая дизъюнкция (OR)
^	побитовое исключающее ИЛИ (XOR)
<<	сдвиг влево
>>	сдвиг вправо
>>>	беззнаковый сдвиг вправо

Первые четыре побитовых оператора обозначаются такими же символами, как логические операторы, и работают так же, с той только разницей, что они обрабатывают свои операнды бит за битом. «Таблица истинности» побитовых логических операторов приведена в таблице 2.10.

Значения побитовых операторов *Таблица 2.10*

x	y	x & y	x | y	x ^ y	~x
0	0	0	0	0	1
1	0	0	1	1	0
0	1	0	1	1	1
1	1	1	1	0	0

Побитовая конъюнкция часто используется как средство сброса (выключения) отдельных битов. Если в одном из операндов выражения имеется нулевой бит, то у числа, получающегося в результате выполнения операции AND, бит, стоящий в этой же позиции, будет равен нулю:

```
10101010 & 11010011 = 10000010
```

Пример такого применения операции «побитовое И» мы рассмотрим далее в программе, заменяющей строчные буквы прописными.

Другим применением побитового AND является проверка того, установлен ли у проверяемого значения определенный бит:

```
if ((value & 8) != 0)
      System.out.println("Четвертый бит установлен");
```

Число 8 в двоичной системе счисления записывается как 1000 (четвертый справа бит равен 1, а остальные — 0), и условие выполняется только тогда, когда в значении переменной `value` четвертый справа бит тоже равен 1.

Побитовую дизъюнкцию удобно использовать для противоположной цели: включения битов, поскольку, если у одного из операндов побитового OR в определенной позиции стоит единица, то и в числе, являющемся результатом операции, в этой позиции будет единица вне зависимости от значения второго операнда:

```
10101010 | 11010011 = 11111011
```

Побитовая операция XOR устанавливает значение бита в 1, когда два сравниваемых бита разные:

```
10111001 ^ 01111111 = 11000110
```

У операции XOR есть очень интересная функция: она обратима. Если вы используете операцию XOR для X и Y, а в результате получается Z, то когда операндами XOR станут Z и Y, результатом будет X! Этим свойством пользуются особенно часто при кодировании сообщений или в простых видах шифровки.

Операция NOT инвертирует свой операнд, то есть заменяет каждый нулевой бит на 1, а каждый единичный — на 0:

```
~11011110 = 00100001
```

Эта операция называется дополнением, потому что всегда `~x == (-x)-1`.

Следующие три оператора выполняют побитовый сдвиг своего первого операнда на количество разрядов, заданное вторым операндом. В отличие от логических операторов в операторах сдвига важен порядок операндов.

При сдвиге влево << освободившиеся справа позиции заполняются нулями. Беззнаковый сдвиг вправо >>>, называемый также логическим сдвигом, заполняет нулями позиции, освободившиеся слева. При знаковом сдвиге вправо >> слева распространяется значение старшего, знакового, бита: это так называемый арифметический сдвиг.

В результате любой операции сдвига теряется хотя бы один бит, и не существует способа его восстановить.

Оператор присваивания

Оператор присваивания уже многократно использовался в нашей книге. Поэтому мы не будем рассказывать о форме его записи, лишь скажем, что он имеет одно приятное свойство, о котором вы, возможно, еще не знаете. Он позволяет осуществлять операцию цепочечного присваивания. Благодаря этому можно присвоить одинаковое значение сразу нескольким переменным при помощи одной команды.

```
// объявляем три переменных
int a, b, c;
//присваиваем всем переменным значение 99
a = b = c = 99;
```

Оператор = присваивает значения справа налево: сначала переменная c получает значение 99, затем переменная b и, наконец, переменная a.

Кроме основного оператора присваивания язык Java располагает несколькими специальными (табл. 2.11), которые могут сделать вашу работу более приятной, а также сэкономить ваши усилия при написании исходного кода.

Специальные операторы присваивания *Таблица 2.11*

Оператор	Назначение
+=	прибавление значения правого операнда
-=	вычитание значения правого операнда
*=	умножение на значение правого операнда
/=	деление на значение правого операнда
%=	остаток от деления на значение правого операнда
&=	операция AND со значением правого операнда
|=	операция OR со значением правого операнда
^=	операция XOR со значением правого операнда

Операторы арифметико-логического присваивания вычисляют значение соответствующего арифметического или логического выражения над своими операндами и присваивают вычисленное значение первому операнду. Так, следующие команды эквивалентны:

```
x = x + 10;
```

и

```
x += 10;
```

Использование специальных операторов присваивания предпочтительнее записи выражения не только потому, что такая запись короче, но и потому, что виртуальная машина Java выполняет их более эффективно, чем последовательное вычисление выражения и присваивание. С этими операторами вы будете встречаться особенно часто в программах опытных разработчиков.

Тернарный оператор

Этот интересный оператор называется тернарным, потому что имеет три операнда:

```
выражение1 ? выражение2 : выражение3;
```

Тернарный оператор служит для организации ветвления программы, заменяя громоздкий оператор `if`. При этом `выражение1` является условием ветвления, оно должно возвращать значение типа `boolean`. Работает тернарный оператор следующим образом: если значение `выражения1` равно `true`, то вычисляется `выражение2` и оператор возвращает его значение; если `выражение1` равно `false`, то вычисляется и возвращается значение `выражения3`. Отсюда следует, что `выражение2` и `выражение3` должны возвращать одинаковый тип. Этот тип может быть любым.

С помощью тернарного оператора часто можно записать ветвление короче и изящнее, чем с помощью оператора `if`:

```
for (int number = -5; number <= 5; number++)
{
   int absval = number < 0 ? -number : number ;
   System.out.println("Абсолютное значение равно " + absval);
}
```

Приоритет операторов

В одном выражении может встретиться несколько операторов, и полезно знать, в каком порядке они будут выполняться. Порядок выполнения операторов определяется их приоритетом. Наивысший приоритет из уже рассмотренных операторов имеют скобки. Поэтому, если вы сомневаетесь, в каком порядке будут выполнены сложение и умножение в выражении 3 + 4 * 5, можете сами указать последовательность его вычисления при помощи скобок: ((3 + 4) * 5) = 35.

В таблице 2.12 перечислены группы операторов языка Java в порядке приоритета от наивысшего к наинизшему.

Операции с одинаковым приоритетом выполняются слева направо, за исключением операций присваивания, которые, как уже сказано, выполняются справа налево.

Приоритет операторов *Таблица 2.12*

()	]	.	
++	--	~	!
*	/	%	
+	-		
>>	>>>	<<	
>	>=	<	<=
==	!=		
&			
^			
|			
&&			
||			
?:			
=	op=		

2.8. Приведение типов

Явное и неявное приведение типов

Очень часто при программировании бывает необходимо присвоить переменной одного типа значение другого типа. Это легко сделать, если диапазон допустимых значений типа переменной, которой присваивается значение, больше диапазона типа присваиваемого значения. Например, легко можно присвоить значение типа `int` переменной типа `long` или `float`:

```
int i = 78;
long l = i; // присвоение значения int переменной типа long
float f = i; // присвоение значения int переменной типа float
```

Ход выполнения такой операции присваивания следующий: сначала присваиваемое значение приводится к нужному типу данных, после чего приведенное значение присваивается переменной этого типа. Такое, неявное, приведение типов без потерь в точности возможно для совместимых типов данных, если диапазон значений целевого типа шире, чем диапазон исходного типа.

При приведении числовых типов данных действует правило повышения, в соответствии с которым диапазон значений целевого типа должен покрывать диапазон типа присваиваемого значения. Это значит, что вы можете присвоить значение типа `long` переменной типа `double`, но обратное преобразование уже не допустимо.

Если нужно присвоить переменной значение несовместимого типа (или типа с более широким диапазоном), то здесь поможет явное приведение типов (type casting). Выполняется оно просто: достаточно перед оператором присваивания или другим оператором, требующим преобразования типов, указать в круглых скобках имя типа, в который вы хотите перевести значение переменной (листинг 2.14).

Листинг 2.14. Пример явного приведения типов

```
class ExplicitCast {

 public static void main(String[] args) {

   long l = 10;
   double d = l; // неявное приведение (long в double)
   l = (long) d; // явное приведение (double в long)

 } // main(String[])
} // ExplicitCast class
```

Если вы используете явное приведение типов для числовых переменных, будьте осторожны, чтобы не потерять данные. Потеря данных может произойти, когда вы преобразуете значение из широкого диапазона к типу, имеющему более узкий диапазон, то есть сужаете «жизненное пространство» переменной. Если бы переменная `d` типа `double` содержала значение, выходящее за рамки диапазона типа данных `long`, преобразование было бы выполнено с потерей данных без какого-либо предупреждения. Точно так же, если вы интерпретируете значение вещественного типа, например `float`, как тип `int`, вы обязательно теряете дробную часть числа.

Приведение типов в выражениях

Как вы уже поняли, переменные, операторы и литералы, соединенные вместе, составляют элементы программы, которые называются выражениями. Правилами языка Java определены допустимые способы сочетания элементов в выражениях. Эти правила мало отличаются от аналогичных им правил в других языках программирования, поэтому мы на них останавливаться не будем. Рассмотрим только неочевидные эффекты преобразования типов в выражениях.

Наличие переменных различных типов в одном выражении — обычное дело. Есть лишь одно условие: типы должны быть совместимы друг с другом. Если в выражении присутствует несколько типов данных, компилятор без предупреждения преобразует их к одному согласно правилу последовательного приведения типов.

Если в выражении присутствуют значения типов `char`, `byte` и `short`, они безоговорочно преобразуются в тип `int`. Если хотя бы один из операндов имеет тип `long`, то все выражение тоже преобразуется в `long`. При использовании хотя бы одного значения `float` значением выражения также будет являться `float`. То же самое и в случае использования `double` — результат будет иметь тип `double`. Значения переменных интерпретируются в качестве нового типа только на время вычисления выражения: объявленный тип переменных остается неизменным.

Пренебрежение этим правилом может привести к ошибкам в программе. Представьте себе, что вы умножаете друг на друга две переменные типа `byte`. Естественно было бы ожидать, что результат тоже будет иметь тип `byte`, однако это не так. Согласно вышеприведенному правилу, перед выполнением умножения оба операнда будут преобразованы к типу `int`, и значение выражения тоже будет иметь тип `int`. Следующий пример наглядно демонстрирует особенности приведения типов в выражениях.

Листинг 2.15. Пример приведения типов в выражениях

```
class GradualCastDemo {

 public static void main(String[] args) {

   byte x, z;
   int y;

   x = 5;
   y = x * x;   // все правильно, результат операции
                // имеет тип int
   z = (byte) (x * x);   // чтобы записать результат
                         // в переменную типа byte,
                       // требуется явное преобразование
                         // типа результата

 } // main(String[]) method

} // GradualCastDemo class
```

Обратите внимание, что в записи `z = (byte) (x * x)` выражение `x * x` взято в круглые скобки. Если бы скобок не было, то операция приведения типа `(byte)` относилась бы только к первому операнду и результат выражения имел бы, согласно правилу последовательного приведения, тип `int`.

2.9. Другие управляющие операторы

Из управляющих команд языка Java вы уже познакомились с условным оператором `if` и оператором цикла `for`. Наличие этих основных операторов в языке программирования позволяет записать любой алгоритм. Другие управляющие операторы позволяют записывать алгоритмы более удобно и наглядно. Для этих целей язык Java располагает следующими средствами: оператор варианта `switch`, операторы цикла `while` и `do... while` и команды перехода `break`, `continue` и `return`.

Варианты оператора if

Пока мы рассмотрели только основной вариант использования оператора ветвления `if`. Язык Java располагает еще несколькими вариантами. Первый из них — это ветвление программы на две части:

```
 if (x > 2) {
  System.out.println("x больше двух");
 } else {
  System.out.println("x меньше или равно двум");
}
```

Этот пример не требует долгих объяснений: если условие (x > 2) истинно, то будет выполнен первый блок команд, в противном случае — второй блок, записанный после ключевого слова `else`. Блок, состоящий только из одной команды, можно не заключать в фигурные скобки.

Дополнительное ветвление можно организовать, вкладывая в блоки новые команды `if`:

```
if (j < 100) {
 if (j < 90) a = b;
 if (k > j)
    c = k - j;
 else
    c = j - k;
} else {
 c = d;
}
```

Если нужно организовать ветвление на три и более блока, то конструкции `if...else` можно записывать в виде так называемой «лестницы»:

```
for (int x = 0; x < 4; x++) {
 if (x == 1)
    System.out.println("один");
 else if (x == 2)
    System.out.println("два");
 else if (x == 3)
    System.out.println("три");
 else
    System.out.println("ноль");
}
```

Результат выполнения этого фрагмента кода будет следующим:

```
ноль
один
два
три
```

Оператор варианта switch

В предыдущем примере был использован отдельный блок `else`, выполняемый тогда, когда выражение в условии принимает значение, не

предусмотренное в условиях остальных блоков `if`. Если задача позволяет предусмотреть все варианты значения условия, выгоднее использовать оператор варианта `switch`:

```
switch (выр_условие) {
 case конст_выражение1: блок1
 case конст_выражение2: блок2
 …
 default: блокDef
}
```

Выражение, образующее условие команды `switch`, может возвращать значение одного из типов `char`, `byte`, `short` или `int` — другие типы недопустимы. Выражения `конст_выражение1`, `конст_выражение2` и т. д. — это константные выражения, то есть такие, значение которых может быть вычислено на этапе компиляции программы. Значения выражений, использованных в одном операторе `switch`, должны быть различны.

Работает оператор выбора так: значение выражения `выр_условие` последовательно сравнивается со значениями `конст_выражение1`, `конст_выражение2` и т. д. Как только оно совпадет с одной из констант, выполняется соответствующий ей блок команд и все следующие за ним блоки.

Соответствующий блок может быть и пустым, тогда выполнение начнется с ближайшего следующего за ним непустого блока. Так можно запрограммировать одинаковую реакцию оператора `switch` на разные значения `выр_условие` (см. листинг 2.16).

Если вам требуется выполнить только один блок команд, то его нужно завершить командой `break`, прерывающей выполнение оператора варианта.

Если значение условия не совпало ни с одной из констант, выполняется блок, стоящий после ключевого слова `default`, поэтому ветвь `default` должна записываться последней. Вся ветвь `default` может отсутствовать, в таком случае при несовпадении значения `выр_условие` ни с одной из констант оператор варианта не делает ничего.

Листинг 2.16. Пример использования оператора варианта

```
class SwitchDemo {

 public static void main(String[] args) {

   for (int x = 1; x <= 13; x++) {
     switch (x) {
       case 1: case 2: case 12:
```

```
            System.out.println(x + ": зима");
            break;
          case 3: case 4: case 5:
            System.out.println(x + ": весна");
            break;
          case 6: case 7: case 8:
            System.out.println(x + ": лето");
            break;
          case 9: case 10: case 11:
            System.out.println(x + ": осень");
            break;
          default:
            System.out.println(x + ": нет такого месяца");
        } // switch
      } // for
    } // main(String[]) method
  } // SwitchDemo class
```

При каждом проходе цикла значение переменной `x` сравнивается с константами, перечисленными после ключевых слов `case`. Как только обнаружено совпадение, начинают выполняться команды, стоящие после найденной константы, вплоть до первой команды `break` или, если команд `break` нет, до конца оператора `switch`. Результат выполнения программы будет следующим:

```
1: зима
2: зима
3: весна
4: весна
5: весна
6: лето
7: лето
8: лето
9: осень
10: осень
11: осень
12: зима
13: нет такого месяца
```

Операторы `switch`, как и операторы `if`, могут быть вложенными, то есть среди команд блока `case` может быть еще один оператор `switch`.

Варианты использования цикла for

Цикл `for` — один из самых гибких операторов языка Java. До сих пор вы встречались только с простейшим его вариантом, в котором значение

переменной цикла увеличивается на 1 при каждом повторе цикла. Шаг цикла не обязательно должен быть равен единице. Вот пример, где переменная цикла уменьшается с шагом пять:

```
for (int x = 100; x > 0; x -= 5) {
 System.out.println("x равно " + x);
}
```

Значение условия проверяется перед каждым повтором цикла. Это значит, что вы можете написать синтаксически правильный оператор цикла, который не будет выполнен ни разу:

```
int z = 10;
for (int x = z; x < 5; x++) {
 System.out.println("Эта команда не выполнится ни разу");
}
```

Интересной возможностью является организация цикла `for` при помощи двух переменных цикла. При обычном способе записи цикл, в котором переменная `x` убывает от 10 до 1, а переменная y возрастает от 1 до 10, выглядит довольно громоздко:

```
int y = 1;
for (int x = 10; x > 0; x--) {
 System.out.println("x равно " + x + ", y равно " + y);
 y++; // увеличение значения второй переменной
} // for x
```

Язык Java позволяет записать тот же самый цикл короче и изящнее, используя обе переменные, `x` и `y`, как переменные цикла:

```
for (int x = 10, y = 1; x > 0; x--, y++) {
 System.out.println("x равно " + x + ", y равно " + y);
} // for x, y
```

Любую из конструкций, заключенных в круглые скобки, — команды инициализации, проверки условия или итерации — можно опускать. Вот пример, где команда увеличения переменной-счетчика цикла перемещена внутрь блока:

```
for (int count = 0; count < 10; ) {
 System.out.println("счетчик равен " + count);
 count ++;
}
```

Точно так же вы можете переместить внутрь блока `for` команды инициализации и проверку условия цикла. Можно опустить все три элемента в круглых скобках, оставив только точки с запятой: таким образом вы получите бесконечный цикл. Вам нужно самим позаботиться о том,

чтобы этот цикл не выполнялся бесконечно, поместив в блок команду выхода из цикла, которая будет выполнена при наступлении некоторого условия.

```
for (;;) {
 // программный код
 // где-то здесь должна быть команда выхода из цикла,
 // например break
} Цикл while
```

Оператор `while` является альтернативным способом организации цикла:

```
while (выр_условие) блок;
```

Выражение `выр_условие` должно возвращать логическое значение. Если это значение равно `true`, то выполняется `блок`, после чего значение выражения вычисляется и проверяется снова. Как только оно станет равно `false`, выполнение цикла `while` прекращается. Если значение `выр_условие` с самого начала равно `false`, то тело цикла не будет выполнено ни разу.

В следующем примере цикл `while` использован для вывода букв в алфавитном порядке.

Листинг 2.17. Пример цикла while

```
class AlphabetWhileDemo {

 public static void main(String[] args) {

   char ch = 'a';
   while (ch <= 'я') {
     System.out.println(ch);
     ch++;
   } // while
 } // main(String[]) method

} // AlphabetWhileDemo class
```

Цикл do...while

Оператор цикла `do...while` отличается от цикла `while` тем, что всегда выполняется хотя бы один раз благодаря тому, что условие проверяется в конце цикла. В следующем примере вывод букв в обратном алфавитном порядке происходит средствами цикла `do...while`.

Листинг 2.18. Пример цикла do...while

```
class AlphabetDoWhileDemo {

 public static void main(String[] args) {
   char ch = 'Я';
   do {
     System.out.println(ch);
     ch--;
   } while (ch >= 'A');

 } // main(String[])
} // AlphabetDoWhileDemo class
```

Выход из цикла по команде break

Команда `break` прерывает выполнение любого цикла — `for`, `while` и `do...while` — вне зависимости от значения условия. Встретив команду `break`, интерпретатор Java пропустит все следующие за ней в теле цикла команды, не будет проверять условие, если это цикл `do...while`, и перейдет к выполнению команды, следующей после цикла.

Листинг 2.19. Пример выхода из цикла по команде break

```
class BreakDemo {

 public static void main(String[] args) {

   int nmax = 25;
   for (int n = 0; n < nmax; n++) {

     /* цикл выполняется до тех пор, пока
        квадрат числа n не станет больше 25 */
     if ((n * n) >= nmax) break;
     System.out.println("n равно " + n +
                        ", n в квадрате равно " + n * n);
                        } // for
   System.out.println("Конец цикла");

 } // main(String[]) method
} // BreakDemo class
```

В результате программа выдает следующие строки:

```
n равно 0, n в квадрате равно 0
n равно 1, n в квадрате равно 1
n равно 2, n в квадрате равно 4
n равно 3, n в квадрате равно 9
n равно 4, n в квадрате равно 16
Конец цикла
```

Хотя в условии цикла `for` указано, что он должен быть выполнен `nmax` раз, выполнение цикла прервется значительно раньше, поскольку в теле цикла со значением `nmax` сравнивается не переменная цикла, а ее квадрат, и, как только это значение достигнуто, команда `break` выполняет выход из цикла.

Если вы используете команду `break` внутри вложенного цикла, помните, что она завершает выполнение только внутреннего цикла, а внешний цикл после нее продолжает выполняться. Команда `break`, завершающая выполнение ветви оператора `switch`, стоящего внутри цикла, прерывает только этот оператор `switch`.

Слишком частое использование команды `break` ухудшает читаемость программы. Старайтесь регулировать количество повторов цикла правильным подбором условия цикла, которое для этого и предназначено.

Выход из блока по команде break

В языке Java нет команды `goto`, которая во многих языках программирования служит для безусловной передачи управления. Наличие в программе этой команды противоречит принципам структурного программирования, поэтому из современного языка Java она исключена. Существуют, однако, ситуации, в которых команда, аналогичная `goto`, была бы очень полезна. Например, вам может понадобиться завершить выполнение нескольких вложенных циклов одной командой в теле самого внутреннего цикла.

Для решения этой задачи Java предлагает расширенную форму команды `break`:

```
break label;
```

Метка `label` — это идентификатор некоторого блока программы, внешнего по отношению к команде `break`. Если обычная команда `break` без метки прерывает выполнение непосредственно того блока, в котором находится (самого внутреннего), то `break` с меткой, ссылающейся на некоторый из объемлющих блоков, прерывает выполнение всех последовательно вложенных блоков, включая этот объемлющий:

```
lab_first:
{
 // блок, помеченный lab_first
 lab_second:
 {
    // блок с меткой lab_second
    break lab_first; // выход из блоков lab_second и lab_first
 } // конец блока lab_second
} // конец блока lab_first
```

Отдельный оператор может быть помечен так же, как и блок. Метка должна стоять перед блоком и отделяться от него двоеточием. Имена меток подчиняются правилам, действующим для любых других идентификаторов. Следующий пример демонстрирует применение расширенной команды `break`.

Листинг 2.20. Пример команды break с меткой

```
class ExtendedBreakDemo {

 public static void main(String[] args) {

   for (int i = 1; i < 4; i++) {

     Блок1: {
       Блок2: {
         Блок3: {

         System.out.println("\ni равно " + i);
         if (i == 1) break Блок1;
         if (i == 2) break Блок2;
         if (i == 3) break Блок3;

         // Эта команда никогда не будет выполнена.
         System.out.println("Эта строка никогда
                             не будет напечатана");

         } // конец блока Блок3
       System.out.println("завершен Блок3");
       } // конец блока Блок2
     System.out.println("завершен Блок2");
     } // конец блока Блок1
   System.out.println("завершен Блок1");

   } // for
```

```
  System.out.println("конец цикла for");

  } // main(String[])
} // ExtendedBreakDemo class
```

В результате запуска данной программы на экран будут выведены следующие строки:

```
i равно 1
завершен Блок1

i равно 2
завершен Блок2
завершен Блок1

i равно 3
завершен Блок3
завершен Блок2
завершен Блок1
конец цикла for
```

При использовании расширенной формы команды `break` обращайте особое внимание на ее положение во вложенных циклах, поскольку невнимательность может привести к тому, что программа поведет себя неожиданным для вас образом.

Также помните, что метка команды `break` не может относиться к текущему или внутреннему блоку, а только к внешнему. Это значит, что полностью аналогичной команде `goto` команда `break` не является и не позволяет программисту запутывать структуру программы. Следующий пример будет отвергнут компилятором:

```
// ОШИБКА. ЭТОТ ПРИМЕР НЕ КОМПИЛИРУЕТСЯ !
class ExtendedBreakErr {

 public static void main(String[] args) {

   Метка1: for (int x = 0; x < 3; x++) {
     System.out.println("Значение x равно " + x);
   }

   for (int y = 0; y < 50; y++) {
     if (y == 10) break Метка1;     // НЕЛЬЗЯ ссылаться
                                    // на посторонний блок
     if (y == 25) break Метка2;     // НЕЛЬЗЯ ссылаться
                                    // на внутренний блок
```

```
      System.out.println("y равно " + y);
      Метка2: {
        System.out.println("y равно 25");
      }
    }

  } // main(String[]) method
} // ExtendedBreakErr class
```

Прерывание одного повтора цикла по команде continue

Часто требуется не прерывать выполнение цикла, а перейти к следующей итерации, пропустив оставшиеся команды в теле цикла. Для этого служит команда `continue`, называемая еще «командой принудительного перехода к следующему повтору цикла». По команде `continue` интерпретатор заново вычисляет условие цикла и, если оно выполнено, продолжает выполнение тела цикла.

Следующий пример демонстрирует использование команды `continue` для вывода на печать только нечетных чисел.

Листинг 2.21. Пример команды continue

```
class OddNumDemo {

  public static void main(String[] args) {

    for (int num = 0; num <= 10; num++) {
      if ((num % 2) == 0) continue;
      System.out.println(num);
    }
  } // main(String[]) method
} // OddNumDemo class
```

У команды `continue`, как и у команды `break`, есть расширенная форма с меткой. В отличие от команды `break`, она передает управление не на конец помеченного блока, а на его начало.

ГЛАВА 3.

ВВЕДЕНИЕ В ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Предыдущая глава познакомила вас с основными конструкциями языка. Однако описание объектно-ориентированного языка Java было бы неполным, если бы в книге не было главы, вводящей в принципы объектного программирования. В этой главе мы рассмотрим основные понятия, относящиеся к классам и методам, создание объектов, передачу параметров и получение возвращаемых значений, использование конструкторов, команды `new` и ключевого слова `this`. В этой же главе мы введем понятие массива, потому что массивы в языке Java реализованы как простые объекты.

Далее в этой главе мы изучим более сложные вопросы, относящиеся к принципам объектного программирования: наследование, полиморфизм, виртуальные методы.

Завершают главу три важные темы. Первая — наследование между объектами и абстракции, позволяющие упорядочить ваши классы и придать им иерархическую структуру. Вторая тема — пакеты классов, которые дают возможность сортировать классы, объединяя их в логические единицы. Последняя большая тема — создание интерфейсов. Вся платформа Java построена на интерфейсах и их реализации, поэтому при изучении языка Java эта тема является ключевой.

3.1. Классы, методы класса, объекты

3.1.1. Что такое класс

Классы — это основа объектного подхода к программированию. Класс содержит как данные, так и программный код, манипулирующий этими данными. Этот код сосредоточен в методах класса. С одним из методов, а именно методом `main()`, вы уже познакомились. Сейчас пришло время расширить свои познания в области объектного подхода, чтобы иметь возможность писать более совершенные программы и, что самое главное, заложить основу, которая поможет вам понять материал последующих глав.

Класс можно представить себе как модель или чертеж, определяющие форму объекта, то есть описание данных, которые объект может хранить, и способов, которыми он может ими оперировать. Иными словами, класс — это спецификация типа объекта, а объект — конкретный экземпляр класса. Сам по себе класс — понятие абстрактное, никаких данных он не обрабатывает. Только создавая объекты на его основе, вы сможете работать с нужными вам данными.

Данные (переменные и константы), описанные в классе, называются членами класса, а процедуры — методами класса. Данные, хранящиеся в экземпляре класса, называются переменными объекта (instance variables).

Объектный подход — это просто

Лучший способ объяснить сущность классов и функциональность классов — это рассмотреть практический пример. Представьте себе, что ваша программа должна обрабатывать данные о различных автомобилях: легковых, грузовых, автобусах и т. п. При классическом способе программирования для каждого типа автомобилей пришлось бы писать отдельные процедуры, которые обрабатывали бы данные, специфические именно для этого типа. Кроме того, для обработки классическим способом информация должна быть представлена в виде, не приспособленном к человеческому восприятию.

Объектный подход избавляет от обоих этих неудобств. Чтобы представить данные об автомобилях в объектном виде, первым делом нужно указать свойства, отличающие автомобиль от всех остальных предметов в материальном мире, в том числе от других видов транспортных средств. Выберем в качестве основных характеристик число колес, количество перевозимых пассажиров, тип двигателя и вид потребляемого горючего. К числу признаков также можно отнести максимальную скорость и средний расход топлива.

Все указанные общие для понятия «автомобиль» атрибуты мы можем объединить под одним названием — автотранспортное средство. Опишем класс `Vehicle` (транспортное средство), содержащий эти атрибуты. Как оформляется класс, вы уже видели: ключевое слово `class`, за которым следует имя класса и тело класса, заключенное в фигурные скобки.

```
class Vehicle {
 int passengers; // количество пассажиров
 int wheels; // количество колес
 int maxspeed; // максимальная скорость
 int burnup; // расход топлива
}
```

Таким образом мы определили новый тип данных, представляющий собой набор переменных passengers, wheels, maxspeed и burnup. С объектно-ориентированной точки зрения каждый класс действительно является самостоятельным типом данных, а термины «класс» и «тип» взаимозаменяемы.

Для создания объекта (экземпляра) класса Vehicle служит команда new:

```
Vehicle car1 = new Vehicle();
```

В результате выполнения этой команды будет выделена область памяти, содержимое которой интерпретируется как новый объект car1, устроенный так, как это записано в объявлении класса Vehicle, то есть располагающий собственными экземплярами переменных passengers, wheels, maxspeed и burnup. Чтобы сослаться на одну из этих переменных объекта, нужно указать ее имя после имени объекта через точку. Например, так выглядит установка максимальной скорости автомобиля равной 130 км/ч:

```
car1.maxspeed = 130;
```

Следующий пример вычисляет, какое расстояние может проехать автомобиль за полчаса при движении с максимальной скоростью. Сохраните текст программы в файл VehicleDemo.java.

Листинг 3.1. Пример использования класса Vehicle

```
class Vehicle {
 int passengers; // количество пассажиров
 int wheels; // количество колес
 int maxspeed; // максимальная скорость
 int burnup; // расход топлива
} // Vehicle class

class VehicleDemo {

 public static void main(String[] args) {

   Vehicle car1 = new Vehicle();
   car1.passengers = 2; // два пассажира
   car1.wheels = 6; // шесть колес
   car1.maxspeed = 130; // max. скорость 130 км/ч
   car1.burnup = 30; // расход топлива 30 литров на 100 км

   // расчет пути, проходимого за полчаса
   // при движении с максимальной скоростью
   double distance = car1.maxspeed * 0.5;
```

```
        System.out.print("За полчаса car1 может проехать ");
        System.out.println(distance + ' км.');

        car1= null;
    } // main(String[])
} // VehicleDemo class
```

Для того, чтобы эту программу можно было запустить, исходный код должен храниться в файле `VehicleDemo.java`, потому что именно в классе `VehicleDemo` определен открытый (`public`) статический (`static`) метод `main()`, с которого начинается выполнение программы на языке Java. Класс `VehicleDemo` является поэтому главным классом программы. В ранних версиях JDK главный класс необходимо было объявлять как открытый (`public`), теперь это делать необязательно.

В дальнейших примерах мы не будем указывать, в каком файле следует сохранять исходный код: вышеприведенное правило определения главного класса программы позволит вам найти имя файла самостоятельно.

Компилятор превратит исходный код в два файла: `Vehicle.class` и `VehicleDemo.class`. Чтобы запустить программу, выполните команду:

```
Y:\>java VehicleDemo
За полчаса car1 может проехать 65.0 км.
```

У вас наверняка возник вопрос, можно ли создать несколько экземпляров класса `Vehicle` и к чему это приведет. Теоретически вы можете создавать неограниченное количество экземпляров одного и того же класса, однако на практике вы ограничены количеством доступной оперативной памяти. Каждый объект типа `Vehicle` имеет свои собственные переменные `passengers`, `wheels`, `maxspeed` и `burnup`, то есть у каждого экземпляра класса `Vehicle` эти переменные могут иметь свои значения. В следующем примере создаются два экземпляра класса `Vehicle` с различными значениями максимальной скорости и вычисляются пути, проходимые этими автомобилями за 1 час 15 минут.

Листинг 3.2. Пример использования нескольких экземпляров класса Vehicle

```
class Vehicle {
        int passengers; // количество пассажиров
        int wheels; // количество колес
        int maxspeed; // максимальная скорость
        int burnup; // расход топлива
} // Vehicle class
```

```
class MoreVehiclesDemo {

 public static void main(String[] args) {

   // объект car1
   Vehicle car1 = new Vehicle();
   car1.passengers = 2;
   car1.wheels = 6;
   car1.maxspeed = 130;
   car1.burnup = 30;

   // другой экземпляр класса Vehicle: объект bus1
   Vehicle bus1 = new Vehicle();
   bus1.passengers = 45;
   bus1.wheels = 4;
   bus1.maxspeed = 100;
   bus1.burnup = 45;

   // расчет пути, пройденного за 1.25 часа
   double interval = 1.25;
   double distanceCar = car1.maxspeed * interval;
   double distanceBus = bus1.maxspeed * interval;

   System.out.print("car1 может проехать
                     за 1 час 15 мин. расстояние в ");
   System.out.print(distanceCar + " км с " +
                    car1.passengers );
   System.out.println(" пассажирами.");
   System.out.print("bus1 может проехать
                     за 1 час 15 мин. расстояние в ");
   System.out.print(distanceBus + " км с " +
                    bus1.passengers );
   System.out.println(" пассажирами.");

 } // main(String[])
} // VehicleDemo class
```

Создание объектов

Рассмотрим синтаксис создания объектов из классов. В примерах мы использовали следующую синтаксическую конструкцию:

```
Vehicle car = new Vehicle();
```

Это выражение выполняет два действия. Во-первых, здесь объявляется переменная `car` типа `Vehicle`. Во-вторых, в памяти отводится место для

размещения экземпляра класса `Vehicle`, а переменной `car` присваивается адрес выделенной области памяти. Таким образом, в переменной car хранится не сам объект, а указатель или ссылка на него.

То же самое создание объекта можно было бы записать двумя командами:

```
Vehicle car; // объявляем переменную – указатель на объект
car = new Vehicle(); /* создаем объект и присваиваем
                        переменной его адрес */
```

В результате объявления переменной `car` она не получает никакого значения, то есть ни на какой объект еще не указывает. Объект физически создается в памяти командой `new`, возвращающей его адрес, который мы присваиваем переменной `car`, и только после этого она становится связанной с объектом.

Из того, что в переменной хранится не сам объект, а указатель на него, следует, что в программе может быть несколько переменных, ссылающихся на один и тот же объект. Продемонстрируем это на примере следующей программы.

Листинг 3.3. Пример использования указателей на объект

```
class SimpleVehicle { // упрощенный вариант класса Vehicle

 int passengers;
}

class RefTypes {

 public static void main(String[] args) {

 SimpleVehicle car1, car2;    // две ссылки на объект
                              // типа SimpleVehicle
 car1 = new SimpleVehicle(); // создаем объект и ссылку
                              // на него
 car1.passengers = 25; // задаем количество пассажиров

 //обе переменные ссылаются на один объект
 car2 = car1;

 // докажем это:
 System.out.println("Количество пассажиров car2 равно "
                    + car2.passengers);
 car2.passengers = 50;
 // если car2 и car1 - это один и тот же объект, то теперь
 // car1.passengers стало равно 50
```

```
    System.out.println("Количество пассажиров car1 равно "
                       + car1.passengers);
  } // main(String[])
} // RefTypes class
```

Отсюда видно, что переменные-объекты отличаются от переменных простых типов (`int`, `double` и т. п.). При присвоении значения одной переменной другой переменной того же простого типа значение переменной в правой части присваивания копируется в переменную, стоящую в его левой части. В случае переменных-объектов копируется адрес, хранящийся в переменной, стоящей в правой части присваивания, в результате чего обе переменные начинают ссылаться на одно и то же место в памяти. Такой тип данных называется ссылочным (reference) типом.

3.1.2. Методы класса

Объявление и вызов метода

До сих пор в наших примерах участвовали классы, содержащие только переменные. Использование классов лишь для группировки данных вполне корректно, но большинство классов все же содержит одну или несколько процедур, определяющих способы, которыми объект класса может работать с этими и другими данными. Эти процедуры называются методами.

Метод состоит из одной или нескольких команд. Рекомендуется планировать набор методов так, чтобы каждый из них решал только одну определенную задачу.

Метод имеет имя (идентификатор), под которым он известен в программе. Вы уже встречались с такими именами методов, как `main()` и `println()`. Имя `main()` закреплено за методом, с которого начинается выполнение программы. Напоминаем, что в качестве имени метода нельзя использовать ни одно из зарезервированных ключевых слов.

Метод может, но не обязан принимать параметры (называемые также аргументами) и возвращать значение. Чтобы объявить метод со списком параметров, нужно перечислить параметры с их типами через запятую в круглых скобках после имени метода. Если метод не имеет ни одного параметра, то круглые скобки остаются пустыми.

В качестве параметра можно передавать как переменную простого типа, так и объект. Областью действия параметра является все тело метода, и им можно манипулировать как локальной переменной.

В предыдущих примерах мы решали задачу расчета пути, пройденного за единицу времени, умножая время на значение переменной объекта

maxspeed. Очевидно, что эту задачу может решить сам объект типа Vehicle, если будет располагать соответствующим методом.

Добавим в класс Vehicle метод distance, рассчитывающий путь, проходимый с максимальной скоростью за заданное время. Время в часах передадим этому методу как аргумент.

Листинг 3.4. Пример использования метода

```
class Vehicle {
 int passengers;
 int wheels;
 int maxspeed;
 int burnup;

 // объявляем метод, вычисляющий пройденный путь.
 // метод принимает параметр interval, задающий время,
 // и не возвращает никакого значения (void)
 void distance(double interval) {
    double value = maxspeed * interval;
    System.out.println("пройдет путь, равный " + value
                       + " км.");
 } // distance(double interval)
} // Vehicle class

class VehicleMethodDemo {

 public static void main(String[] args) {

   Vehicle car = new Vehicle();
   car.passengers = 2;
   car.wheels = 4;
   car.maxspeed = 130;
   car.burnup = 30;

   // другой экземпляр класса Vehicle
   Vehicle bus = new Vehicle();
   bus.passengers = 45;
   bus.wheels = 4;
   bus.maxspeed = 100;
   bus.burnup = 25;

   // расчет пути, пройденного за 0.5 часа
   double time = 0.5;
```

```
        System.out.print("автомобиль с " + car.passengers +
                         " пассажирами ");
        car.distance(time);
        System.out.print("автобус с " + bus.passengers +
                         " пассажирами ");
        bus.distance(time);

    } // main(String[])
} // VehicleMethodDemo class
```

Сохраните исходный код в файл `VehicleMethodDemo.java`, скомпилируйте его, получив два файла `Vehicle.class` и `VehicleMethodDemo.class`, и запустите класс `VehicleMethodDemo`:

```
Y:\>java VehicleMethodDemo
автомобиль с 2 пассажирами пройдет путь, равный 65.0 км.
автобус с 45 пассажирами пройдет путь, равный 50.0 км.
```

Рассмотрим ключевые элементы вышеприведенного исходного кода, выделенные в листинге жирным шрифтом.

```
void distance(double interval) {
```

Это объявление метода с именем `distance()`, принимающего один параметр типа `double`, известный в теле метода под именем `interval`. Ключевое слово `void`, стоящее перед именем метода, указывает на то, что метод не возвращает никакого значения. Тело метода открывается левой фигурной скобкой.

Собственно тело метода `distance()` составляют две команды:

```
double value = maxspeed * interval;
System.out.println("пройдет путь, равный " + value + " км.");
```

Каждый экземпляр класса `Vehicle` содержит собственную копию переменной `maxspeed`, поэтому для каждого объекта типа `Vehicle` его метод `distance()` вычисляет путь, пройденный именно этим объектом. Результат вычисления мы выводим на экран прямо в теле метода, поэтому он может не возвращать значения.

Тело метода закрывается правой фигурной скобкой. После выполнения всех команд метода управление передается обратно к той части программы, откуда метод был вызван.

```
System.out.print("автомобиль с " + car.passengers
                 + " пассажирами ");
```

Здесь мы вызвали метод `print()` вместо привычного метода `println()`. Он отличается тем, что не переводит строку, то есть не добавляет к выводимой на печать строке символа конца строки. Мы сделали это для

того, чтобы текст, выводимый методом `distance()`, был напечатан в той же строке.

Команда

```
car.distance(time);
```

вызывает метод `distance()`, принадлежащий объекту `car`. Как и в случае переменных объекта, метод объекта идентифицируется именем объекта и именем метода, разделенными точкой. В круглых скобках после имени метода перечисляются переменные, значения которых передаются методу как параметры.

Обратите внимание, что в «формуле» расчета пути, состоящей в вызове метода `distance()`, участвует только время, но не скорость. Скоростью заведует объект, метод которого вызывается — это значение его внутренней переменной `maxspeed`. Таким образом, вам не требуется знать подробностей устройства объекта, чтобы заставить объект выполнить нужное вам действие.

Точно так же команда

```
bus.distance(time);
```

вычисляет путь, пройденный другим объектом — **bus**.

Выход из метода

Существует два способа завершить выполнение метода. Первый состоит в последовательном выполнении всех команд, составляющих тело метода, и достижении конца кода, отмеченного закрывающей правой фигурной скобкой. Другой способ — выполнение команды `return`. Встретив эту команду, интерпретатор Java пропускает все следующие за ней в теле метода команды и немедленно передает управление туда, откуда метод был вызван. Команда `return` может стоять в любом месте кода метода:

```
void countNums() {
 for (int i = 0; i < 5; i++) {
   if (i == 3) return;
   System.out.println(i);
 }
 System.out.println("Эта строка не будет напечатана");
} // countNums()
```

В одном методе может быть несколько команд `return`:

```
void twoReturns() {
 boolean flag;
 boolean error;
```

```
  /*
  где-то здесь переменные flag и error получают значения
  */
  if (flag) return;
  /* еще какие-то команды */
  if (error) return;
 } // twoReturns()
```

Метод может возвращать значение

Методы, не возвращающие в результате своей работы никакого значения — объявленные с ключевым словом `void` — встречаются не слишком часто. Гораздо полезнее методы, способные вернуть результат своей работы: итог выполненного вычисления, информацию об успешности проведенной операции, новое значение какой-либо из переменных объекта и т. п. Способность возвратить значение — одно из важнейших свойств метода.

При объявлении метода, возвращающего значение, вместо ключевого слова `void` нужно указывать тип этого значения. Это может быть один из простых типов или объект. Сейчас мы перепишем пример 3.4 так, чтобы метод `distance()` не печатал вычисленное значение пути, а передавал его в вызвавшую метод программу, которая может распорядиться этим значением каким угодно образом.

Листинг 3.5. Пример метода, возвращающего значение

```
class Vehicle {
 int passengers;
 int wheels;
 int maxspeed;
 int burnup;

 // объявляем метод, вычисляющий пройденный путь.
 // теперь он возвращает вычисленное значение типа double
 double distance(double interval) {
    double value = maxspeed * interval;
    return value;
 } // distance(double interval)
} // Vehicle class

class VehicleRetMethod {

 public static void main(String[] args) {
```

```
    Vehicle car = new Vehicle();
    car.passengers = 2;
    car.wheels = 4;
    car.maxspeed = 130;
    car.burnup = 30;

    // другой экземпляр класса Vehicle
    Vehicle bus = new Vehicle();
    bus.passengers = 45;
    bus.wheels = 4;
    bus.maxspeed = 100;
    bus.burnup = 25;

    // расчет пути, пройденного за 0.5 часа
    double time = 0.5;
    double distanceCar = car.distance(time);
    double distanceBus = bus.distance(time);

    System.out.print("автомобиль с " + car.passengers +
                     " пассажирами ");
    System.out.println("пройдет за полчаса путь " +
                       distanceCar + " км.");
    System.out.print("автобус с " + bus.passengers +
                     " пассажирами ");
    System.out.println("пройдет за полчаса путь " +
                       distanceBus + " км.");

  } // main(String[])
} // VehicleRetMethod class
```

Поскольку мы объявили, что метод `distance()` возвращает значение типа `double`, в команде `return` мы должны указать переменную или выражение именно этого типа:

```
double value = maxspeed * interval;
return value;
```

Если значение, которое вы собираетесь вернуть, не соответствует типу метода, вы должны явно привести его к нужному типу.

Если вы собираетесь использовать возвращенное значение только в одном выражении, но не дальше в программе, то не обязательно записывать его в переменную. Вызов метода может быть элементом выражения или аргументом другого метода. Вы могли бы, не объявляя дополнительной переменной **distanceCar**, записать команду вывода пути так:

```
System.out.println("пройдет за полчаса путь " +
                   car.distance(time) + " км.");
```

Метод `distance()` объекта `car` можно вызывать сколько угодно раз из любого места программы, входящего в область действия переменной `car` при условии, что объект, на который указывает переменная `car`, существует в этом месте. Можете, например, добавить в конец метода `main()` команду, сравнивающую результаты расчета пути для автомобиля и автобуса:

```
if (car.distance(time) > bus.distance(time))
 System.out.println("Разница " +
    (car.distance(time) - bus.distance(time)) +
    " км в пользу автомобиля");
```

Разумеется, с точки зрения производительности этот код не оптимален — метод `distance()` вызывается несколько раз для вычисления одного и того же значения — но он вполне корректен и к тому же нагляден. Так можно писать программы, если производительность для вас несущественна или вы уверены в том, что в теле метода не производятся сложные вычислительные операции.

3.1.3. Конструкторы

Что такое конструктор

В предыдущих примерах мы задавали значения внутренних переменных каждого объекта типа `Vehicle` «вручную», непосредственно в коде. Этот прием не самый распространенный в обычном программировании. Его недостаток состоит в высокой вероятности сделать ошибку, например, просто забыв присвоить значение одной из переменных.

Существует гораздо лучший способ инициализировать объект, то есть присвоить его переменным начальные значения. Этот способ состоит в использовании конструктора. **Конструктор — это специальный метод, который вызывается в момент создания объекта.** Имя этого метода совпадает с именем класса. Никакого значения он не возвращает и используется только для присвоения начальных значений переменным объекта.

В языке Java у каждого класса есть хотя бы один конструктор. Он может быть не объявлен явно: в этом случае компилятор добавляет к классу конструктор по умолчанию, присваивающий переменным объекта значения по умолчанию. Как только вы объявите для некоторого класса собственный конструктор, автоматически сгенерированный конструктор станет недоступен.

Листинг 3.6. Пример конструктора

```
class Ten {
 int x;
 // объявляя конструктор, мы закрываем доступ
 // к конструктору по умолчанию
 Ten() {
    x = 10;
 } // Ten() constructor
} // Ten class

class TenDemo {

 public static void main(String[] args) {

    Ten s1 = new Ten();
    Ten s2 = new Ten();

    if (s1.x == s2.x)
        System.out.println(s1.x + " = " + s2.x);
 } // main(String[]) method
} // TenDemo class
```

Жирным шрифтом выделено объявление конструктора класса `Ten`. Как видите, объявляется он точно так же, как обычный метод.

В методе `main()` мы создали два объекта типа `Ten`. В результате работы нового конструктора, заменившего конструктор по умолчанию, внутренняя переменная `x` каждого объекта получила начальное значение 10.

Как и обычному методу, конструктору можно передать параметры. В большинстве случаев конструктор с параметрами удобнее, чем без них, поскольку позволяет инициализировать переменные объекта из вызывающей программы. Обобщим класс `Ten`, член которого инициализируется числом 10, до класса `Number`, член которого может быть инициализирован любым числом.

Листинг 3.7. Пример конструктора с параметрами

```
class Number {
 int x;
 Number(int num) {
    x = num;
 } // Number(int) constructor
} // Number class
```

```
class NumberDemo {

 public static void main(String[] args) {

   Number t1 = new Number(9);
   Number t2 = new Number(27);

   System.out.println(t1.x + " *** " + t2.x);
 } // main(String[]) method
} // NumberDemo class
```

Создавая объекты `t1` и `t2`, мы вызываем конструктор `Number()`, передавая ему в качестве параметра то число, которое хотим присвоить их внутренним переменным `x` в качестве начального значения: 9 и 27 соответственно.

Теперь вы знаете достаточно, чтобы написать конструктор класса `Vehicle`, присваивающий начальные значения его членам `passengers`, `wheels`, `maxspeed` и `burnup`.

Листинг 3.8. Еще один пример конструктора с параметрами

```
class Vehicle {
 int passengers; //количество пассажиров
 int wheels; // число колес
 int maxspeed; // max. скорость
 int burnup; // расход топлива

 // конструктор с параметрами
 Vehicle(int p, int w, int ms, int bu) {
   passengers = p;
   wheels = w;
   maxspeed = ms;
   burnup = bu;
 } // Vehicle(int, int, int, int)

 // расчет длины пройденного пути
 double distance(double interval) {
   double value = maxspeed * interval;
   return value;
 } // distance(double)

} // Vehicle class

class VehicleConstrDemo {
```

```
  public static void main(String[] args) {

    Vehicle car = new Vehicle(2, 4, 130, 30);
    Vehicle bus = new Vehicle(45, 4, 120, 25);

    double interval = 1;
    double distanceCar = car.distance(interval);
    double distanceBus = bus.distance(interval);

    System.out.println("Автомобиль с " + car.passengers
                       + " пассажирами " +
  "проедет за 1 час " + distanceCar + " км.");
    System.out.println("Автобус с " + bus.passengers
                       + " пассажирами " +
  "проедет за 1 час " + distanceBus + " км.");
  } // main(String[]) method

} // VehicleConstrDemo class
```

Переменные обоих экземпляров класса `Vehicle` — `car` и `bus` — инициализируются в момент создания этих объектов значениями, переданными конструктору класса `Vehicle`.

Оператор new

Рассмотрим подробнее оператор `new`, который совместно с конструктором служит для создания экземляров любого класса. Обычно команда вызова этого оператора выглядит так:

```
class_var = new class_name();
```

Переменная `class_var` хранит указатель на создаваемый объект класса по имени `class_name`. Если конструктор класса `class_name` не определен программистом, оператор `new` вызывает конструктор по умолчанию.

Объем памяти в распоряжении виртуальной машины Java ограничен, поэтому может возникнуть ситуация, когда оператор `new` не сможет создать новый объект по причине нехватки памяти. В отличие от языка C/C++ при этом оператор `new` не вернет нулевой адрес, а будет сгенерировано исключение.

С целью упрощения примеров мы не учитываем возможность нехватки памяти в ходе выполнения программы. Однако при работе с реальными приложениями вы должны учитывать и это.

Ключевое слово this

Прежде чем завершить данную главу, необходимо упомянуть о существовании ключевого слова `this`. В рамках какого-либо метода с помощью этого ключевого слова можно сослаться на текущий экземпляр класса (объект). Чтобы лучше понимать, приведем пример для рассмотренного выше метода `distance()` класса `Vehicle`:

```
double distance(double interval) {
 double val = this.maxspeed * interval;
 return double;
}
```

Из примера видно, что ключевое слово `this` используется в качестве указателя на текущий экземпляр класса для доступа к его внутренней переменной `maxspeed`. Поскольку `this.maxspeed` записывается длиннее, чем просто `maxspeed`, вторая запись вам может показаться более удобной, однако у первой есть свои преимущества.

Ее нужно использовать, когда в теле метода или конструктора локальная переменная (или параметр) имеет то же имя, что и переменная — член класса. Например, мы можем переписать конструктор класса `Vehicle` так, чтобы имена его параметров совпадали с именами внутренних переменных класса `Vehicle`, а то такие имена параметров, как `p`, `w`, `ms` и `bu`, ни о чем не говорят.

Листинг 3.9. Пример использования ключевого слова this

```
class Vehicle {
 int passengers; // количество пассажиров
 int wheels; // число колес
 int maxspeed; // max. скорость
 int burnup; // расход топлива

 // конструктор
 Vehicle(int passengers, int wheels, int maxspeed,
         int burnup) {
   this.passengers = passengers;
   this.wheels = wheels;
   this.maxspeed = maxspeed;
   this.burnup = burnup;
 } // Vehicle конструктор
 // расчет пройденного пути
 double distance(double interval) {
   double value = this.maxspeed * interval;
   return value;
 } // distance(double) method
} // Vehicle class
```

3.2. Массивы

В отличие от многих других языков массивы в языке Java реализованы как объекты.

Массив можно представить себе как последовательность переменных одинакового типа, доступных по одному и тому же имени. Массивы предназначены для хранения некоторой категории однородных данных: например, прайс-листа, ежедневных температур воздуха и т. п. Существуют одномерные и многомерные массивы.

Главным преимуществом массивов является их способность объединять данные в логические ряды и давать доступ к отдельным элементам этих рядов при помощи индекса. Когда данные сгруппированы, с ними проще осуществлять некоторые операции: классифицировать их, подсчитывать средние величины, выделять максимальное и минимальное значения, рассчитывать сумму, стандартное отклонение и т. д. Более того, благодаря особенностям способов размещения данных массива в памяти, они идеально подходят для сохранения информации в файл.

Одномерные массивы

Одномерный массив — это ряд значений одного и того же типа, хранящихся в соседних ячейках памяти. Использовать его можно для хранения таких данных, как список имен компьютеров в локальной сети. Синтаксис объявления одномерного массива выглядит так:

```
Тип_массива имя_массива[] = new тип_массива[размер];
```

Квадратные скобки после `имя_массива` указывают на то, что объявляется не переменная простого типа, а именно массив. `Тип_массива` указывает тип значений, которые будут храниться в массиве. `Размер` указывает на количество элементов массива.

Поскольку массив является объектом, процесс его создания состоит из двух шагов. Сначала объявляется переменная с именем `имя_массива`, затем с помощью оператора `new` выделяется память, достаточная для размещения всех элементов массива, в соответствии с указанным в `Тип_массива` типом данных и количеством элементов, и адрес этой области памяти присваивается переменной `имя_массива`. Таким образом, переменная `имя_массива` всегда указывает на первый элемент массива.

Номер элемента в массиве называется индексом массива. Он всегда отсчитывается от нуля.

Многомерные массивы

Многомерный массив — это массив, элементами которого являются массивы. Самая простая форма многомерного массива — двумерный массив (таблица). Например, объявление таблицы 30 на 40 будет выглядеть так:

```
int mytable[][] = new mytable[30][40];
```

Как видите, в языке Java синтаксис многомерных массивов немного отличается от других языков программирования: индекс элемента многомерного массива состоит из нескольких индексов, каждый из которых заключен в собственные квадратные скобки.

Двумерный массив не обязан быть прямоугольной таблицей. Например, нам требуется хранить в массиве данные о количестве пассажиров каждого автобусного рейса за неделю, а автобус совершает шесть рейсов в день по будним дням и два рейса в день по выходным. Java позволяет при объявлении массива указывать только размер по первому измерению (количество строк таблицы), а память под эти строки отводить позже, по мере надобности. Таким образом, строки могут содержать разное количество элементов. Массив со строками разной длины называется произвольным:

```
int passengers[][] = new int[7][];
                // 7 строк таблицы по числу дней в неделе
passengers[0] = new int[6];  // понедельник
passengers[1] = new int[6];  // вторник
passengers[2] = new int[6];  // среда
passengers[3] = new int[6];  // четверг
passengers[4] = new int[6];  // пятница
passengers[5] = new int[2];  // суббота
passengers[6] = new int[2];  // воскресенье
```

У нас получился массив, в котором строки, описывающие рейсы по будним дням, имеют по 6 элементов, а по выходным — по 2 элемента.

Инициализация массива

Инициализация массива — это присвоение его элементам начальных значений. Для инициализации многомерного массива необходимо каждый инициализирующий список переменных заключить в фигурные скобки, отделенные друг от друга запятой. Следующий пример показывает инициализацию двумерного массива, хранящего число и его куб:

```
int power3[][] = {
 {1, 1},
 {2, 8},
```

```
  {3, 27},
  {4, 64},
  {5, 125}
};
```

Можно присваивать значение элементам массива и непосредственно, обращаясь к каждому элементу по его индексу/индексам. Вы могли бы инициализировать массив так:

```
int array[][] = new int[4][3];
int i, count;
//цикл для первой размерности массива
for(count = 0; count < 4; count ++){
 i = 0;
 //цикл для второй размерности массива
 while (i < 3){
    //присвоение значения элементам массива
    array[count][i]=i;
    System.out.print("array[" + count + "][" + i + "]="
                     + i + " ");
    i++;
 } // while (i <=3)
 System.out.println();
}// for
```

Этот фрагмент кода последовательно перебирает элементы массива с помощью двух циклов. В учебных целях мы реализовали циклы разными операторами: `for` и `while`

Помните, что нумерация индексов в любой размерности начинается с нуля, то есть для массива размером 7 x 8, объявленного следующим образом:

```
int p [][] = new int[7][8];
```

первым будет элемент `p[0][0]`, а последним — элемент `p[6][7]`.

3.3. Идем дальше...

3.3.1. Дальнейшие сведения о классах и методах

Этот раздел посвящен более сложным вопросам, касающимся классов и методов. Здесь вы узнаете, как управлять доступом к членам и методам объекта, как передавать методу объект в качестве параметра и как метод может возвращать объект. Вы научитесь использовать перегружаемые ме-

тоды и конструкторы и писать рекурсивные методы. Вы узнаете, для чего служит ключевое слово `static` и зачем нужны внутренние классы.

Ограничение доступа к данным

Как вы уже знаете, методы объекта манипулируют как с внешними данными, так и с внутренними переменными этого объекта, избавляя вызывающую эти методы программу от необходимости обращаться к внутренним переменным самостоятельно и вообще знать что-либо о внутреннем устройстве объекта. Изоляция данных внутри объекта — один из важнейших принципов объектно-ориентированного программирования. В идеале вызывающей программе должен быть запрещен непосредственный доступ к переменным объекта, чреватый неправильным обращением с этими переменными, вызывающим крах программы, и разрешен только «правильный» доступ — через специально предназначенные для этого методы.

Например, наш класс `Vehicle` написан так, что позволяет использовать объекты этого класса очевидно неправильным способом — скажем, присваивать их внутренним переменным бессмысленные отрицательные значения. Правильно было бы запретить присваивать им значения непосредственно, а предусмотреть метод, который проверял бы присваиваемые значения на корректность.

Доступ к членам и методам класса регулируется с помощью спецификаторов доступа.

Спецификаторы доступа

В языке Java существует четыре типа доступа к элементу класса (члену или методу), регулируемых тремя ключевыми словами:

- `private` (закрытый): доступ к элементу возможен только из класса, где этот элемент объявлен;
- `public` (открытый): доступ к элементу возможен откуда угодно;
- `protected` (защищенный): доступ к элементу возможен из подклассов того класса, где этот элемент объявлен, и из других классов того же пакета;
- дружественный: доступ к элементу возможен из любого класса в том пакете, где этот элемент декларируется. Этот тип доступа назначается по умолчанию, то есть тогда, когда не указан ни один спецификатор доступа.

При объявлении метода или переменной спецификатор доступа записывается первым, до всех остальных ключевых слов. До сих пор вы видели спецификатор доступа только в объявлении метода `main()`:

```
public static void main(String[] args) {
```

В описании спецификаторов мы упомянули слово пакет. Пакет — это набор классов, которые объединены для решения одной задачи или сгруппированы ввиду того, что служат схожим целям. Более подробная информация о пакетах будет приведена позднее, сейчас мы ограничимся простой констатацией факта их существования.

Поскольку защищенный доступ тесно связан с понятием наследования, о котором мы будем говорить позднее, мы рассмотрим только спецификаторы `public` и `private`.

Спецификатор `private` ограничивает доступ к члену класса рамками самого класса, исключая возможность его использования из другого класса. Допустим, в классе `Vehicle` мы хотим сделать максимальную скорость автомобиля недоступной для просмотра и изменения, чтобы ее можно было только задать раз и навсегда при создании объекта через параметр конструктора. Тогда следует объявить и использовать класс `Vehicle` так, как показано в листинге 3.10.

Листинг 3.10. Пример закрытого члена класса

```
class Vehicle {
 int passengers; // количество пассажиров
 int wheels; // число колес
 private int maxspeed; // max. скорость
 int burnup; // расход топлива

 // конструктор класса Vehicle
 Vehicle(int passengers, int wheels, int maxspeed,
         int burnup) {
    this.passengers = passengers;
    this.wheels = wheels;
    this.maxspeed = maxspeed;
    this.burnup = burnup;
 } // Vehicle(int, int, int, int) constructor

 // расчет пройденного пути
 double distance(double interval) {
    double val = this.maxspeed * interval;
    return val;
 } // distance(double) method

} // Vehicle class

class VehicleAccessDemo {
```

```
  public static void main(String[] args) {

    Vehicle ferrari = new Vehicle(2, 4, 360, 12);

    double distance = ferrari.distance(0.5);
    System.out.println("Ferrari за полчаса проедет "
                       + distance + " км.");

    // следующая команда вызовет ошибку компиляции.
    // Закомментируйте ее
    System.out.println("Скорость Ferrari: "
                       + ferrari.maxspeed + " км/ч");

  } // main(String[]) method

} // VehicleAccessDemo class
```

Как вы можете видеть из примера, указание спецификатора `private` для переменной `maxspeed` никак не повлияло на остальные элементы класса. Конструктор по-прежнему имеет возможность инициализировать переменную `maxspeed`, а метод `distance()` может использовать ее значение. Изменилось только одно: значение переменной `maxspeed` теперь нельзя ни прочитать, ни записать извне класса `Vehicle`.

Теперь для того, чтобы получить извне доступ к переменной `maxspeed`, требуется написать отдельный метод. Модифицируем пример, добавив в него метод чтения `maxspeed`, а также метод, не позволяющий при создании объекта-автомобиля задать для него некорректное количество колес. Корректным будем признавать число от 1 до 24.

Листинг 3.11. Пример методов чтения и записи закрытых членов класса

```
class Vehicle {
  int passengers; // количество пассажиров
  private int wheels; // число колес
  private int maxspeed; // max. скорость
  int burnup; // расход топлива

  // конструктор класса Vehicle
  Vehicle(int passengers, int wheels, int maxspeed,
          int burnup) {
    this.passengers = passengers;
    this.setWheels(wheels);
    this.maxspeed = maxspeed;
    this.burnup = burnup;
  } // Vehicle(int, int, int, int) constructor
```

```
  // расчет пройденного пути
  double distance(double interval) {
     double val = this.maxspeed * interval;
     return val;
  } // distance(double) method

  // метод чтения значения maxspeed
  int getMaxSpeed() {
     return this.maxspeed;
  }

  // метод чтения значения количества колес
  int getWheels() {
     return this.wheels;
  }

  // метод записи количества колес
  void setWheels(int wheels) {
     // проверяем переданный параметр на корректность
     if ((wheels < 1) || (wheels > 24)) {
        System.out.println("Неверно указано число колес.");
        return;
     }
     this.wheels = wheels;
  }

} // Vehicle class

class VehicleGetSetMethod {

 public static void main(String[] args) {

    Vehicle ferrari = new Vehicle(2, -2, 360, 12);
    System.out.println("Max скорость: "
                       + ferrari.getMaxSpeed() + " км/ч");
    System.out.println("Число колес: "
                       + ferrari.getWheels());

    ferrari.setWheels(4);
    System.out.println("Число колес (повторно): "
                       + ferrari.getWheels());

 } // main(String[]) method

} // VehicleGetSetMethod class
```

Мы определили метод `getMaxSpeed()`, возвращающий значение переменной `this.maxspeed`, и метод `getWheels()`, возвращающий `this.wheels`. Для установки числа колес с предварительной проверкой задаваемого значения на допустимость мы определили метод `setWheels()`, принимающий один параметр — число колес. Если его значение выходит за пределы допустимого диапазона, то на экран выводится предупреждающее сообщение, а метод прекращает работу, не присваивая неверного значения переменной объекта.

Заметьте, что метод `setWheels()` вызывается прямо из конструктора класса `Vehicle`, то есть мы исключили возможность задать неверное количество колес при инициализации объекта. Для проверки мы попробовали (в методе `main()`) вызвать конструктор `Vehicle` с запрещенным аргументом `-2`, и это привело к тому, что переменная объекта `wheels` получила значение по умолчанию 0:

```
Y:\>java VehicleGetSetMethod
Неверно указано число колес.
Max скорость: 360 км/ч
Число колес: 0
Число колес (повторно): 4
```

Вы познакомились еще с одним неписаным правилом программистов на Java, а именно с правилом именования методов, служащих для доступа к закрытым переменным класса. Имя метода чтения переменной должно состоять из имени этой переменной с префиксом `get`, а имя метода записи — с префиксом `set`. Коротко они называются `get`- и `set`-методами.

Два способа передачи параметров

Мы упомянули, что в качестве параметра методу можно передавать не только значение простого типа, но и объект. Сейчас мы покажем, что данные разного типа передаются разными способами.

Если аргумент метода имеет простой тип данных, то передаваемое значение будет скопировано в параметр, и в теле метода будет использоваться эта копия. В результате, даже если метод изменит переданное ему значение, после завершения работы метода исходная переменная останется без изменений. Такой способ передачи аргументов называется передачей по значению (by value).

Листинг 3.12. Пример передачи параметров по значению

```
class ParaByValue {

 void callByVal(int x, int y) {
    x = x + y;
    y = y + 1;
    System.out.println("x = " + x);
    System.out.println("y = " + y);
 } // callByVal(int, int) method

} // ParaByValue class

class ParaByValueDemo {

 public static void main(String[] args) {
    int a = 2;
    int b = 3;

    // передача параметров по значению
    ParaByValue test = new ParaByValue();
    test.callByVal(a, b);

    System.out.println("a = " + a); // вывод 2
    System.out.println("b = " + b); // вывод 3
 } // main(String[]) method

} // ParaByValueDemo class
```

Эта программа напечатает следующие строки:

```
x = 5
y = 4
a = 2
b = 3
```

Иначе обстоит дело, когда в качестве аргумента передается объект. Переменная объекта содержит указатель на область памяти, где размещается объект. Поэтому, когда вы передаете такую переменную в метод, внутри метода действует ссылка на объект и, если в ходе выполнения метода содержимое объекта (его переменные) претерпит изменения, то объект сохранит свое новое состояние и после завершения работы метода. Такой способ передачи параметра называется передачей по ссылке (by reference).

Листинг 3.13. Пример передачи параметра по ссылке

```
class ParaByRef {
 int x, y;

 ParaByRef(int x, int y) {
   this.x = x;
   this.y = y;
 } // конструктор ParaByRef

 // передача объекта (по ссылке)
 void callByRef(ParaByRef o) {
   o.x = o.x + this.x; // чтение переменной x
   o.y = o.y + this.y; // чтение переменной y
 } // callByRef(ParaByRef) method

} // ParaByRef class

class ParaByRefDemo {

 public static void main(String[] args) {
   ParaByRef p = new ParaByRef(2, 3); // первый объект
   ParaByRef q = new ParaByRef(3, 2); // второй объект

   System.out.println("q.x = " + q.x); // вывод 3
   System.out.println("q.y = " + q.y); // вывод 2

   p.callByRef(q);

   System.out.println("q.x = " + q.x); // вывод 5
   System.out.println("q.y = " + q.y); // вывод 5
 } // main(String[]) method
} // ParaByRefDemo class
```

В этом примере метод `callByRef()` класса `ParaByRef` принимает в качестве параметра объект типа `ParaByRef`, известный в теле метода под именем o. Метод прибавляет значение своих переменных `x` и `y` к значениям одноименных переменных переданного ему объекта и сохраняет сумму в переданном объекте. После завершения работы метода вызвавшей его программе становятся доступны новые значения внутренних переменных объекта, с которым работал метод `callByRef()`:

```
q.x = 3
q.y = 2
q.x = 5
q.y = 5
```

3.3.2. Перегружаемые методы

Что такое перегрузка методов

Из того, что вам уже известно об областях видимости имен, должно быть ясно, что в одном классе не может быть двух одинаковых методов, то есть методов с одинаковыми именами и наборами параметров. Набор из имени метода, количества параметров и их типов называется сигнатурой метода. Именно по сигнатуре различает методы компилятор. Таким образом, в классе могут быть определены несколько методов с одинаковым именем, отличающихся друг от друга количеством или типом параметров.

Использование методов с одинаковым именем, но разной сигнатурой называется перегрузкой методов (overloading), а сами методы — перегружаемыми.

Перегрузка методов удобна, когда требуется выполнять одну и ту же по смыслу операцию над данными различных типов. Например, вы можете написать процедуру учета оплаты товара по имени `pay()`, обрабатывающую любой вид оплаты: наличными, по кредитной карте, чеком и т. п. Компилятор сам определяет, какой из методов `pay()` следует вызвать, ориентируясь по количеству и типу параметров.

Помните, что тип возвращаемого значения не входит в сигнатуру метода, то есть перегружаемые методы не могут различаться только им.

Листинг 3.14. Пример перегружаемых методов

```
class Basket { // "корзина покупок"

 // оплата наличными
 void pay(double money) {
    System.out.println("Оплачено наличными: " + money);
 }

 // оплата кредитными картами
 void pay(String cardNum) {
    System.out.println("Оплачено по кредитной карте #" +
cardNum);
 }

 // оплата чеком
 void pay(String accountNum, String bankCode) {
    System.out.println("Переведено на счет #" + accountNum +
                       " в банке " + bankCode);
 }
```

```
} // Basket class

class BasketDemo {

 public static void main(String[] args) {

   Basket b1 = new Basket();
   Basket b2 = new Basket();
   Basket b3 = new Basket();

   System.out.print("b1: ");
   b1.pay(1200.0); // оплата наличными
   System.out.print("b2: ");
   b2.pay("123456789"); // оплата по карте
   System.out.print("b3: ");
   b3.pay("987654321", "5500"); // оплата переводом

} // main(String[]) method

} // BasketDemo class
```

Эта программа выведет на экран следующие строки:

```
b1: Оплачено наличными: 1200.0
b2: Оплачено по кредитной карте #123456789
b3: Переведено на счет #987654321 в банке 5500
```

Для объекта `b1` класса `Basket` компилятор видит вызов метода с аргументом типа `double` и определяет, что следует вызвать метод `pay(double money)`. Для объекта `b2` компилятор видит в качестве аргумента строку, поэтому вызывает метод `pay(String cardNum)`. Методу `b3.pay` передаются две строки, поэтому компилятор выбирает метод `pay(String accountNum, String bankCode)`.

Несмотря на то, что перегрузка методов — очень полезный прием, начинающим программистам рекомендуется не увлекаться ею, потому что неправильное использование этого приема является популярным источником ошибок. Если вам не удается избежать перегрузки методов, постарайтесь по крайней мере определить их так, чтобы методы различались не только типом, но и числом параметров. Тогда программисту, использующему ваш класс, будет ясно, какой из одноименных методов выберет компилятор. Если одинаковое количество параметров неизбежно, то обеспечьте хотя бы взаимную неприводимость типов параметров.

Правилами хорошего тона требуется, чтобы методы, носящие одинаковые имена, решали одинаковые или по крайней мере очень похожие задачи.

Иначе пользователь вашего класса ни за что не разберется в хаосе перегружаемых методов.

Перегружаемые конструкторы

Поскольку конструкторы — это не совсем обычные методы, поговорим об их перегрузке отдельно. Вспомните пример с конструктором по умолчанию класса `Vehicle`. Когда вы определили новый конструктор с четырьмя параметрами, это сделало невозможным использование старого конструктора без параметров, то есть создание объекта типа `Vehicle` без указания начальных значений. Если вы хотите, чтобы конструктор без параметров был доступен, его нужно объявить дополнительно.

Листинг 3.15. Пример перегружаемых конструкторов

```
class Vehicle {
 int passengers; // количество пассажиров
 private int wheels; // число колес
 private int maxspeed; // max. скорость
 int burnup; // расход топлива

 /* конструктор без параметров, инициализирующий
 ** переменные объекта стандартными значениями */
 Vehicle() {
    this.passengers = 4;
    this.wheels = 4;
    this.maxspeed = 160;
    this.burnup = 13;
 } // Vehicle() конструктор

 /* конструктор с параметрами, инициализирующий
 ** переменные объекта значениями, переданными из
вызывающей программы */
 Vehicle(int passengers, int wheels,
         int maxspeed, int burnup) {
    this.passengers = passengers;
    this.wheels = wheels;
    this.maxspeed = maxspeed;
    this.burnup = burnup;
 } // Vehicle(int, int, int, int) constructor

} // Vehicle class
```

Теперь вы можете создавать объекты класса `Vehicle` любым из способов:

```
Vehicle moskvitch= new Vehicle(); // типовой автомобиль
// автомобиль с особыми параметрами:
Vehicle ferrari = new Vehicle(2, 4, 360, 12);
```

Повторное использование кода в перегружаемых методах

При написании программ старайтесь как можно меньше дублировать код. В этом вам могут помочь перегружаемые конструкторы и методы. Например, как вы могли заметить, в приведенном выше примере в обоих конструкторах повторяется блок присвоения значений переменным. Если бы вы изменили название одной из переменных, пришлось бы вносить изменения в код в обоих конструкторах. Избежать этого можно, вызвав из общего конструктора (метода) частный конструктор (метод).

Листинг 3.16. Пример замены дублирующегося кода

```
class Vehicle {
 int passengers; // количество пассажиров
 private int wheels; // число колес
 private int maxspeed; // max. скорость
 int burnup; // расход топлива

 // конструктор без параметров
 Vehicle() {
   this(4, 4, 160, 13);
 } // Vehicle() конструктор

 // конструктор с параметрами
 Vehicle(int passengers, int wheels, int maxspeed,
         int burnup) {
   this.passengers = passengers;
   this.wheels = wheels;
   this.maxspeed = maxspeed;
   this.burnup = burnup;
 } // Vehicle(int, int, int, int) конструктор

 // методы расчета длины пройденного пути
 double distance(int interval) {
   return distance((double) interval);
 } // distance(int)
 double distance(double interval) {
   double value = this.maxspeed * interval;
   return value;
 } // distance(double)

} // Vehicle class
```

Как видите, конструктор без параметров вызывает другой конструктор — с параметрами, передавая ему типовые значения. Вызов осуществляется при помощи ключевого слова `this`, после которого в круглых скобках указываются параметры.

Когда вы вызываете перегруженный метод, количество и типы параметров определяют, какой именно из одноименных методов следует вызвать, поэтому преобразовывать типы параметров следует явно. Несмотря на то, что тип `int` совместим с типом `double` и неявно приводится к нему, вызов `distance(interval)` без явного приведения к типу `double` целого значения переменной `interval` привел бы к вызову того варианта метода `distance()`, который работает с целочисленным параметром, что привело бы к ошибке.

3.3.3. Рекурсия

Метод может вызывать сам себя. Такой вызов называется рекурсией, а метод, способный вызывать во время выполнения сам себя, называется рекурсивным. Рекурсивные вызовы образуют бесконечный цикл, поэтому в коде рекурсивного метода должно быть предусмотрено условие завершения цикла, то есть прекращения самовызовов и возврата управления в вызвавший экземпляр метода. Понятно, что вызывать новый экземпляр метода нужно не с теми параметрами, которые переданы текущему экземпляру, иначе цикл никогда не завершится.

Когда рекурсивный метод вызывает новый экземпляр себя, параметры и локальные переменные текущего экземпляра сохраняются на стеке; когда вызванный экземпляр возвращает управление, они извлекаются из стека. Таким образом, чем больше шагов самовызова делает метод, тем больше растет стек. Учитывайте это при написании рекурсивных алгоритмов (например, заменяющих цикл) и, главное, следите за количеством вложенных вызовов метода. Если оно слишком велико, стек может переполниться и интерпретатор Java выдаст сообщение об ошибке.

Рекурсия — простое и естественное решение для некоторых типов задач, например для просмотра папок на диске. Классическим примером использования рекурсии служит вычисление факториала числа. Без рекурсии было бы сложно организовать алгоритм быстрой сортировки Quicksort, который будет приведен далее. Несмотря на то, что рекурсивные методы обычно работают медленнее своих нерекурсивных альтернатив (это связано с расходованием системных ресурсов на повторные вызовы), в алгоритме Quicksort это замедление с запасом компенсируется эффективностью и позволяет очень быстро сортировать большие массивы данных.

Подведем итог: в рекурсивном методе обязательно должны быть команда самовызова, «счетчик повторов» и граничное условие, при выполнении которого повторные вызовы метода прекращаются.

3.3.4. Статические методы и члены класса

Спецификатор static

Спецификатор `static` вы уже много раз видели при объявлении метода `main()`. Этот спецификатор указывает, что метод или переменную можно использовать независимо от какого-либо объекта данного класса. Такой метод или переменная называются статическими. Метод `main()`, с которого начинается выполнение программы, обязан быть статическим, потому что на момент начала работы программы еще не создано ни одного экземпляра класса, содержащего этот метод.

Обычно метод связан со своим экземпляром класса и значения внутренних переменных, которыми оперирует метод, различны в зависимости от того, к какому экземпляру класса они относятся. Чтобы было возможно объявить метод статическим, он должен подчиняться следующим ограничениям:

- метод имеет доступ только к статическим переменным;
- метод может вызывать только статические методы;
- метод не может использовать ключевое слово `this`.

Эти ограничения не распространяются только на метод `main()` в силу его особого предназначения.

Чтобы вызвать статический метод, укажите имя класса (!), поставьте точку и далее укажите имя статического метода.

Рассмотрим пример с вычислением факториала. Здесь оба основных метода — `computeI()` и `computeR()` — можно спокойно объявить статическими, так как они удовлетворяют всем перечисленным выше правилам. В результате вы будете избавлены от необходимости создавать экземпляр класса `Factorial` для вызова функции расчета факториала.

Листинг 3.17. Пример статических методов

```
class Factorial {

 // вычисление факториала в цикле
 static long computeI(int n) {
   long result = 1;
   for (int i = 1; i <= n; i++)
     result *= i;
 return result;
}
 // вычисление факториала рекурсивным способом
 static long computeR(int n) {
```

```
    if (n == 1) return n;
    return computeR(n - 1) * n;
  }
} // Factorial class

class FactorialDemo {

 public static void main(String[] args) {

   // Factorial f = new Factorial();
   System.out.println("Вычисление факториала в цикле:");
   for (int i = 6; i > 0; i--) {
      System.out.println(i + "! = " + Factorial.computeI(i));
   }
   System.out.println("Вычисление факториала
                       рекурсивным методом:");
   for (int r = 1; r <= 6; r++) {
      System.out.println(r + "! = " + Factorial.computeR(r));
   }

 } // main(String[]) method
} // FactorialDemo class
```

Программа напечатает следующие результаты:

```
Y:\>java FactorialDemo
Вычисление факториала в цикле:
6! = 720
5! = 120
4! = 24
3! = 6
2! = 2
1! = 1
Вычисление факториала рекурсивным методом:
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
```

Статические блоки

Иногда требуется выполнить какую-либо инициализацию класса еще до того, как станет возможно физическое создание первого объекта этого класса. Например, такая инициализация может состоять в установке соединения с удаленным компьютером или базой данных; перед первым

вызовом статического метода класса может потребоваться инициализация статических переменных и т. п.

Для этой цели служат статические блоки. Так называются блоки команд, которые выполняются при объявлении класса, то есть намного раньше того момента, когда на основе класса будут физически созданы объекты.

Листинг 3.18. Пример статического блока

```
class StaticBlock {
 static double rootOf9; // корень из 9
 static double rootOf27; // корень из 27
 int number;

 static {
    System.out.println("Инициализация статических
                        членов класса....");
    rootOf9 = Math.sqrt(9.0);
    rootOf27 = Math.sqrt(27.0);
 } // static block

 StaticBlock(int number) {
    System.out.println("Инициализация переменных
                        экземпляра класса....");
    this.number = number;
 }
} // StaticBlock class

class StaticBlockDemo {
 public static void main(String[] args) {

    System.out.println("Корень из 9 равен "
                       + StaticBlock.rootOf9);
    System.out.println("Корень из 27 равен "
                        + StaticBlock.rootOf27);

    StaticBlock sb1 = new StaticBlock(4);
    System.out.println("sb1.number = " + sb1.number);

    StaticBlock sb2 = new StaticBlock(16);
    System.out.println("sb2.number = " + sb2.number);

 } // main(String[]) method
} // StaticBlockDemo class
```

В результате выполнения данной программы со статическим блоком на экран будут выведены строки:

```
Инициализация статических членов класса....
Корень из 9 равен 3.0
Корень из 27 равен 5.196152422706632
Инициализация переменных экземпляра класса....
sb1.number = 4
Инициализация переменных экземпляра класса....
sb2.number = 16
```

3.3.5. Вложенные и внутренние классы

Вложенные классы, попросту говоря, — это классы, которые объявлены внутри другого класса. Внешний класс ограничивает область действия вложенного, однако вложенные классы имеют доступ к методам и переменным внешнего. Существенное различие между вложенным и внешним классом состоит в том, что у вложенного класса есть доступ также и к закрытым (private) членам класса верхнего уровня, а внешний класс, напротив, не имеет доступа к закрытым методам и переменным вложенного класса.

Вложенные классы являются составной частью Java, начиная с версии JDK 1.1. Вложенные классы часто используются в качестве «хранилищ» внутренних вспомогательных процедур внешнего класса, например, для определения наибольшего и наименьшего из переданных значений.

Вложенные классы делятся на две основные категории: статические и т. н. «нестатические». «Нестатические» вложенные классы называются внутренними.

3.4. Принципы объектного программирования

К принципам объектно-ориентированного программирования относятся изоляция данных, наследование и полиморфизм. Изоляцию данных мы уже рассмотрели. Она сводится к тому, что как можно меньше данных о внутреннем устройстве класса должно быть известно снаружи класса.

Переменные класса должны быть скрыты от внешнего мира, а доступ к ним должен быть возможен только через `get-` и `set-`методы, предотвращающие некорректные способы их использования.

3.4.1. Наследование

Принцип наследования

Наследование — один из основополагающих принципов объектно-ориентированного программирования. Целью наследования является упорядочение классов в иерархическую древовидную структуру.

Образуется эта структура так: в основу ее кладется самый общий класс, описывающий только основные признаки, позволяющие отнести объекты к одной категории. Например, наш класс `Vehicle` характеризует типовое механическое транспортное средство, «транспортное средство вообще». Затем на основе этого базового класса определяются специализированные классы, описывающие конкретные разновидности данной категории: например, мы могли бы создать классы `Auto` и `Moto` для автомобилей и мотоциклов. Говорят, что эти классы наследуют базовому классу, являются его потомками.

Общий класс называется в языке Java надклассом (superclass) или базовым классом (base class). Наследующий класс называется подклассом (subclass). Таким образом, подкласс является уточненным, специализированным, вариантом надкласса.

Вообще в объектно-ориентированном программировании существует понятие множественного наследования, то есть подкласс может наследовать свойства двух и более классов. Однако язык Java такой возможности не поддерживает: в Java у каждого подкласса может быть только один надкласс.

Наследование можно представить себе как расширение базового класса дополнительными элементами и методами, поэтому для объявления данного класса наследником другого служит ключевое слово `extends` (англ. «расширяет»). Покажем, как определить новый класс «автомобиль» на основе имеющегося класса `Vehicle`.

Листинг 3.19. Пример наследования класса

```
// базовый класс Vehicle
class Vehicle {
 // возьмите тело класса из примера 3.11 и добавьте
 // конструктор без параметров из примера 3.15
} // Vehicle class

// подкласс (потомок) Auto базового класса Vehicle
class Auto extends Vehicle {
 boolean sunroof; // наличие люка
} // Auto class
```

```
public class ExtendsVehicleDemo {

 public static void main(String[] args) {

    // создание объекта подкласса Auto
    Auto bmw = new Auto();
    bmw.sunroof = true; // люк есть

    // пример обращения к методам и переменным
      // надкласса и подкласса
    System.out.println("Путь, пройденный за 1.5 часа: "
                       + bmw.distance(1.5) + " км.");
    System.out.println("Max.скорость: "
                       + bmw.getMaxSpeed() + " км/ч.");
    System.out.println("Наличие люка: " + bmw.sunroof);

 } // main(String[]) method

 } // ExtendsVehicleDemo class
```

Результат выполнения программы `ExtendsVehicleDemo` выглядит следующим образом:

```
Путь, пройденный за 1.5 часа: 240.0 км.
Max.скорость: 160 км/ч.
Наличие люка: true
```

Через объект подкласса `Auto` вызывающая программа имеет доступ ко всем доступным членам и методам надкласса `Vehicle`. Слово «доступным» означает здесь то, что наследование не дает права доступа к закрытым членам надкласса.

Конструкторы подкласса и ключевое слово super

Очевидно, что подкласс может иметь свой собственный конструктор. В таком случае сразу встает вопрос: кто несет ответственность за создание экземпляра подкласса? Может быть, конструктор надкласса? Нет. За формирование объекта подкласса всегда отвечают только конструкторы подкласса. Это следует из того, что конструктор надкласса не может ничего знать об устройстве класса-потомка. Поэтому конструкторы надкласса, в отличие от переменных надкласса, извне недоступны. Следующая команда вызовет ошибку компиляции:

```
Auto bmw = new Auto(2, 4, 320, 9); // ОШИБКА
```

Как же тогда создается объект класса `Auto`? Поскольку мы не определяли никаких собственных конструкторов, компилятор сам создал для класса `Auto` конструктор по умолчанию — без параметров. Этот конструктор

по умолчанию вызывает конструктор надкласса, также без параметров, а тот (пример 3.15) вызывает конструктор с параметрами, передавая ему стандартные значения. Поэтому переменные нашего объекта bmw, унаследованные от класса Vehicle, инициализируются значениями (4, 4, 160, 13).

Мы можем определить конструктор класса Auto, инициализирующий новую переменную, которой не было в классе-предке Vehicle:

```
class Auto extends Vehicle {
 boolean sunroof; // наличие люка

 Auto(boolean sunroof) {
    this.sunroof = sunroof;
 } // Auto(boolean) constructor

} // Auto class
```

Как теперь инициализировать унаследованные переменные? Если они открытые, то все в порядке: к ним можно обратиться из нового конструктора класса Auto. А что делать, если некоторые из этих переменных в классе-предке объявлены закрытыми и для них предусмотрен только метод чтения (в нашем случае это переменная maxspeed)?

В этом случае следует вызвать конструктор надкласса. Для этого предназначено ключевое слово super, которое используется так же, как уже знакомое вам ключевое слово this. Отличие только в том, что оно указывает, что нужно вызвать конструктор надкласса:

```
class Auto extends Vehicle {
 boolean sunroof; // наличие люка

 Auto(boolean sunroof) {
    super(4, 4, 200, 12);   // вызов конструктора
                            // надкласса Vehicle
    this.sunroof = sunroof;
 } // Auto(boolean) constructor
} // Auto class
```

Вызов конструктора надкласса должен быть первой командой в теле конструктора подкласса. Это правило нужно соблюдать всегда, иначе компилятор откажется обрабатывать программу. Ключевое слово super можно также использовать и для доступа к членам надкласса.

Мы можем добавить к классу Auto и конструктор с параметрами:

```
class Auto extends Vehicle {
 boolean sunroof; // наличие люка
```

```
  Auto(boolean sunroof) {
    this(4, 4, 200, 12, sunroof);
  } // Auto(boolean) constructor

  Auto(int passengers,
       int wheels,
       int maxspeed,
       int burnup,
       boolean sunroof) {
    super(passengers, wheels, maxspeed, burnup);
    this.sunroof = sunroof;
  } // Auto(int, int, int, int, boolean) constructor

} // Auto class
```

Таким образом мы избежали дублирования кода в конструкторе `Auto(boolean)`, вызвав из него новый конструктор класса `Auto` с пятью параметрами, а уже из него вызывая конструктор надкласса `Vehicle(int, int, int, int)` при помощи ключевого слова `super`.

Переопределение методов

Вполне естественно, что некоторые методы подкласса должны работать не в точности так же, как унаследованные методы надкласса. В таком случае достаточно в теле подкласса переопределить данный метод, то есть написать код этого метода, специфический для подкласса.

Добавим в класс `Vehicle` метод `toString()`, который будет возвращать строку, представляющую внутреннее состояние объекта `Vehicle`. Продемонстрируем, что этот же метод возвращает строку состояния объекта `Auto`, поскольку каждый объект типа `Auto` является в то же время объектом `Vehicle`.

Листинг 3.20. Пример вызова метода надкласса

```
class Vehicle {

  // тело класса из примера 3.19

  // новый метод
  public String toString() {
    return "Vehicle(passengers=" + passengers + ";" +
      "wheels=" + wheels + ";" +
      "maxspeed=" + maxspeed + ";" +
      "burnup=" + burnup +
      ")";
```

```
  } // toString() method
}   // class Vehicle

class Auto extends Vehicle {
 boolean sunroof; // наличие люка

 Auto(boolean sunroof) {
    this(4, 4, 200, 12, sunroof);
 } // Auto(boolean) constructor

 //конструктор подкласса Auto с 5 параметрами
 Auto(int f, int g, int h, int j, boolean sunroof) {
    super(f,g,h,j);
    this.sunroof = sunroof;
 } // Auto(int,int,int,int,boolean) constructor
}    // class Auto

class VehicleOverrideDemo {

 public static void main(String[] args) {

    Vehicle v = new Vehicle(2, 2, 100, 9);
    Auto a = new Auto(2, 4, 180, 12, true);
    System.out.println("Vehicle.toString(): " + v.toString());
    System.out.println("Auto.toString(): " + a.toString());

 } // main(String[]) method
} // VehicleOverrideDemo class
```

Вывод этой программы может вас немного разочаровать, потому что состояние объекта `Auto` представлено только теми переменными, которые роднят его с объектом `Vehicle`, и не отражает наличия люка:

```
Vehicle.toString(): Vehicle(passengers=2;wheels=2;
                    maxspeed=100;burnup=9)
Auto.toString(): Vehicle(passengers=2;wheels=4;
                 maxspeed=180;burnup=12)
```

Именно в таких случаях требуется переопределять метод надкласса. Напишем для класса `Auto` собственный метод `toString()`:

```
class Auto extends Vehicle {

 // остальная часть кода опущена

 // метод toString(), повторенный в классе Auto с изменениями
 public String toString() {
```

```
    return "Auto(passengers=" + passengers + ";" +
      "wheels=" + super.getWheels() + ";" +
      "maxspeed=" + super.getMaxSpeed() + ";" +
      "burnup=" + burnup + ";" +
      "sunroof=" + sunroof +
      ")";
  } // toString() method
} // Auto class
```

Добавьте этот метод в код примера 3.20, скомпилируйте и запустите. Теперь состояние обоих объектов выводится так, как требовалось, с учетом разницы их типов:

```
Y:\>java VehicleOverrideDemo
Vehicle.toString(): Vehicle(passengers=2;wheels=2;
                    maxspeed=100;burnup=9)
Auto.toString(): Auto(passengers=2;wheels=4;maxspeed=180;
                 burnup=12;sunroof=true)
```

Таким образом, если в классе-предке и классе-потомке определены два метода с одинаковыми именами, то для экземпляров каждого из этих классов вызывается метод, определенный именно в его классе; если же в подклассе нужного метода нет, то для экземпляра подкласса вызывается соответствующий метод надкласса, поскольку объект подкласса является разновидностью объектов надкласса.

3.4.2. Полиморфизм

Приведенный выше пример показывает, как используется переопределение методов, но не отражает настоящих возможностей этой техники. Нет ничего удивительного в том, что для объекта подкласса из одноименных методов вызывается именно тот, который определен в подклассе. Удивительно то, что переменная типа надкласса может указывать на экземпляр подкласса, и в этом случае интерпретатор Java выберет соответствующий метод динамически, во время выполнения программы. Таким образом, мы можем обращаться с объектами надкласса и подкласса единообразно, а реагировать на такое обращение они будут каждый по-своему. Такое поведение называется полиморфизмом.

Проиллюстрируем сказанное примером.

Листинг 3.21. Пример полиморфизма

```
// классы Vehicle и Auto возьмите из примера 3.20
class PolyVehicleDemo {
```

```
public static void main(String[] args) {

  Auto a = new Auto(true); // экземпляр подкласса Auto
  Vehicle v = new Vehicle(); // экземпляр класса Vehicle

  /*
  **  поместим оба объекта в массив типа Vehicle.
   **  Первый элемент массива
  **  pvd[0] будет содержать экземпляр подкласса Auto,
   **  а второй - экземпляр класса Vehicle.
  */
  Vehicle[] pvd = {a, v};

  for (int i = 0; i < pvd.length; i++) {
       // динамический выбор версии
       // переопределенного метода toString()
      System.out.println(pvd[i].toString());
  } // for


 } // main(String[]) method
} // PolyVehicleDemo class
```

В результате одна и та же команда

```
System.out.println(pvd[i].toString());
```

выведет две разные строки:

```
Auto(passengers=4;wheels=4;maxspeed=200;burnup=12;
     sunroof=true)
Vehicle(passengers=4;wheels=4;maxspeed=160;burnup=13)
```

3.4.3. Абстрактные классы и методы

По мере изучения особенностей наследования объектов в языке Java вам может понадобиться создать класс, характеризующий объект обобщенно, на базе которого вы будете впоследствии создавать подклассы. Подклассы будут уточнять свойства объектов, формируя таким образом иерархию классов.

Самый общий класс в данном случае будет абстрактным; он формирует «внешнюю оболочку» для подклассов, и только подклассы наполняют эту оболочку конкретным содержанием — кодом, реализующим задачи программы. Абстрактный класс, как правило, содержит один или несколько абстрактных методов. Абстрактный метод — метод с пустым телом,

в котором нет программного кода. Абстрактные методы не могут быть объявлены как статические.

Для объявления класса или метода абстрактным служит ключевое слово `abstract`. Если класс содержит хотя бы один абстрактный метод, сам класс обязательно должен быть объявлен абстрактным. Обратное неверно: абстрактный класс не обязан содержать ни одного абстрактного метода. Он может включать обычные реализованные методы.

Поскольку абстрактный класс реализован не полностью, невозможно создать экземпляр этого класса. Объектов абстрактного класса не бывает!

Класс-потомок абстрактного класса обязан либо переопределить все абстрактные методы надкласса, либо быть объявлен тоже абстрактным. Атрибут `abstract` наследуется до тех пор, пока все методы не будут переопределены.

Суть абстрактных классов и методов легко объяснить на примере расчета площади фигур различной геометрической формы. Создадим абстрактный класс под названием `Shape` с абстрактным методом `area()`, который должен вычислять площадь геометрической фигуры. Поскольку в классе `Shape` неизвестна форма этой фигуры и, следовательно, формула для расчета ее площади, метод `area()` должен быть абстрактным.

На основе класса `Shape` создадим три подкласса `Point`, `Triangle` и `Circle`, описывающих точку, прямоугольный треугольник и круг. Для этих фигур формулы вычисления площади известны, поэтому для каждого из этих надклассов мы можем реализовать метод `area()`.

Листинг 3.22. Пример абстрактного класса

```
abstract class Shape {
 abstract double area(); // метод area()
} // Shape class

class Point extends Shape {
 public String toString() {return "Точка";}
 double area() {return 0;} // площадь точки равна 0
} // Point class

class Triangle extends Shape {
 int cathetus1; // первый катет
 int cathetus2; // второй катет

 Triangle(int cathetus1, int cathetus2) {
   this.cathetus1 = cathetus1;
   this.cathetus2 = cathetus2;
```

```
  } // Triangle(int, int) constructor

  public String toString() {return "Треугольник";}

  double area() {
     return ((cathetus1 * cathetus2) / 2.0);
  } // area() method of Triangle class
} // Triangle class

class Circle extends Shape {
 int radius; // радиус

 Circle(int radius) {
    this.radius = radius;
 } // Circle(int) constructor

 public String toString() {return "Круг";}

 double area() {
    return ((radius * radius) * 3.14);
 } // area() method of Circle class
} // Circle class

public class ShapeDemo {

 public static void main(String[] args) {

    Point p = new Point();
    Triangle t = new Triangle(5, 3);
    Circle c = new Circle(9);
    Shape[] s = {p, t, c};

    System.out.println("Расчет площади фигур");
    for (int i = 0; i < s.length; i++) {
        System.out.println(s[i].toString() + " :"
                                  + s[i].area());
    } // for

 } // main(String[]) method

} // ShapeDemo class
```

Этот пример демонстрирует еще и полиморфизм: один и тот же вызов

```
System.out.println(s[i].toString() + " :" + s[i].area());
```

вызывает для элементов массива — объектов разного типа — разные методы `toString()` и разные методы `area()`. Вывод программы выглядит следующим образом:

```
Точка:0.0
Треугольник:7.5
Круг:254.34
```

3.4.4. Окончательные члены: ключевое слово final

Пользователи вашего класса имеют полное право расширять его, создавая собственные подклассы и переопределяя реализованные вами методы. Иногда это нежелательно и требуется защитить элементы программы от изменения их пользователем. Для этого служит объявление этих элементов окончательными с помощью ключевого слова `final`.

Значение окончательной переменной класса не может быть изменено не только извне класса, но и методами самого класса и любого его подкласса. Такая переменная может быть только инициализирована — либо при объявлении, либо в статическом блоке, либо в конструкторе класса. В сущности, окончательную переменную можно рассматривать как константу. Поэтому часто вместе со спецификатором `final` указывается спецификатор `static`: это дает возможность использовать константу без создания экземпляра класса.

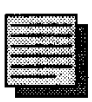
Окончательным может быть объявлен и класс. Для класса спецификатор `final` означает, что от этого класса нельзя наследовать, то есть он не может служить надклассом для другого класса.

В качестве примера напишем класс `MathConst`, предназначенный исключительно для хранения констант: числа «пи» и основания натурального логарифма. Понятно, что было бы жестоко предоставить пользователю возможность переопределить эти константы, а потом долго искать ошибку в его вычислениях.

```
final class MathConst {

 public static final double PI = 3.14159;
 public static final double E = 2.71828;

} // MathConst class
```



Примечание

Это упражнение — учебное, потому что в состав JDK входит готовый класс `Math`, содержащий константы PI и E, а также методы для выполнения распространенных математических операций.

Можно объявить окончательным и метод, в результате чего попытка переопределить этот метод вызовет ошибку компиляции:

```
class A {
 final void metoda() {
    System.out.println("Окончательный метод");
 } // metoda() method
} // A class

class B extends A {
 void metoda() { // ОШИБКА, т. к. метод надкласса окончательный
    System.out.println("Невозможно!");
 } // metoda() method
} // B class
```

Поскольку наследование — мощный инструмент языка Java, им часто злоупотребляют из-за того, что неправильно понимают его суть. Создавать класс на основе другого имеет смысл только тогда, когда описываемая новым классом сущность является разновидностью сущностей, описываемых базовым классом. Обращаясь к нашим примерам, наследование класса для описания велосипедов без мотора от класса `Vehicle` создаст больше проблем, чем решит.

3.5. Исключения

Исключения — чрезвычайно важный механизм языка Java, служащий для обработки ошибок в программах. Его понимание абсолютно необходимо при программировании на Java.

О возникновении исключения говорят, когда нормальная последовательность выполнения программы прерывается из-за того, что возникла нештатная ситуация: попытка деления на ноль, нехватка памяти, отсутствие нужного файла на диске, обращение к элементу массива по индексу, выходящему за пределы массива, и т. п.

Исключения реализованы в языке Java как классы, и к ним относятся все объектные свойства языка. В своей программе вы можете определить собственные классы исключений на основе стандартных.

При возникновении нештатной ситуации интерпретатор Java создает объект класса «исключение». Этот объект содержит описание выявленной ошибки и позволяет изменить последовательность выполнения программы с целью ликвидировать или уменьшить ущерб от этой ошибки.

В Java существует три типа состояний, связанных с некорректной работой программы, которые могут на первом этапе вас немного запутать:

- проверяемые исключения (базовый класс `Exception`);
- непроверяемые исключения в ходе исполнения программы (подкласс `RuntimeException` класса `Exception`);
- ошибки (базовый класс `Error`).

Проверяемые исключения описывают ситуации, которые разработчик метода может предвидеть. В других языках программирования разработчик придумывает для таких ситуаций набор кодов ошибки, и при возникновении одной из предусмотренных ошибок метод возвращает соответствующий ей код, который должна распознавать и обрабатывать вызывающая программа. Программист на Java в таких случаях создает объект исключения и «выбрасывает» его по команде `throw`. Выполнение метода на этом прекращается, и вызвавшая его программа обрабатывает исключение.

Обработка исключения начинается с его перехвата. Чтобы исключение, порожденное при вызове метода, можно было перехватить, вызов помещается в блок `try...catch` после ключевого слова `try`. Если в ходе работы метода возникло исключение, управление передается на команду, стоящую после ключевого слова `catch`. При этом метод должен быть объявлен как порождающий исключения с помощью ключевого слова `throws`, после которого через запятую перечисляются классы исключений, для которых предусмотрена внешняя процедура обработки.

Листинг 3.23. Пример проверяемого исключения

```
class DivisionByZeroException extends Exception {

 public String getMessage() {
    return "Деление на ноль запрещено!";
 }
} // DivisionByZeroException class

class ExceptionDemo {

 private static double divide (double dividend,
      double divisor)
    throws DivisionByZeroException {

    if (divisor == 0)
      throw new DivisionByZeroException();
    return dividend / divisor;
 } // divide() method

 public static void main(String[] args) {
```

```
        try {
          divide (8, 0);
        } catch (DivisionByZeroException dz) {
          System.out.println(dz.getMessage());
        }
    } // main(String[]) method

} // ExceptionDemo class
```

Предвидя, что наш метод `divide()` могут вызвать с такими аргументами, которые вызовут попытку деления на ноль, мы первым делом объявляем класс `DivisionByZeroException` для обработки такой ситуации, наследуя его от стандартного класса `Exception` из пакета JDK. Мы расширяем стандартный класс единственным методом `getMessage()`, возвращающим описание исключения.

При объявлении метода `divide()` мы указываем, что он может выбросить исключение типа `DivisionByZeroException`. В теле метода мы проверяем условие исключения и, если оно выполнено, создаем и выбрасываем экземпляр класса исключения:

```
throw new DivisionByZeroException();
```

В противном случае выполнение метода продолжается и он возвращает частное от деления своих параметров по команде `return`.

В методе `main()` находится блок перехвата исключения `DivisionByZeroException`. Он начинается с ключевого слова `try`, после которого следует вызов метода. После ключевого слова `catch` стоят команды обработки перехваченного исключения. Перехват состоит в том, что объект исключения, созданный в методе `divide()`, доступен в блоке `catch` и мы можем вызывать его методы. Нас интересует только метод `getMessage()`, выдающий словесное описание исключения.

Попробуйте вызвать метод `divide()` вне блока `try...catch`. Вы получите ошибку компиляции «unreported exception DivisionByZeroException; must be caught or declared to be thrown». Компилятор следит, чтобы все проверяемые исключения были перехвачены, то есть либо метод, порождающий исключение, должен вызываться в блоке `try`, либо сам вызывающий метод должен быть объявлен как порождающий исключение (ключевое слово `throws`).

Исключения типа `RuntimeException` и потомков этого класса относятся к непроверяемым. Они возникают в ходе исполнения программы и свидетельствуют об ошибках в логике программы.

Исключения типа `Error` и потомков этого класса спецификацией языка Java зарезервированы для нештатных ситуаций, вызванных недостатком

системных ресурсов, сбоем виртуальной машины Java и тому подобными причинами.

Примечание

Благодаря объектным свойствам языка Java вы имеете право определять для исключений собственные классы, которые не являются потомками ни одного из стандартных классов исключений (`Exception`, `RuntimeException` и `Error`). Никаких преимуществ перед стандартными такие классы не имеют, приводят только к путанице в программе, и мы настоятельно рекомендуем никогда не пользоваться этим правом.

Не всегда легко определить, следует ли при разработке метода пользоваться механизмом исключений. Как правило, метод должен выбрасывать исключение, когда в ходе его выполнения возникла нештатная ситуация, для обработки которой средств самого метода недостаточно. Например, метод вызван с неправильными параметрами: для этого случая Java предусматривает стандартные классы `NullPointerException`, `IllegalArgumentException` и другие.

Если в ходе работы вашего метода могут возникнуть условия, при которых его дальнейшее выполнение невозможно или бессмысленно, рекомендуется выбрасывать исключение, выбрав для него подходящий стандартный класс или определив собственный класс на основе класса `Exception`.

У вас может возникнуть соблазн написать программу, управляемую ошибками, то есть намеренно вызывать методы так, чтобы произошла ошибка и в результате ее обработки управление получила определенная ветвь (блок) программы. Этого делать категорически не рекомендуется.

Если метод может породить несколько исключений, то в вызывающей программе для каждого из них должен быть написан отдельный блок `catch`.

Перехваченные исключения можно не только обрабатывать, но и игнорировать. В этом случае блок `catch` должен быть пустым. Вот пример перехвата и игнорирования трех исключений:

```
try {
 // вызов метода, порождающего любое из трех исключений
} catch (IllegalArgumentException iae) {
} catch (IllegalStateException ise) {
} catch (NullPointerException npe) {}
```

В выбрасывании исключений есть свое преимущество, так как вы даете пользователю возможность обработать их. С другой стороны, такой подход имеет и недостаток: вы не только даете возможность, но и принуждаете к этому. Если метод или класс будет порождать много исключений, пользователи в конце концов откажутся от его использования.

В таблице 3.1 приведены чаще всего используемые стандартные классы исключений и типичные ситуации, для которых они предназначены.

Стандартные классы исключений *Таблица 3.1*

Класс	Назначение
IllegalArgumentException	Неверный аргумент
IllegalStateException	Состояние объекта не позволяет вызвать метод
NullPointerException	Параметр-указатель имеет значение null
IndexOutOfBoundsException	Значение индекса выходит за пределы диапазона массива
ConcurrentModificationException	Обнаружено параллельное изменение объекта
UnsupportedOperationException	Объект не поддерживает метод

Только тогда, когда ни одно из стандартных исключений не отвечает в полной мере вашим требованиям, имеет смысл создавать собственное исключение на основе класса `Exception` или `RuntimeException`.

Если вы сомневаетесь и не знаете, на основе какого класса создавать исключение, обратитесь к Java API, где вы найдете достаточно много примеров, которые помогут при выборе решения. При выборе имени для подкласса-исключения руководствуйтесь соглашениями об именовании (Code Conventions): имя исключения должно состоять из краткого описания ситуации, для которой оно предназначено (см. табл. 3.1), за которым следует слово `Exception`.

3.6. Пакеты классов

В этой части мы разберем все оставшиеся темы, которые касаются объектно-ориентированного программирования на языке Java. Они помогут вам гораздо лучше контролировать исходный код, придать ему логичную структуру с помощью специальных средств — пакетов (packages) и интерфейсов (interfaces).

Пакеты служат для упорядочивания классов по назначению в группы и подгруппы. Основное назначение пакетов состоит в том, чтобы облегчить понимание задач отдельных классов и управление ими.

Пакеты

По мере того как ваша программа будет разрастаться, расширяя свои функции, возникнет необходимость сгруппировать разрозненные классы в связанные логические единицы, чтобы как-то упорядочить и структурировать программу. Для этих целей Java предлагает использовать пакеты классов, образующие иерархическую древовидную структуру.

Назначение пакетов классов не ограничивается только группировкой классов. С помощью пакетов вы также сможете управлять доступом к отдельным классам, методам и переменным членов классов. Например, в программе не может быть двух классов с одинаковым именем. Если же класс содержится в пакете, то имя класса состоит из имени пакета, за которым следует собственное имя класса:

```
пакет1[.пакет2[.пакет3. ... ]].класс
```

Такое уточненное имя называется полным именем класса (fully qualified name). Имена классов, интерфейсов и подпакетов должны быть уникальны в пределах пакета. Говорят, что пакет (без подпакетов) образует отдельное пространство имен. Таким образом, вы можете использовать в программе несколько одноименных классов при условии, что они находятся в разных пакетах.

Любой класс относится к какому-либо пакету. Если вы не указали, в какой пакет должен быть помещен класс, то компилятор помещает его в пакет по умолчанию — безымянный (unnamed package).

Чтобы указать компилятору, в какой пакет записать скомпилированный класс, в самом начале файла исходного кода, до объявления класса, напишите строку:

```
package имя_пакета;
```

Если компилятор не найдет пакета с указанным именем, он его создаст. В именах пакетов, как и в прочих идентификаторах, Java различает регистр символов.

Скомпилированный класс представляет собой файл байт-кода с расширением `class`. Все такие файлы, принадлежащие одному пакету, хранятся в одной папке, имя которой совпадает с именем пакета. Подпакеты соответствуют ее подпапкам. Так иерархия пакетов отображается на структуру файловой системы.

Например, если вы объявите классы, описывающие обслуживание двигателя нашего автомобиля `Vehicle`, принадлежащими пакету

```
package Automoto.engine.util;
```

то компилятор включит эти классы в пакет `Automoto.engine.util` и поместит их файлы в папку `Automoto/engine/util`.

Откуда отсчитывается относительный путь `Automoto/engine/util`? Первой интерпретатор Java просматривает в поисках пакетов текущую папку. Далее просматриваются папки, перечисленные через точку с запятой в системной переменной CLASSPATH. Чтобы просмотреть или изменить значения системных переменных под Windows XP, откройте

окно свойств системы (**Пуск→Панель управления→Система**), перейдите на вкладку **Дополнительно** и нажмите кнопку **Переменные среды**.

Примечание

К сожалению, значение системной переменной CLASSPATH меняется с каждым новым JDK. Текущую информацию о переменной CLASSPATH вы найдете на сайте Sun: http://java.sun.com.

Вы можете указать корневую папку пакетов и при запуске интерпретатора Java **java.exe** с помощью ключа `-cp` или `--classpath`:

```
java -cp %CLASSPATH%;c:\project\Automoto VehicleDemo
```

Как пакеты влияют на доступность классов, их членов и методов

Ранее мы рассмотрели управление доступом к переменным и методам класса с помощью спецификаторов `public` и `private`. Нахождение класса в том или ином пакете тоже влияет на доступ к его членам и ему самому. Правила следующие:

- Если для переменной, метода или класса (далее называемых элементами языка) не указан ни один спецификатор доступа, то доступ по умолчанию пакетный: этот элемент доступен из класса, в котором он содержится, и всех классов того же пакета, включая все подклассы классов этого пакета.
- Элементы со спецификатором `public` доступны отовсюду: из всех классов.
- На закрытые (`private`) элементы членство класса в пакете не влияет: доступ к ним возможен только из того класса, в котором эти элементы определены.
- Элементы со спецификатором `protected` доступны из подклассов того класса, в котором эти элементы определены, а кроме того, из всех классов того же пакета.

Импорт пакетов

Чтобы использовать в программе класс из какого-либо внешнего пакета, вы можете обратиться к классу по его полному имени:

```
java.io.File f = new java.io.File("myfile.txt");
```

Этот способ записи из-за своей длины не слишком удобен, особенно если нужный класс находится достаточно глубоко в иерархии пакетов. Чтобы можно было обращаться к классу по его собственному имени, включите в программу пакет, содержащий этот класс, с помощью ключевого слова `import`:

```
import java.io.*;
File f = new File("myfile.txt");
```

Звездочка означает импорт всех классов из указанного пакета. Можно было бы импортировать классы и по одному, но количество импортированных классов/пакетов никак не влияет на скорость выполнения программы, поэтому удобнее указывать сразу весь пакет. Количество импортированных классов увеличивает только затраты времени на компиляцию программы.

Основные пакеты Java

В состав JDK входят несколько пакетов, классы из которых нужны чаще всего. Эти пакеты составляют единое целое и называются Java API (Application Programming Interface — интерфейс прикладного программирования). Пакеты Java API организованы как подпакеты основного пакета `java`.

Пакеты Java API *Таблица 3.2*

Пакет	Назначение
java.lang	Классы общего назначения, составляющие основу языка
java.io	Классы, обслуживающие операции ввода/вывода
java.net	Классы, поддерживающие работу в сети и интернете
java.applet	Классы для разработки апплетов — программ, выполняемых интернет-браузером
java.awt	Классы для разработки графического интерфейса приложений
java.util	Вспомогательные классы для работы со структурами данных, с функциями времени и случайными числами
java.text	Вспомогательные классы для обработки и оформления текста
java.math	Классы, реализующие математические функции с плавающей точкой, поддержку больших чисел и др.
java.beans	Классы поддержки компонентов архитектуры JavaBeans
java.sql	Классы для доступа к данным в реляционных базах данных через JDBC
java.rmi	Классы поддержки распределенного программирования
java.security	Классы поддержки криптографических операций

До сих пор мы использовали только классы из пакета `java.lang`, который не требует импорта: все программы на языке Java получают доступ к его классам автоматически. Из классов пакета `java.lang` мы использовали класс `System` с методами `print()` и `println()`. Другие классы из пакетов Java API будут подробно рассматриваться далее.

Существует еще один комплексный пакет, с которым вам необходимо ознакомиться — `javax`, в который включены классы, расширяющие функции всей платформы Java. Некоторые из них описаны в табл. 3.3.

Классы пакета `javax` *Таблица 3.3*

Пакет	Назначение
javax.swing	Поддержка графического интерфейса пользователя Swing
javax.sql	Дополнительные классы для доступа к реляционным базам данных
org.w3c.dom	Классы для работы с XML-документами при помощи DOM
org.xml.sax	Классы для работы с XML-документами через SAX
javax.xml.transform	Поддержка XSLT-операций над XML-документами
javax.xml.parsers	XML-парсеры
javax.transaction	Поддержка распределенных транзакций
javax.sound	Поддержка работы со звуком
javax.accessibility	Поддержка управления GUI
javax.naming	Поддержка операций JNDI
javax.rmi	Дополнение для распределенного программирования

3.7. Интерфейсы

Ранее мы рассматривали абстрактные методы и классы и говорили, что абстрактные методы содержат только сигнатуру метода (то есть его имя, число и тип параметров), но не содержат его реализации (под реализацией понимается программный код метода, то есть набор команд, обеспечивающий выполнение методом всех его функций). Все методы абстрактного класса реализованы в его подклассах. Таким образом, все подклассы заключают со своим надклассом «контракт», по которому обязуются реализовать каждый из методов. Такой «контракт» в объектно-ориентированном программировании называется интерфейсом. Любой класс может «подписать» этот «контракт», объявив, что он реализует данный интерфейс. Класс может реализовать любое количество интерфейсов.

Что такое интерфейс

Интерфейс — это особый тип абстрактного класса, который содержит только объявления методов, но не их реализацию. В интерфейсе может быть один или несколько методов, а также переменные — но только окончательные статические, то есть по сути константы. При объявлении в интерфейсе таких переменных спецификаторы `static` и `final` указывать не нужно — они принимаются по умолчанию. Каждый класс, в объявлении которого указано, что он реализует некоторый интерфейс, обязан реализовать все методы этого интерфейса.

Казалось бы, зачем вводить новое понятие интерфейса, если уже есть абстрактные классы? Дело в том, что понятие интерфейса решает про-

блемы, связанные с наследованием. Чтобы реализовать абстрактный метод класса, новый класс должен быть потомком того класса, в котором метод объявлен. А, как уже сказано, наследование нового класса от имеющегося имеет смысл только тогда, когда объекты нового класса можно считать разновидностью объектов старого класса (например, «геометрическая фигура» — «треугольник» — «прямоугольный треугольник»). Но представьте себе, что вы пишете игру, где по экрану должны перемещаться разнородные объекты — люди, монстры, снаряды и т. п. Движок игры должен уметь перемещать изображения предметов единообразно и независимо от их типа. Значит, все классы, описывающие объекты игры, должны реализовать метод перемещения — назовем его `move()`, — объявленный абстрактным где-то в общем предке этих классов. Но для классов, описывающих такие разные категории объектов, очень трудно подобрать общего предка, и организация этих классов в иерархию создаст больше проблем, чем решит.

Здесь-то и оказываются полезны интерфейсы. Интерфейсы не входят в иерархию классов, и реализовать один и тот же интерфейс имеют право классы, никак не связанные друг с другом. В примере с игрой мы просто объявим интерфейс, содержащий метод `move()`, который будет отвечать за перемещение объектов любого типа, и укажем, что все классы, описывающие объекты игры, его реализуют.

Интерфейс объявляется с помощью ключевого слова `interface`:

```
public interface Moveable {

 int GAP = 1; // public final static int GAP
 void move(int left, int top);

} // Moveable interface
```

Компилятор автоматически приписывает переменным, объявленным в интерфейсе, спецификаторы `final` и `static`, в результате чего они становятся константами. Значение константы нужно указывать при объявлении интерфейса.

Кроме того, все переменные и методы интерфейса являются открытыми, и спецификатор `public` явно указывать не обязательно.

Еще одно преимущество интерфейсов перед абстрактными классами состоит в том, что интерфейс решает проблему множественного наследования. Множественное наследование, разрешенное в других объектных языках, но запрещенное в Java, состоит в том, что подкласс объявляется потомком двух и более надклассов, получая в свое распоряжение методы всех своих предков. Отношения между классом и интерфейсом не явля-

ются отношениями наследования, хотя похожи на них, поэтому в Java одному классу разрешено реализовать несколько интерфейсов.

Для указания на то, что объявляемый класс реализует такие-то интерфейсы, служит ключевое слово `implements`, за которым через запятую перечислены имена интерфейсов:

```
class Auto extends Vehicle implements Moveable {

 // остальные переменные и методы класса Auto

 // реализация всех методов интерфейса Moveable
 public void move(int left, int top) {
 // код, перемещающий объект в положение, заданное параметрами
 } // move(int, int) method of Auto class

} // Auto class
```

Имена методов, количество и типы их параметров в классе, реализующем интерфейс, должны совпадать с объявлением этих методов в интерфейсе.

Указатель на интерфейс

Как вы помните, на объект подкласса может указывать переменная, типом которой является надкласс. Точно так же можно объявить переменную типа «интерфейс», которая будет указывать на объект класса, реализующего этот интерфейс:

```
Moveable m; // указатель на интерфейс
// экземпляр класса Auto
Auto auto = new Auto();
// экземпляр класса Moto, тоже реализующего
// интерфейс Moveable
Moto moto = new Moto ();

// переменная-указатель на интерфейс может указывать
// как на объект типа Auto…
m = auto;
m.move(10, 56);

// так и на и объект типа Moto
m = machine;
m.move(43, 29);
```

Обращение к методу интерфейса в этом случае вызовет соответствующий метод того объекта, на который указывает переменная-указатель на интерфейс. Это еще один пример полиморфизма, на этот раз благодаря использованию интерфейсов.

Расширение интерфейса

Один интерфейс может наследовать свойства другого, но ни в коем случае не свойства класса. Говорят, что первый интерфейс «расширяет» второй, и для указания этого служит то же самое ключевое слово `extends`, с помощью которого один класс объявляется наследником другого.

Если некоторый класс берется реализовать расширенный интерфейс, то этот класс должен содержать реализацию всех методов как самого интерфейса, так и всех его предков:

Листинг 3.24. Реализация расширенного интерфейса

```
interface A {
 void metodA(); // метод A() интерфейса A
} // A interface

interface B extends A {
 void metodB(); // метод B() интерфейса B
} // B interface

class IExample implements B {

 public void metodA() {System.out.println("Метод  A");}
 public void metodB() {System.out.println("Метод  B");}

} // IExample class

public class IExampleDemo {

 public static void main(String[] args) {

    IExample ie = new IExample();
    ie.metodA(); // вызов метода интерфейса A
    ie.metodB(); // вызов метода интерфейса B

 } // main(String[]) method
} // IExampleDemo class
```

Попробуйте закомментировать реализацию `metodA` в классе `IExample`, и вы получите ошибку компиляции.

3.8. Программное определение типа класса

Если вы хотите во время выполнения программы проверить, принадлежит ли данный объект к некоторому классу или реализует ли он некоторый интерфейс, воспользуйтесь оператором `instanceof`. Он возвращает значение `true`, если левый операнд (переменная объекта) является экземпляром правого (имя класса или интерфейса), и `false` в противном случае:

```
Auto a = new Auto(true);

// если класс Auto реализует интерфейс Moveable, то
// выполняется следующий после условия блок
if (a instanceof Moveable) {
 Moveable m = a;
 m.move(10, 34);
}
```

Оператор `instanceof` чрезвычайно полезен, когда вы используете полиморфизм. Пусть, например, у вас есть массив элементов типа `Vehicle`. Его элементами могут быть объекты класса `Vehicle` или его потомка — класса `Auto`. Из них только объекты `Auto` реализуют интерфейс `Moveable`, поэтому, если вы хотите вызвать метод `move()` для всех элементов массива по очереди, вам нужно предварительно проверить, подходящий ли тип у текущего элемента:

Листинг 3.25. Пример использования оператора instanceof

```
public class InstanceOfDemo {

  public static void main(String[] args) {

    Auto a = new Auto();
    Vehicle v = new Vehicle();

    Vehicle[] va = {a, v};
    for (int i = 0; i < va.length; i++) {

      System.out.println(va[i].toString());
      if (va[i] instanceof Moveable) {
        Moveable m = (Moveable) va[i];
        m.move(10+i*34, 34);
      }
    } // for
  } // main(String[]) method
} // InstanceOfDemo class
```

Обратите внимание, что для того, чтобы присвоить переменной типа `m` «указатель на интерфейс `Moveable`» элемент массива `va[i]` — указатель на объект типа `Vehicle` — потребовалось явное приведение типов, потому что тип `Vehicle` не поддерживает интерфейса `Moveable`.

ГЛАВА 4.

СТРОКИ И КОЛЛЕКЦИИ

4.1. Строки

Строки в Java

В отличие от классических языков программирования в Java строки реализованы не как основной тип данных, а как класс `String`. Данный класс входит в пакет классов `java.lang`, то есть не требует импорта для обращения к строкам. Он содержит множество методов, предназначенных для выполнения наиболее распространенных задач.

Однако использование класса `String` для операций со строками имеет определенную особенность: все его методы возвращают новый экземпляр класса, непосредственно не меняя саму строку.

Класс `String` относится к числу неизменяемых классов (`immutable classes`). Неизменяемыми называются классы с методами, которые не изменяют внутреннее состояние объекта, созданного на их основе, а возвращают новый экземпляр класса. Например, метод `concat()` не присоединяет строку, переданную в аргументе, к уже существующей строке объекта, а просто формирует новый объект — экземпляр класса `String`, содержащий полную строку. Докажем это на примере:

```
String str = "Красота спасет ";
String sentence = str.concat("мир");

System.out.println(sentence); // "Красота спасет мир"
System.out.println(str); // "Красота спасет"
```

Создание нового экземпляра класса `String`, которое приводит к повторному выделению памяти для новой строки, дает в результате снижение производительности. В связи с этим был создан класс `StringBuffer`, входящий в состав пакета `java.lang`. Этот класс обеспечивает выделение достаточного количества ресурсов памяти для хранения строк.

Далее в настоящей главе вы ознакомитесь с наиболее распространенными операциями со строками, сможете производить их слияние, осуществлять поиск, замену строки или преобразовывать текстовый тип данных в числовой.

Оптимизация строковых операций

Операции со строками чаще всего производятся с использованием класса `String`; и наиболее распространенной из них является слияние строк с помощью оператора +. Надо отметить, что такие операции требуют больших затрат системных ресурсов (особенно для выделения памяти и управления ею):

```
String s1 = "Произвольная строка символов - ";
String s2 = "Присоединенная строка";
String s3 = s1 + s2;
```

Как видите, в данном случае создается три объекта: один — с исходным текстом, второй — с присоединяемым, третий содержит обе части текста. Очевидно, что такой способ слияния не является оптимальным, так как каждый из создаваемых объектов требует выделения памяти. Для повышения эффективности программы предлагается использовать класс `StringBuffer`, который работает аналогично буферу обмена. Однако в некоторых случаях (и даже в очень многих) этим проблема не решается:

```
StringBuffer sb = new StringBuffer();
sb.append("Произвольная строка символов - ");
sb.append("Присоединенная строка");
String s3 = sb.toString();
```

Класс `StringBuffer` имеет следующий нюанс: во время своей инициализации с помощью конструктора по умолчанию, он выделяет память на 16 символов. Этих 16 символов недостаточно для размещения полной строки «Произвольная строка символов - Присоединенная строка» длиной в 52 символа. Компилятор при вызове метода `append()` вынужден выделить дополнительную память, что с каждым новым вызовом снижает эффективность выполнения всей операции. Память выделяется вызовом закрытого метода `expandCapacity()`, который удваивает объем буфера.

Для достижения оптимальной производительности попробуйте сначала приближенно рассчитать нужную длину итоговой строки. Далее укажите эту длину при создании буфера и только после этого начинайте соединять строки:

```
StringBuffer sb = new StringBuffer(52);
sb.append("Произвольная строка символов - ");
sb.append("Присоединенная строка");
String s3 = sb.toString();
```

Извлечение части строки

Для извлечения из строки определенной ее части (подстроки) вам следует вызвать метод `String.substring()`, в качестве параметров которого необходимо указать начало и длину подстроки, которая вам нужна:

```
int start = 6;
int end = 11;
String greetings = "Hello World!";
String substr = greetings.substring(start, end);
System.out.println(substr); // извлечет слово World
```

Поиск текста

Для поиска первого вхождения подстроки в строку используйте метод `indexOf()`, для поиска последнего - `lastIndexOf()` класса `String`. Указанные методы возвращают позицию (порядковый номер) первого символа искомой строки:

```
int index;
String greetings = "Hello World";
// первое вхождение
index = greetings.indexOf("e"); // index = 1
// последнее вхождение
index = greetings.lastIndexOf("o"); // index = 7
// не найдено
index = greetings.lastIndexOf("x"); // index = -1
```

Проверка на наличие записи в начале или в конце текста

Если вам необходимо узнать, начинается (или заканчивается) ли определенный текст указанной строкой, используйте метод `startsWith()` (или `endsWith()`) класса `String`. Оба метода возвращают значение типа `boolean`, соответствующее результату выполнения операции:

```
String str = "Hello from Universe";
boolean result = str.startsWith("Hell"); // true
boolean result = str.endsWith("Universe"); // true
```

Извлечение символа из строки

Для извлечения символа из строки достаточно использовать метод `charAt()` класса `String`, в котором нужно указать позицию символа в строке:

```
char c = greetings.charAt(6); // 'W' из "Hello World"
```

Замена символов в строке

Для замены символов в строке используйте метод `replace()` класса `String`. Он имеет два параметра, первый соответствует тому, что заменять; второй — чем заменять. Например:

```
// Заменяет букву «о» на букву «а» в строке greetings.
String newString = greetings.replace("o", "a");
```

Замена части строки

Для замены части строки вы можете воспользоваться несколькими из уже представленных методов класса `String`: `indexOf()`, `substring()` и методов класса `StringBuffer`:

```
// замена части строки
static String replace(String str, String pattern,
                      String replace) {
    int start = 0;
    int pos = 0;
    StringBuffer result = new StringBuffer();

    while ((pos = str.indexOf(pattern, start)) >= 0) {
      result.append(str.substring(start, pos));
      result.append(replace);
      start = pos + pattern.length();
    }

    result.append(str.substring(start));
    return result.toString();
} // replace(String, String, String)
```

Изменение регистра всех букв в строке

Для замены всех букв в строке строчными или прописными служат методы `toLowerCase()` и `toUpperCase()` класса `String`.

```
String greetings = "Hello World";
String lower = greetings.toLowerCase(); // hello world
String upper = greetings.toUpperCase(); // HELLO WORLD
```

Слияние строк

Для слияния строк вы можете использовать не только оператор +, но также метод `concat()` класса `String`.

```
String greetings = "Hello";
greetings += " ";
greetings.concat("World!");
```

Преобразование строки в числовой тип

Если вы хотите преобразовать строку в какой-нибудь числовой тип, используйте статические методы `parseXXX()` (здесь `xxx` — название числового типа) классов числовых типов — `Integer`, `Long`, `Float` и `Double`.

```
double d = Double.parseDouble("421.5e10");
float f = Float.parseFloat("421.5");
long l = Long.parseLong("421");
int i = Integer.parseInt("421");
```

Преобразование простых типов в строковый

Вы можете осуществить преобразование простых типов данных в строковый тип двумя способами. Неявный способ основан на применении оператора слияния (+) строк.

```
// Использование оператора слияния +
String s;
s = "" + true; // true
s = "" + ((byte) 0x12); // 18
s = "" + ((byte) 0xFF); // -1
s = "" + 'a'; // a
s = "" + ((short) 132); // 132
s = "" + 987; // 987
s = "" + 987L; // 987
s = "" + 9.87F; // 9.87
s = "" + 9.87D; // 9.87
```

Прямой способ основан на вызове метода `valueOf()` класса `String`.

```
// Использование метода String.valueOf()
String str = String.valueOf(true); // true
str = String.valueOf((byte) 0x12); // 18
str = String.valueOf((byte) 0xFF); // -1
str = String.valueOf('a'); // a
str = String.valueOf((short) 132); // 132
str = String.valueOf(987); // 987
str = String.valueOf(987L); // 987
str = String.valueOf(9.87F); // 9.87
str = String.valueOf(9.87D); // 9.87
```

Порядок преобразования числа в строку может быть основан и на использовании метода `toString()`:

```
class IntString {
public static void main(String[] args) {

    // преобразование целого числа в строку
    int intValue = 101;
    String stringValue = Integer.toString(intValue);

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

Перевод текста в кодировку Unicode и UTF-8

Строки записываются в Java в кодировке Unicode. Когда вам понадобится сохранить строку в файл и отправить этот файл через интернет, преобразуйте строку в UTF-8. Эту кодировку вы можете использовать в различных системах:

```
try {
    // Перевод текста в массив байтов UTF-8 из Unicode
    String string = "abc\u5639\u563b";
    byte[] utf8 = string.getBytes("UTF-8");

    // Преобразование массива байтов UTF-8 в текст
    // кодировки Unicode
    string = new String(utf8, "UTF-8");
} catch (UnsupportedEncodingException e) {
}
```

Для того, чтобы «восстановить» первоначальный вид текста в целевой системе, она должна поддерживать формат UTF-8. Иначе будет вызвано исключение `UnsupportedEncodingException`.

4.2. Библиотека коллекций

Коллекции: что это такое и зачем они нужны

В пакете `java.util` содержится библиотека коллекций (`collection framework`), которая предоставляет большие возможности для работы с множествами, хэш-таблицами, векторами, разными видами списков и т.д.

Коллекция — это объект, способный хранить группу одинаковых элементов. Она содержит методы для операций с однородными данными. Изначально Java поддерживала работу с коллекциями в рамках классов `Vector` и `Hashtable`, но с появлением JDK 1.2 возможности работы с коллекциями были расширены, возникло много открытых интерфейсов и различных видов классов, которые были включены в библиотеку коллекций.

Основные преимущества классов **collection framework** (перед классами, разрабатываемыми самостоятельно) заключаются в следующем:

- ускоряется процесс разработки и улучшается качество кода;
- обеспечивается поддержка повторного использования кода;
- производится стандартизация интерфейса ваших классов;
- реализуется поддержка многопоточного доступа.

Проверке готовых классов-коллекций уделялось много внимания, поэтому правомерно говорить об улучшении качества кода. Данные классы можно считать хорошо отлаженными с минимальным количеством ошибок, что зачастую невозможно обеспечить в самостоятельных проектах.

Основу библиотеки составляют открытые интерфейсы, которые можно использовать для создания собственных коллекций. Каждый интерфейс объявляет набор методов, которые вы обязаны реализовать в своей программе:

- `Collection` — группа элементов (охватывает `Set` и `List`);
- `Set` — множество элементов (без дублирования);
- `SortedSet` — то же самое, что `Set`, только элементы упорядочены;
- `List` — упорядоченный список;
- `Map` — словарь, то есть коллекция, в которой каждый элемент имеет уникальный ключ;
- `SortedMap` — то же самое, что `Map`, однако элементы упорядочены;
- `Queue` — интерфейс для работы с очередью.

Разумеется, интерфейсы были бы «пустыми», если бы в них не существовало встроенных классов, реализующих необходимые функции:

- `ArrayList` — список `List` как массив элементов;
- `LinkedList` — список `List`, выполняющий функции связанного списка;
- `HashSet` — множество `Set` как хэш-таблица;
- `TreeSet` — множество `SortedSet`, используемое как дерево;
- `HashMap` — индексированный словарь хэш;
- `TreeMap` — коллекция `SortedMap` древовидной структуры.

Задача каждого из интерфейсов — обеспечить простоту и удобство работы с большим количеством однотипных данных. Рассмотрим подробнее назначение каждого из этих интерфейсов.

- `Collection` — общий интерфейс, объединяющий интерфейсы `Set` и `List`. Содержит методы для добавления и удаления элементов коллекции, проверки их правильности, наличия и др.
- `Set` — неупорядоченный набор неповторяющихся элементов. Расширяет интерфейс `Collection`. Если производится попытка добавить в набор элемент, который уже в нем содержится, она будет проигнорирована.
- `List` — служит для работы с упорядоченными коллекциями. К каждому элементу такой коллекции можно обратиться по индексу. Расширяет интерфейс `Collection`.
- `Map` — предназначен для работы с коллекциями-словарями, в которых содержатся ключи и соответствующие им значения (каждому ключу соответствует только одно значение). Словарь может содержать произвольное число элементов.
- `Queue` — содержит методы для работы с очередями: в них элементы добавляются с одного конца, а извлекаются — с другого.

Методы интерфейсов и их описание сосредоточены в табл. 4.1.

Интерфейсы и их методы *Таблица 4.1*

Интерфейс	Методы	Описание метода
Collection	`boolean add (Object o)`	Добавляет новый элемент `o` в коллекцию, если ранее он там отсутствовал
	`boolean addAll (Collection b)`	Добавляет все элементы коллекции `b` в текущую
	`void clear ()`	Очищает коллекцию
	`boolean contains (Object o)`	Проверяет, содержит ли коллекция указанный элемент, и возвращает `true`, если результат проверки положителен
	`boolean containsAll (Collection b)`	Возвращает `true`, если коллекция включает все элементы коллекции `b`
	`boolean equals (Object o)`	Устанавливает эквивалентность двух коллекций
	`int hashCode ()`	Возвращает хэш-код
	`boolean isEmpty ()`	Устанавливает, является ли текущая коллекция пустой (если да, возвращает `true`)
	`Iterator iterator ()`	Создание итератора коллекции
	`boolean remove (Object o)`	Удаление элемента коллекции (если такой имеется)
	`boolean removeAll (Collection b)`	Удаляет все элементы коллекции `b` из данной, если они в ней содержатся
	`boolean retainAll (Collection c)`	Обратная операция: удаляет все элементы из текущей коллекции, кроме тех, которые содержатся в коллекции `b`
	`int size ()`	Возвращает количество элементов коллекции
	`Object [ ] toArray ()`	Преобразует все элементы коллекции в массив
	`Object [ ] toArray (Object [ ] a)`	Преобразует все элементы коллекции в массив типа `a`
Set	Аналогичный `Collection` набор методов	

Таблица 4.1 (продолжение)

Интерфейс	Методы	Описание метода
List	Все методы интерфейса `Collection`, а также следующие:	
	`void add (int index, Object obj)`	Добавляет элемент на позицию `index`, сдвигая остальные элементы коллекции, индексы которых в результате увеличиваются на единицу
	`boolean addAll (int index, Collection a)`	Добавляет все элементы коллекции `a`
	`Object get (int index)`	Возвращает элемент, соответствующий указанному индексу
	`int indexOf (Object a)`	Возвращает индекс первого вхождения объекта в коллекции
	`int lastIndexOf (Object a)`	Возвращает индекс последнего вхождения объекта в коллекции
	`ListIterator listIterator ()`	Возвращает итератор коллекции
	`ListIterator listIterator (int index)`	Возвращает итератор конца коллекции от позиции
	`Object set (int index, Object a)`	Замещает элемент с индексом `index` на объект a
Map	`boolean containsKey (Object key)`	Возвращает `true`, если ключ `key` содержится в коллекции (иначе — `false`)
	`boolean containsValue (Object value)`	Возвращает `true`, если значение `value` содержится в коллекции (иначе — `false`)
	`Set entrySet()`	Преобразует коллекцию во множество, состоящее из пар ключ-значение данной коллекции для работы с помощью вложенного интерфейса `Map.Entry`.
	`int size ()`	См. описание аналогичного метода выше
	`boolean isEmpty ()`	См. описание аналогичного метода выше
	`Object get(Object key)`	Возвращает соответствующий ключу `key` объект
	`Object put(Object key, Object value)`	Добавляет новые значение и ключ, или заменяет значение новым, если ключ уже использован
	`Set keySet()`	Преобразует ключи коллекции во множество
	`boolean remove (Object o)`	См. описание аналогичного метода выше
	`Object putAll(Map m)`	Добавляет в коллекцию все ключи и значения из словаря `m`
	`void clear ()`	См. описание аналогичного метода выше
	`Collection values()`	Преобразует все значения в коллекцию
	`boolean equals (Object o)`	См. описание аналогичного метода выше
	`int hashCode ()`	См. описание аналогичного метода выше

В таблице часто употребляется слово итератор, с которым мы ранее не сталкивались. Итератор — это дополнительный объект, используемый для перебора элементов коллекции. Так же, как и коллекции, итераторы основаны на интерфейсе, в данном случае интерфейсе `Iterator` из пакета `java.util`. Каждый итератор имеет три основных метода:

- `boolean hasNext()` — осуществляет проверку наличия элементов в коллекции после текущего;
- `Object next()` — возвращает следующий элемент коллекции;
- `void remove()` — удаляет текущий элемент из коллекции.

В библиотеке существуют также абстрактные классы, которые можно использовать для создания собственных коллекций. Такие абстрактные классы легко отличить от других по названию: в начале имени класса обязательно стоит слово `Abstract`, после чего указано название интерфейса (например, `AbstractList` — это абстрактный класс для интерфейса `List`).

Используя классы из библиотеки коллекций, следует придерживаться одного полезного правила. Если вы создаете экземпляр класса коллекции, тип переменной экземпляра должен соответствовать интерфейсу, а не самому классу:

```
TreeSet ts = new TreeSet();  // не рекомендуется
SortedSet ts = new TreeSet();// рекомендуется
```

Этим вы обеспечите возможность замены класса `TreeSet` на другой с этим же интерфейсом `SortedSet` без каких-либо последствий для других частей программы.

Далее мы рассмотрим ряд наглядных примеров, благодаря которым легче понять суть коллекций и особенности их использования.

Добавление данных простого типа в коллекцию

Ввиду особенностей языка, в коллекцию можно сохранять только объекты, а к ним не относятся простые типы данных, вроде `int` или `double`. Несмотря на это, вы сможете добавить их в коллекцию, если используете класс-оболочку для простого типа данных. Для `int` используйте класс `Integer`, для `double` - `Double` и т.д.

```
// создание словаря
Map map = new HashMap();

// создание объекта-оболочки для типа double
Double refDouble = new Double(1.23);

// добавление объекта в коллекцию
map.put("key", refDouble);

// получение класса-оболочки из коллекции
refDouble = (Double)map.get("key");

// получение значения double из объекта оболочки класса
double d = refDouble.doubleValue();
```

Перебор элементов коллекции

Для последовательного перебора элементов коллекции вы должны сначала создать объект с интерфейсом `Iterator`. Приведенный ниже фрагмент кода показывает способы просмотра коллекций и словарей.

```
// перебор объектов коллекции Set или List
for (Iterator it = collection.iterator(); it.hasNext(); ) {
Object element = it.next();  // следующий элемент
}
// перебор значений словаря
for (Iterator it = map.values().iterator(); it.hasNext(); ) {
Object value = it.next();    // следующее значение
}
// перебор ключей словаря
for (Iterator it = map.keySet().iterator(); it.hasNext(); ) {
Object key = it.next();      // следующий ключ
}
// перебор ключей и значений множества
for (Iterator it = map.entrySet().iterator();
it.hasNext(); ) {
      Map.Entry entry = (Map.Entry)it.next();
      // преобразование объекта-словаря во множество
      Object value = entry.getValue();
      // получение объекта-значения
      Object key = entry.getKey();
      // получение объекта-ключа
}
```

Создание коллекции только для чтения

Если вам требуется создать такую коллекцию, в которую нельзя добавлять новые элементы или удалять старые, сделать это просто. Добавьте в коллекцию специальный объект `unmodifiableList`, который преобразует ее в коллекцию с доступом только для чтения. Любая попытка изменить число элементов коллекции или сами элементы вызовет исключение `UnsupportedOperationException`.

```
// Создание списка имен
List imena = Arrays.asList(new String[]{"ivan", "vova"});
// делаем список неизменяемым
List spisok = new ArrayList(imena);
spisok = Collections.unmodifiableList(list);
try {
    // попытка изменения списка
    spisok.set(0, "попытка");
} catch (UnsupportedOperationException e) {
 // ошибка, список изменить не получается
}
```

Перечисление элементов во множестве

Для перечисления элементов множества можно использовать метод `hasNext()` вместе с итератором множества в цикле команды `while`.

```
Set mnozestvo = new HashSet(); // создание множества
mnozestvo.add("stepan"); // добавление элементов
mnozestvo.add("pavel");
mnozestvo.add("serg");

Iterator it = mnozestvo.iterator(); // получение итератора
while (it.hasNext()) { // цикл while
    String element = (String) it.next();
    System.out.println("Элемент: " + element);
}
```

Другие операции с множествами и списками

С множеством (`Set`) и списком (`List`) можно осуществлять большое количество разных операций. Например, копировать элементы одной коллекции в другую, удалять из одной коллекции элементы другой коллекции, создавать их пересечения или копировать коллекцию в массив. Приведем небольшой пример таких операций:

```
// создание двух множеств (или списков)
Set mnozestvo1 = new HashSet(); // List spisok1 = new ArrayList()
Set mnozestvo2 = new HashSet(); // List spisok2 = new ArrayList()

/* добавление элементов одного множества в другое
** с целью создания объединения */
mnozestvo1.addAll(mnozestvo2);
/* удаление всех элементов из множества всех элементов,
которые входят также в множество 2 (асимметричное
разделение) */
mnozestvo1.removeAll(mnozestvo2);
/* получение элементов, входящих в оба множества
(пересечение)
*/
mnozestvo1.retainAll(mnozestvo2);
// удаление всех элементов множества
mnozestvo1.clear();
```

С помощью метода `subList()` интерфейса `List` вы можете без труда уменьшить число элементов в списке до необходимого:

```
List spisok = ArrayList();
int novyRazmer = 3;
spisok.subList(novyRazmer, spisok.size()).clear();
```

ГЛАВА 5.

РАБОТА С ДАННЫМИ. АЛГОРИТМЫ

Цель данной главы — показать, как в языке Java можно двигаться от решения самых простых задач к сложнейшим алгоритмам. Все примеры в ней построены на основе пакета `java.lang`, который включает только базовые функции платформы Java.

Примеры в настоящей главе рассмотрены по принципу «от простого к сложному». Сначала мы рассмотрим использование команды `while` для создания бесконечного цикла, сможем избежать деления на ноль благодаря всего одной строке кода. Эти два решения покажут, что в языке Java можно просто решить многие вопросы, если использовать нестандартный подход.

Несколько примеров посвящено использованию побитовых операций, которые зачастую упускаются из вида, несмотря на то, что они реализуют оригинальные решения многих задач. Например, с их помощью вы можете переводить число в двоичную форму или шифровать текст.

В общем, дерзайте — и все у вас получится.

5.1. Простые, но полезные приемы

Запись бесконечного цикла while

Одной из задач, реализованных в Java, является написание надежных и безопасных программ, допускающих минимальное число ошибок. Серьезные системные ошибки, которые могут быть вызваны в ходе работы программы, выявляются компилятором языка, и он отказывается запускать такую программу.

Однако в некоторых случаях компилятор ошибается сам. Например, если программа содержит кажущийся бесконечным цикл. Бесконечные циклы очень часто используются в языке C, в Java они имели бы следующий вид:

```
while (true) {}
```

Однако компилятор Java откажется обрабатывать такой цикл, даже если он был включен в текст программы сознательно. Компилятор «пропустит» такой цикл, только если существует условие выхода из цикла, например, с помощью команды `break`:

```
while (true) {
if (...) break;
}
```

Хотя можно согласиться с авторами языка Java, что бесконечный цикл является потенциально опасным и может быть заменен другим алгоритмом, все же многие программисты с удовольствием стали бы его использовать.

Для них существует простое решение. Вполне будет достаточно, если вместо непосредственного указания значения `true` в условии команды `while`, вы будете использовать переменную типа `boolean`, как в следующем примере:

```
boolean flag = true;
while (flag) {
}
```

Компилятору было бы уже нечего возразить в ответ на приведенный выше код, так как в условии цикла `while` проверяется значение переменной, которое может измениться в ходе выполнения программы.

Как избежать деления на ноль

Вы можете использовать тернарный оператор для того, чтобы предотвратить деление на ноль. Этот оператор используется для проверки делителя на равенство нулю. Если ее результат отрицательный, вы выполняете деление, если нет — возвращаете значение ноль, как в следующем примере (см. листинг 5.1).

Листинг 5.1. Пример защиты от деления на ноль

```
class SecureDivisionOp {

public static void main(String[] args) {
int result;

    for (int divisor = -5; divisor <= 5; divisor++)
    {
      // использование тернарного оператора для защиты
      // от деления на ноль
```

```
        result = divisor != 0 ? 100 / divisor : 0;
        if (result != 0) {
        System.out.println("100 / " + divisor + " = " + result);

        }
    }
} // main(String[]) method

} // SecureDivisionOp class
```

5.2. Преобразования

Преобразование символов к верхнему регистру

Вы можете использовать побитовую операцию AND для замены строчных букв прописными. Поскольку порядковый номер строчных букв в ASCII-таблице на 32 меньше, чем у прописных, достаточно просто выставить шестой бит значения символа на ноль, и его регистр преобразуется в верхний.

```
// перевод строчных букв в прописные с помощью
// побитовой конъюнкции
public class UpCase {

public static void main(String[] args) {
    char ch;

    for (int i = 0; i < 10; i++) {
      ch = (char) ('a' + i);
      System.out.print(ch + ";");

      // эта команда отключает шестой бит
      ch = (char) ((int) ch & 0xffdf);
      System.out.print(ch + " ");
    }

} // main(String[]) method

} // UpCase class
```

Указанный код заменяет десять строчных букв алфавита (начиная с «а») прописными. Десятичное значение 65503, используемое в бинарной операции AND, в двоичной системе представлено как 1111 1111 1101 1111. Поэтому операция AND устанавливает значение шестого бита

в переменной ch на ноль. В результате выполнения программы на экран выводится следующая запись:

```
a;A b;B c;C d;D e;E f;F g;G h;H i;I j;J
```

Преобразование к нижнему регистру

В предыдущем примере мы показали, как использовать побитовую конъюнкцию для замены строчных букв прописными. Теперь мы произведем обратную операцию — переведем буквы в верхний регистр с помощью побитовой дизъюнкции, которая включит шестой бит символа.

```
// перевод прописных букв в строчные
public class LowCase {

public static void main(String[] args) {
   char ch;

   for (int i = 0; i < 10; i++) {
      ch = (char) ('A' + i);
      System.out.print(ch);

      //эта команда включает шестой бит при помощи числа 32
      ch = (char) ((int) ch | 0x20);

      System.out.print(ch + " ");
   }

} // main(String[])
} // LowCase class
```

Принцип работы программы аналогичен принципу программы UpCase. Единственным отличием является использование десятичного числа 32 в побитовой операции OR. Это вызвано тем, что число 32 представлено в двоичной системе следующим образом: 0000 0000 0010 0000. Результат запуска данной программы следующий:

```
Aa Bb Cc Dd Ee Ff Gg Hh Ii Jj
```

Преобразование чисел в двоичную форму

При программировании на системном уровне (например, при использовании сокетов (sockets) или при написании драйвера) может быть полезным просматривать передаваемые данные в двоичном виде.

Для этой цели напишем класс ShowBits, который отображает указанное число в двоичном виде и позволяет указать, сколько битов числа должно

быть отображено. Другой особенностью этого класса является возможность графически отличить при выводе на экран отдельные байты путем объединения битов в группы по 8 элементов. Текст класса приведен ниже, однако он написан так, чтобы его вызывал другой класс, то есть самостоятельно работать он не будет. Для этого мы напишем еще один класс, использующий методы этого класса.

Листинг 5.2. Пример вывода двоичной записи числа

```
import java.io.PrintStream;

public class ShowBits {

private int numbits; // количество битов для отображения

public ShowBits(int numbits) {
    this.numbits = numbits;
}
public String getBinaryForm(long val) {
    long mask = 1;
    String form = "";
    mask <<= numbits - 1;

    int spacer = 0;
    for(; mask != 0; mask >>>= 1) {

      if ((val & mask) != 0) {
      form += "1"; // бит включен
      } else {
      form += "0"; // бит выключен
      }

      // разделение битов на группы по 8
      spacer++;
      if ((spacer % 8) == 0) {
      form += " ";
      spacer = 0;
      }
    } // for

    return form;
} // getBinaryForm(long) method

public void show(long val, PrintStream out) {
String binaryForm = getBinaryForm(val);
```

```
out.println(binaryForm);
} // show()

} // ShowBits class
```

Пользоваться данным классом просто. При создании экземпляра класса укажите число нужных вам битов. Затем достаточно лишь вызвать метод `getBinaryForm()` для получения строки, состоящей только из единиц и нулей, или метод `show()`, который запишет такую строку в поток типа `PrintStream`. Пример использования класса `ShowBits` — это класс `ShowBitsDemo`:

```
import java.io.PrintStream;

public class ShowBitsDemo {

private static PrintStream out = System.out;

public static void main(String[] args) {
    ShowBits b = new ShowBits(8);
    ShowBits i = new ShowBits(32);
    ShowBits li = new ShowBits(64);

    out.print("128 в двоичном виде ");
    b.show(128, out);

    out.print("87987 в двоичном виде ");
    i.show(87987, out);

    out.print("237658768 в двоичном виде ");
    li.show(237658768, out);

    out.print("Младшие 8 бит числа 87987 в двоичном виде ");
    b.show(87987, out);
} // main(String[]) method
} // ShowBitsDemo class
```

В примере `ShowBitsDemo` показано применение класса `ShowBits` для трех разных чисел. Обратите особое внимание на последнюю запись, которая содержит только 8 младших битов 16-битного числа 87987: класс `ShowBits` можно использовать и для такой операции.

В отличие от предыдущих примеров, здесь исходный код разделен на два файла: `ShowBits.java` и `ShowBitsDemo.java`. Это обусловлено тем, что в одном файле не может быть двух классов, которые декларированы как открытые (`public`). В этом случае необходимо поместить каждый класс в отдельный файл.

Представление битов в байте

Из предыдущего примера вы узнали, как преобразовать целое число в двоичную форму записи. Подобную операцию можно проделать и с типом данных byte, в этом случае можно несколько упростить приведенный выше код.

```
import java.io.PrintStream;
// отображение одиночных битов переменной с типом данных byte
public class ByteBits {

private static PrintStream out = System.out;

public static void main(String[] args) {
    int t;
    byte val;

    val = 123;
    for (t = 128; t > 0; t = t / 2) {
      if ((val & t) != 0) {
      out.print("1");
      } else {
      out.print("0");
      }
    } // for

} // main(String[]) method

} // ByteBits class
```

В результате работы программы на экран будет выведено число 123 в двоичном виде:

```
01111011
```

В этом классе в цикле `for` в каждом повторе проверяется определенный бит переменной `val` при помощи операции AND и, если бит установлен, на экран выводится единица. Проверка осуществляется восемь раз ($2^7 = 128$).

5.3. Простое кодирование по алгоритму Base64

Очень часто в программе нужно закодировать определенную информацию, причем во многих случаях не требуется применения мощного алгоритма шифрования, который трудно взломать, а достаточно простой защиты от прямого прочтения файла. В настоящее время много интернет-стандартов применяют алгоритм Base64. Наиболее интересная сфера его применения — это, безусловно, в интернет-страницах и для передачи электронных сообщений.

Примечание

Почти каждый, кто когда-нибудь занимался вопросами безопасности, подтвердит, что расшифровать текст, закодированный при помощи алгоритма Base64, для современного компьютера не составит труда (это займет считанные секунды). Если вам требуется, чтобы была обеспечена действительно высокая степень безопасности, используйте другие алгоритмы, например MD5, RSA и т.д.

Если вы хотите написать собственную программу, реализующую данный алгоритм, советуем сначала ознакомиться с документом RFC 1521 (`http://www.ietf.org/rfc/rfc1521.txt`). Однако, если вы не хотите тратить время на изобретение велосипеда, вам лучше воспользоваться уже существующим решением и использовать классы `BASE64Encoder` и `BASE64Decoder` пакета `sun.misc`.

Листинг 5.3. Пример шифрования данных

```
import java.io.*;
import sun.misc.*;

public class Base64Demo {
public static void main(String[] args) {

/* программа должна быть запущена
**по крайней мере с одним параметром —
**строкой, подлежащей шифрованию */
if (args.length == 0) {
System.out.println("usage: Base64Demo text");
System.exit(1);
}
```

```
// кодирование текста
String text = args[0];
BASE64Encoder enc = new BASE64Encoder();
String encoded = enc.encode(text.getBytes());
enc = null;

// декодирование текста
String decoded = null;
try {
    BASE64Decoder dec = new BASE64Decoder();
    decoded = new String(dec.decodeBuffer(encoded));
    dec = null;
} catch (IOException ioe) {
    ioe.printStackTrace();
}

// вывод результатов
System.out.println("Закодировано:" + encoded);
System.out.println("Декодировано:" + decoded);

encoded = null;
decoded = null;
} // main
} // Base64Demo
```

Кодирование текста осуществляет класс `BASE64Encoder`, в примере используется его метод `encode()`. В качестве параметра он принимает массив байтов, соответствующий строке, которая разбита на отдельные символы.

Для декодирования используется класс `BASE64Decoder`. В примере используется метод `decodeBuffer()`, который принимает закодированный текст как параметр.

5.4. Применение рекурсии

Вычисление факториала при помощи рекурсии

Для вычисления факториала удобно использовать рекурсию. Факториал числа N является произведением всех чисел от единицы до N. В математических записях он обозначается как N!. Так, 6! равно (1 * 2 * 3 * 4 * 5 * 6) = 720.

Листинг 5.4. Пример вычисления факториала

```
class Factorial {

// этот метод вычисляет факториал при помощи цикла
long computeI(int n) {
long result = 1;

for (int i = 1; i <= n; i++)
result *= i;

return result;
}

// этот метод вычисляет факториал при помощи рекурсии
long computeR(int n) {
if (n == 1) return n;
return computeR(n - 1) * n;
} // compute(int)

} // Factorial class

class RecursionDemo {

public static void main(String[] args) {

Factorial f = new Factorial();

System.out.println("Факториал, рассчитанный в цикле ");
for (int i = 6; i > 0; i--) {
System.out.println(i + "! = " + f.computeI(i));
}

System.out.println("Факториал, рассчитанный с помощью рекурсии");
for (int r = 1; r <= 6; r++) {
System.out.println(r + "! = " + f.computeR(r));
}

} // main(String[]) method

} // RecursionDemo class
```

Результат выполнения программы будет следующим:

```
Факториал, рассчитанный в цикле
6! = 720
```

```
5! = 120
4! = 24
3! = 6
2! = 2
1! = 1
Факториал, рассчитанный с помощью рекурсии
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
```

Алгоритм метода `computeI()`, построенного на цикле, наверняка уже для вас очевиден. Он использует цикл, который начинается со значения `1`, и умножает общее произведение на каждое следующее число.

Рекурсивный метод `computeR()` несколько сложнее. Если он вызывается с аргументом `1`, метод также возвращает значение `1`. Иначе он возвращает значение выражения `computeR(n - 1) * n`. Условие повтора проверяется каждый раз до тех пор, пока значение аргумента не станет равным `1`.

Рекурсивный просмотр содержимого папки

Во многих случаях вам может потребоваться осуществить рекурсивный перебор (просмотр) содержимого папки. Например, когда нужно выставить одинаковую дату создания/изменения для всех хранящихся в ней файлов.

Сначала создадим объект типа `File`, где в конструкторе указывается папка, просмотр которой мы собираемся осуществить. После этого при помощи метода `list()` мы получаем список всех элементов в папке. Осуществляя просмотр папки, мы будем устанавливать (применяя метод `isDirectory()`), является ли ее элемент каталогом. Если да, то метод будет вызывать сам себя. Если нет, значит, мы работаем с файлом, для которого изменим дату на текущую.

Листинг 5.5. Пример рекурсивного просмотра содержимого папки

```
public static void sameDateToDirFiles(String dir) {

long modified = System.currentTimeMillis();
                              // текущее время (миллисекунды)
```

```
File walkDir = new File(dir);     // просматриваемая папка
String[] dirList = walkDir.list(); // список элементов в папке
// последовательный просмотр папки
for (int i = 0; i < dirList.length; i++) {

    File f = new File(dirList[i]);
    if (f.isDirectory()) {
       // элемент является также каталогом, осуществляется
       // рекурсивный вызов метода
       sameDateToDirFiles(f.getPath());
       continue;
}
// для файлов устанавливаем дату последнего изменения
f.setLastModified(modified);

} // for (int i = 0; dirList[i]; i++)

} // sameDateToDir(String)
```

5.5. Сортировка массива

Пузырьковый алгоритм

Популярным способом сортировки элементов массивов является пузырьковая сортировка. Ее широкое распространение во многом обусловлено относительной простотой в использовании. К сожалению, она не слишком эффективна в больших массивах, однако идеально подходит для упорядочивания элементов в малых массивах.

Название пузырьковой сортировки связано с тем, что принцип работы алгоритма похож на поведение пузырьков воздуха в воде. Он основан на повторном сравнении и перестановке соседних элементов массива. В ходе сортировки меньшие значения элементов передвигаются в один конец массива, большие значения — в другой. Под пузырьком здесь понимается текущий элемент массива, который движется к поверхности водоема в поисках своего места.

Количество просмотров массива на единицу меньше числа его элементов. При каждом проходе соседние элементы, стоящие не по порядку, меняются местами. Ввиду того, что данный алгоритм предполагает большое количество переборов элементов массива, он неэффективен для крупных массивов.

Листинг 5.6. Пример сортировки чисел по пузырьковому алгоритму

```
class BubbleSort {

static void sort(int nums[]) {
int t; // вспомогательная переменная

for (int a = 1; a < nums.length; a++) {
    // повтор для каждого элемента массива
    for (int b = nums.length - 1; b >= a; b--) {
      // повтор для проверки порядка соседних элементов
      if (nums[b - 1] > nums[b]) {
      // перестановка элементов
      t = nums[b - 1];
      nums[b - 1] = nums[b];
      nums[b] = t;
    }
 } // for (b)
} // for (a)
} // sort(int) method

} // BubbleSort class

public class BubbleSortDemo {

public static void main(String[] args) {

int nums[] = {99, -10, 23, 123972, 17, -654, 46, 87, -9};

System.out.print("Исходный порядок массива:");
for (int i = 0; i < nums.length; i++)
System.out.print(nums[i] + " ");

System.out.println();
BubbleSort.sort(nums);

System.out.print("Новый порядок:");
for (int i = 0; i < nums.length; i++)
System.out.print(nums[i] + " ");

System.out.println();
} // main(String[]) method

} // BubbleSortDemo class
```

Алгоритм `BubbleSort` был использован в примере для сортировки числовых значений, сохраненных в массив. Весь способ реализации основан на двух вложенных циклах `for`. Внутренний цикл контролирует процесс упорядочения элементов массива. Если результат упорядочения двух соседних элементов не соответствует требованиям, то элементы меняются местами.. При каждом проходе через массив наименьший из оставшихся элементов перемещается в правильное место. Внешний цикл просто повторяет вышеуказанный порядок действий до тех пор, пока весь массив не будет упорядочен.

Результат запуска программы будет следующим:

Исходный порядок массива: 99 -10 23 123972 17 -654 46 87 -9

Новый порядок: -654 -10 -9 17 23 46 87 99 123972

Быстрый алгоритм Хоара

Алгоритм `QuickSort`, или быстрая сортировка Хоара, относится к числу самых эффективных алгоритмов сортировки. Этот алгоритм используется для сортировки последовательностей, состоящих из большого количества элементов.

Метод `QuickSort` основан на разделении сортируемых элементов по группам. Сначала вы выбираете значение (называемое компарандом), а затем разделяете массив на две части относительно компаранда. В первую часть перемещаются все элементы, которые больше или равны компаранду; остальные элементы лежат в другой части массива. Затем процесс повторяется для каждой из созданных частей до тех пор, пока массив не будет полностью отсортирован.

Рассмотрим его работу на примере. Положим, вам надо упорядочить последовательность символов `'h'`, `'e'`, `'d'`, `'a'`, `'c'` а `'i'`. В качестве первого компаранда возьмем букву `d`. В результате первого прохода по алгоритму `QuickSort` последовательность будет разделена следующим образом:

- Начальное состояние: `hedaci`
- Первый проход: `cadehi`

Данный процесс повторяется для двух частей, состоящих из символов `cad` и `ehi`. Как можно видеть, алгоритм `QuickSort` по природе является рекурсивным, сам его принцип предполагает использование рекурсии.

Ключевым моментом алгоритма является выбор значения компаранда. На практике его выбирают либо произвольно, либо как среднюю величину для небольшой выборки из сортируемой последовательности.

Здесь мы показываем программу сортировки массива символов (см. листинг 5.5), однако вы легко можете изменить ее так, чтобы она осуществляла сортировку других типов данных.

Листинг 5.7. Пример сортировки символов по алгоритму Хоара

```
class QuickSort {

static void sort(char items[]) {
quicksort(items, 0, items.length - 1);
} // sort(char)

private static void quicksort(char items[], int left, int right) {
    int top, bottom;
    char comparand, value;

    top = left; // первая позиция
    bottom = right; // последняя позиция
    // выбор компаранда – символ из середины массива
    comparand = items[((left + right) / 2)];

    // разделение последовательности на две части
    do {
      while ((items[top] < comparand) && (top < right)) top++;
      while ((comparand < items[bottom]) &&
             (bottom > left)) bottom--;

      if (top <= bottom) {
        value = items[top];
        items[top] = items[bottom];
        items[bottom] = value;
        top++;
        bottom--;
      }
    } while (top <= bottom);

/* Если закомментированный код включить в программу,
**будет виден промежуточный результат сортировки
if (((right + 1) - left) == items.length) {
for (int i = 0; i < items.length; i++)
System.out.print(items[i]);
System.out.println();
}
*/
```

```
        // рекурсивная сортировка первой части
        if (left < bottom) quicksort(items, left, bottom);
        // рекурсивная сортировка второй части
        if (top < right) quicksort(items, top, right);

    } // quicksort(char, left, right)

    } // QuickSort class

    class QuickSortDemo {

    public static void main(String[] args) {
    char seq[] = {'h', 'e', 'd', 'a', 'c', 'i'};

    System.out.print("Исходный порядок символов:");
    for (int i = 0; i < seq.length; i++)
    System.out.print(seq[i]);

    System.out.println();
    QuickSort.sort(seq);

    System.out.print("Сортированный порядок:");
    for (int i = 0; i < seq.length; i++)
    System.out.print(seq[i]);

    System.out.println();
    } // main(String[]) method
    } // QuickSortDemo class
```

Результат выполнения программы будет следующим:

```
Исходный порядок символов: hedaci
Сортированный порядок: acdehi 5.6. Стек и очередь
```

Создание стека

Стек (stack) — это хранилище данных, которое позволяет добавлять и удалять элементы по принципу «добавлен последним — взят первым» (LIFO — Last Input, First Output). Последний добавленный элемент первым извлекается из стека. JDK содержит собственный класс `Stack`, который вполне можно было бы использовать для целей создания стека, хотя нередко лучше применить собственное решение. В примере создается стек для любого типа объектов.

Листинг 5.8. Пример создания стека

```
public class Stack {
private Object[] theArray;
private int topOfStack;

static final int DEFAULT_CAPACITY = 10;

/**
* Установка объема стека по умолчанию
*/
public Stack() {
theArray = new Object[ DEFAULT_CAPACITY ];
topOfStack = -1;
} // Stack constructor

/**
* проверка, пуст ли стек.
* @return Возвращает true если стек пустой, иначе false.
*/
boolean isEmpty() {
return topOfStack == -1;
} // isEmpty() method

/**
* @return возвращает последний добавленный элемент стека
* Не изменяет стек.
*/
Object top() {
if(isEmpty())
return null;
return theArray[ topOfStack ];
} // top() method

/**
* Извлекает элемент из стека.
*/
void pop() {
if(isEmpty())
return;
topOfStack--;
} // pop() method

/**
*@return  Извлекает и возвращает элемент с вершины стека.
*/
```

```
Object topAndPop() {
if(isEmpty())
return null;
return theArray[ topOfStack-- ];
} // topAndPop() method

/**
* Добавляет новый элемент в стек
* @param x – добавляемый объект.
*/
void push(Object x) {
if( topOfStack + 1 == theArray.length )
doubleArray( );
theArray[ ++topOfStack ] = x;
} // push(Object) method

/**
* Очистка стека
*/
void makeEmpty() {

topOfStack = -1;
} // makeEmpty() method

/**
*внутренний метод удвоения размера стека.
*/
private void doubleArray() {
Object [ ] newArray;

newArray = new Object[ theArray.length * 2 ];
for( int i = 0; i < theArray.length; i++ ) {
System.arraycopy(theArray, 0, newArray, 0,
                 theArray.length);
}
theArray = newArray;
} // doubleArray() method

} // Stack class
```

Данный стек имеет интересное свойство — автоматическое удвоение своего размера, если он переполнен.

Обратите внимание, комментарии к методам, возвращающим значение, содержат текст `@return`, а комментарии методов с параметрами содержат текст `@param`. Это — так называемые тэги JavaDoc. Утилита javadoc

использует их, чтобы автоматически сгенерировать документацию к программе.

Работа с очередями

Кроме стека, работающего по принципу LIFO, часто используется принцип FIFO (First Input, First Output), особенно это касается приложений, работающих с очередями. Очевидно, что принцип очереди понятен (кто первый вошел, тот первый и вышел), поэтому давайте сразу рассмотрим его реализацию для работы с символьным типом данных.

Листинг 5.7. Пример использования очереди

```
class Queue {
private char q[];
private int putloc; // входной индекс
private int getloc; // выходной индекс

// создание пустой очереди
Queue(int size) {
q = new char[size + 1]; // резервирование памяти
getloc = putloc = 0;
} // Queue(int) constructor

//создание очереди из другой очереди
Queue(Queue ob) {
putloc = ob.putloc;
getloc = ob.getloc;
q = new char[ob.q.length];

// копирование элементов из предыдущей очереди
System.arraycopy(ob.q, 0, q, 0, ob.q.length);
}
} // Queue(Queue) constructor

// создание очереди из массива
Queue(char a[]) {
putloc = 0;
getloc = 0;
q = new char[a.length + 1];

// копирование элементов символьного массива в очередь
for (int i = 0; i < a.length; i++)
put(a[i]);
```

```
} // Queue(char[])

// проверка переполнения очереди
boolean isFull() {
return (putloc == (q.length - 1));
} // isFull() method

// проверка пустоты очереди
boolean isEmpty() {
return (getloc == putloc);
} // isEmpty() method

// добавление символа в очередь
void put(char ch) throws IllegalStateException {
if (isFull()) {
throw new IllegalStateException("Очередь переполнена.");
}

putloc++;
q[putloc] = ch;
} // put(char) method

// извлечение символа из очереди
char get() throws IllegalStateException {
if (isEmpty()) {
throw new IllegalStateException("Fronta je pr6zdn6.");
}

getloc++;
return q[getloc];
} // get() method

} // Queue class
```

Обратите внимание на то, что использование методов `put()` и `get()` может в некоторых ситуациях привести к исключению `IllegalStateException`. Намного более правильным было бы реализовать собственные классы исключений `FullQueueException` и `EmptyQueueException`, которые будут вызваны в случае переполнения или пустоты очереди.

ГЛАВА 6.

СТАНДАРТНЫЕ БИБЛИОТЕКИ

6.1. Управление работой программы

Простая программа с параметрами

Каждая программа на Java должна начинаться открытым статическим методом `main()`. Параметры, с которыми запускается программа, передаются ей в аргументе `args` этого метода как строковый массив (тип `String`). В следующем примере все переданные программе аргументы выводятся на экран.

```
public class ByeBye {

public static void main(String[] args) {

for (int i = 0; i < args.length; i++) {
System.out.println(args[i]);
}
System.out.println("Bye Bye...");

} // main(String[]) method

} // ByeBye class
```

Выход из программы

Работа программы может быть завершена с помощью метода `exit()` класса `System`. Если этот метод вызывается с параметром `0`, работа завершена корректно. Если параметр отличен от нуля, он соответствует коду ошибки, послужившей причиной экстренного выхода.

```
public class ByeBye {

public static void main(String[] args) {

if (args.length == 0) {
System.out.println("Требуется аргумент");
System.exit(1);
}

//остальная часть программы опущена

System.exit(0);
} // main(String[]) method

} // ByeBye class
```

Выполнение программы завершается с кодом ошибки `1`, если ее запуск был осуществлен без параметров.

Открытие файлов во внешнем приложении

Зачастую необходимо открыть определенный файл с помощью того приложения, которое за ним закреплено. Например, если вы хотите отобразить интернет-страницу со справкой к вашей программе в браузере. Однако, если нужно запускать таким образом программы, сразу же возникает вопрос, как определить их местоположение. В Windows Java может определять местоположение программ самостоятельно, вы только указываете, какой файл нужно открыть при помощи команды `start` в стандартном методе `getRuntime`.

```
// пример открытия файлов во внешнем приложении
String appHelp = null;
Runtime rtm = Runtime.getRuntime();
boolean win =
System.getProperty("os.name").toLowerCase().startsWith("win");

if (win == true) {
// на платформе Windows используем команду start
appHelp = "start MyHelpPage.html";
} else {
```

```
// на другой платформе придется указать путь к программе целиком
appHelp = "<путь_к_программе> MyHelpPage.html";
}

try {
rtm.exec(appHelp);
} catch (Exception e) {
System.err.println(e.getMessage());
}
```

Обратите внимание на то, как применяется свойство `System.getProperty("os.name")`, которое определяет, запущена программа в MS Windows или нет.

В других операционных системах, например в Linux, вместо команды `start` вы должны точно указать путь к запускаемому файлу интернет-браузера данной платформы.

Запуск приложения с именованными параметрами

Как было сказано выше, мы можем передать программе аргументы при запуске. Более того, непосредственно при запуске мы можем присвоить им определенное значение, то есть практически объявить переменную еще до начала работы программы. Для этого достаточно добавить к параметру ключ `-D` в командной строке:

```
java -Duser=monty -Dlevel=expert F1Player
```

В этой строке запускается Java-программа F1Player с параметрами `user` и `level`, равными `monty` и `expert` соответственно.

Программа F1Player получает переданные параметры следующим образом:

```
public class F1Player {

public static void main(String[] args) {

String user = System.getProperty("user", "unknown");
System.out.println("Пользователь" + user +
                   " запустил игру F1Player!");

F1Starter.start();
} // main

} // F1Player class

class F1Starter {
```

```
static void start() {
String level = System.getProperty("java.version",
                                  "beginner");
System.out.println("уровень: " + level);
}

} // F1Starter class
```

Аргументы, заданные с помощью ключа `-D`, не передаются в качестве параметров метода `main()` (в переменной `args[]`), а становятся переменными среды. Порядок указания аргументов может быть произвольным, а поменять их значение можно в любом месте программы. Другое преимущество заключается в том, что вы можете присвоить аргументам значение по умолчанию, которое будет использоваться, если они не были заданы при запуске программы.

После запуска примера на экране должен появиться следующий текст:

```
Пользователь monty запустил игру F1Player!
Уровень: expert
```

Последним большим преимуществом именованных параметров является то, что их значения доступны всем классам вашей программы, и нет необходимости передавать значения в каждый класс по отдельности — достаточно лишь вызвать метод `System.getProperty()`.

6.2. Вывод

Краткая запись вывода на консоль

При написании приложения вам может надоесть без конца повторять `System.out.println()` в его коде. В этом случае можно немного сократить запись, учитывая, что `System.out` — статический член:

Листинг 6.1. Пример сокращения записи команды вывода

```
public class ShortcutOutDemo {

// объявление статического члена типа PrintStream
private static PrintStream out = System.out;

public static void main(String[] args) {

// применение сокращенной формы записи
out.println("Пример...");
```

```
out.println("...краткой записи команды вывода на консоль...");

} // main(String[])

} // ShortcutOutDemo class
```

Вывод текста в файл

При разработке программы или апплета обычно вывод идет на консоль и осуществляется с помощью команд `System.out.println()` или `System.err.println()`.

После завершения отладки программы и передачи ее пользователям может быть полезным направить отладочные сообщения записи в файл, чтобы не отвлекать пользователя на сообщения, вместе с тем имея возможность выявлять и устранять ошибки в программе.

Листинг 6.2. Пример вывода в файл

```
import java.io.*;
public class MyApp {

public static void main(String[] args) {

try {
PrintStream errOut =
      new PrintStream(new FileOutputStream("Error.log"));
System.setErr(errOut);

PrintStream sysOut =
      new PrintStream(new FileOutputStream("Debug.log"));
System.setOut(sysOut);
} catch (Exception e) {}

System.out.println("Сообщения о нормальной работе
программы");
System.err.println("Сообщения об ошибках");

} // main(String[]) method
} // MyApp class
```

Перенаправление вывода строки в файл осуществляется с помощью статических методов `setOut()` и `setErr()` класса `System`. Оба метода в качестве параметра принимают экземпляр класса `PrintStream`.

В приведенной записи кода записи об ошибках перенаправляются в файл `Error.log` а отладочные записи — в файл `Debug.log`.

Чтение выводимых другим приложением сообщений

Чтение выводимых другим приложением сообщений является распространенной стратегией, особенно на unix-платформах, где консольные программы имеют надстройки, реализующие графический интерфейс. Например, если вы используете интегрированную среду разработки, наверняка знаете, что она может отображать все выводимые на консоль сообщения непосредственно в своем графическом интерфейсе. Более того, среда разработки анализирует выводимые сообщения об ошибках, облегчая их исправление.

Далее мы рассмотрим программу, способную читать вывод внешнего приложения. Небольшая схема ее работы приведена на рис. 6.1.

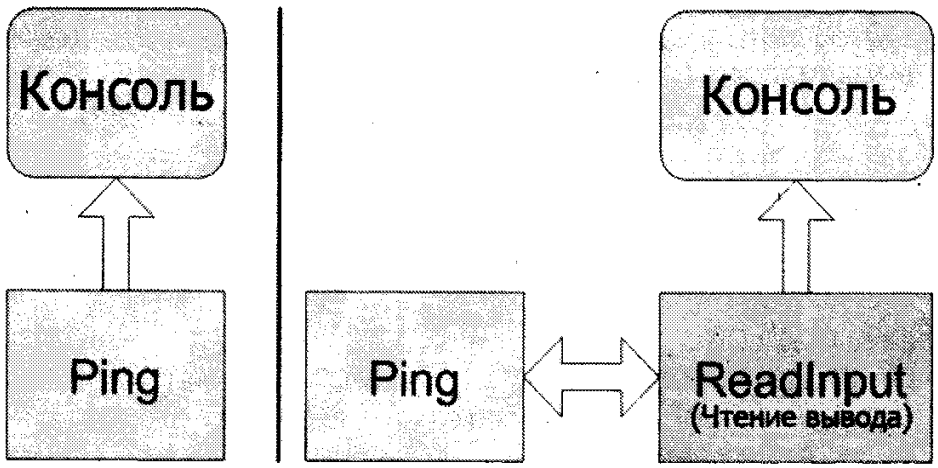


Рис. 6.1. *Структура программы для чтения данных внешнего приложения*

Программа основана на использовании команды **ping**, которую можно найти как в Unix, так и в MS Windows. Утилита **ping** используется для проверки конфигурации и доступности соединения двух узлов (чаще всего компьютеров) в сетях TCP/IP и в Интернете, она выводит на экран ряд данных о соединении. Мы будем перенаправлять эти данные в свою программу.

```
try {
//запуск программы
Process proc = Runtime.getRuntime().exec("ping localhost");
//получение нового экземпляра класса InputStream
```

```
InputStream input = proc.getInputStream();
BufferedReader in = new BufferedReader(new InputStreamReader(input));
// остальной код
catch (IOException ioe) {
ioe.printStackTrace();
}
```

Первое ограничение в использовании данного способа состоит в том, что вы должны быть владельцем процесса, данные которого собираетесь читать. Самый простой способ стать владельцем процесса — запустить его из своей программы. Для этого можно использовать метод `exec()` класса `Runtime`, как описано в п.6.3.

Метод `exec()` при вызове в качестве результата возвращает экземпляр класса `Process`, который соответствует запущенному процессу (программе). С помощью этого экземпляра класса в дальнейшем при помощи метода `getInputStream()` осуществляется доступ к выводимой данным процессом информации.

Для чтения данных лучше всего воспользоваться каким-нибудь потомком абстрактного класса `Reader`. Так как мы будем читать поток символов по строчкам, выберем класс `BufferedReader`. Он служит для чтения символов, при этом символы могут быть объединены в группы, массивы или строки.

Поскольку входной поток `InputStream` является двоичным, нам требуется как-то преобразовать его в символы. Роль посредника в данном случае сыграет экземпляр класса `InputStreamReader`.

```
String line = "";
while ((line = in.readLine()) != null)
System.out.println("java>" + line);
```

Построчное чтение входного потока обеспечивает метод `readLine()` уже упомянутого класса `BufferedReader`. Он возвращает весь текст очередной строки входного потока. Если вывод программы прекратился и считывать нечего, метод возвращает значение `null`.

```
String line = in.readLine();
while ((line == "\r") | (line =="") ){
line = in.readLine();
}
return line;
```

Однако при запуске указанного кода в некоторых операционных системах с использованием определенных виртуальных машин (например, в среде MS Windows XP с использованием Sun JVM Hotspot) некоторые строки исходного входного потока отделены друг от друга не одним

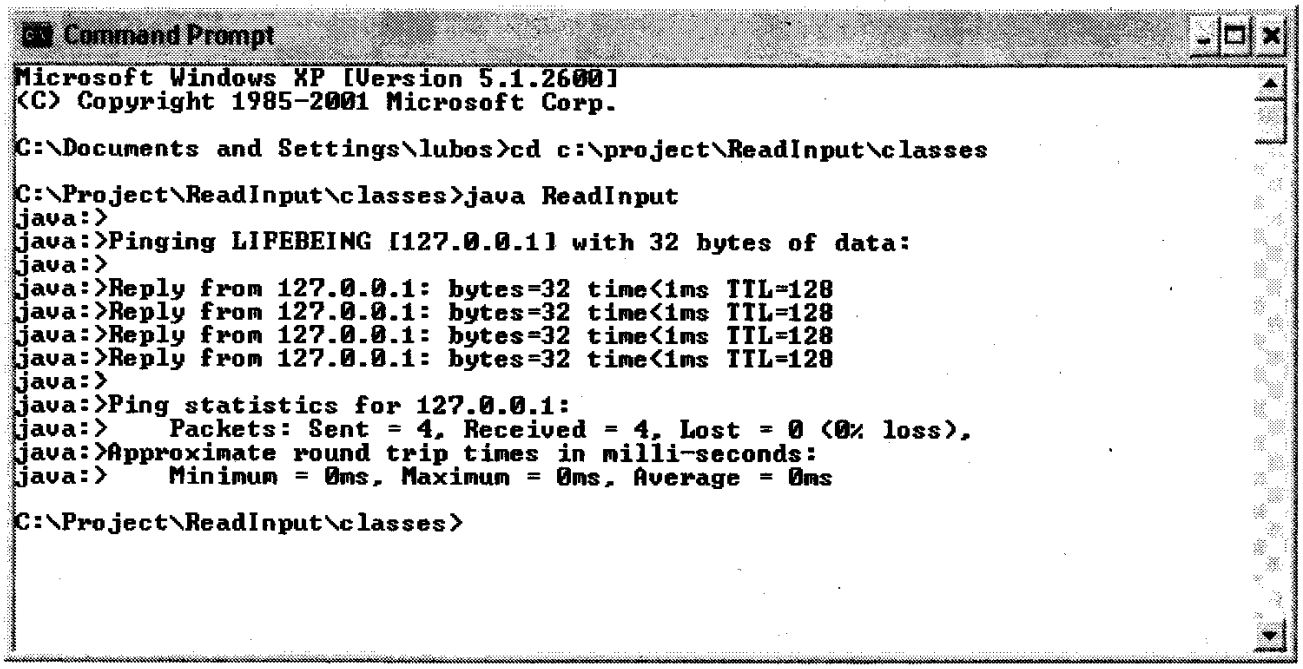


Рис. 6.2. *Отображение перенаправления команды ping localhost*

разделителем (символом возврата каретки), а двумя. Это, по-видимому, вызвано некорректной реализацией метода `readLine()`, который не может справиться со стандартными разделителями строк в MS Windows. Приведенный выше код направлен на решение данной проблемы.

Тем, кто предпочитает готовые решения, мы предлагаем написать собственный класс с названием `ReadInput`, который после получения имени программы (или команды) в параметрах запуска стартует соответствующий процесс, подключается к его входному потоку и обеспечивает чтение выводимой этим процессом информации при помощи открытого метода `readLine()`.

Листинг 6.3. Пример чтения потока вывода внешнего приложения

```
import java.io.*;

public class ReadInput {
String cmd = null;
Process proc = null;
InputStream input = null;

BufferedReader in = null;
/*Конструктор класса ReadInput
**может вызвать исключение IOException*/
public ReadInput(String cmd) throws IOException {
this.cmd = cmd;
```

```
try {
//запуск процесса, заданного именем исполняемого файла
proc = Runtime.getRuntime().exec(cmd);
//перехват его потока вывода
input = proc.getInputStream();
in = new BufferedReader(new InputStreamReader(input));
} catch (IOException ioe) {
throw ioe;
}
} // ReadInput(cmd) constructor
//метод, считывающий очередную строку вывода процесса
public String readLine() throws IOException {
String line = in.readLine();
//Проверка пустоты строки
while ((line == "\r") | (line =="") ){
line = in.readLine();
}
return line;
} // readLine() method
//Закрытие запущенной программы
public void close() {
try {
in.close();
input.close();
proc.destroy();
} catch (IOException ioe) {
ioe.printStackTrace();
} finally {
in = null;
input = null;
proc = null;
}
} // close() method

public static void main(String[] args) {
String cmd = null;
ReadInput ri = null;
//если исполняемый файл не указан, запускаем ping localhost
if (args.length > 0) {
cmd = args[0];
} else {
cmd = "ping localhost";
}

try {
```

```
ri = new ReadInput(cmd);

String line = null;
//вывод каждой строки на экран
while ((line = ri.readLine()) != null )
if (line != "\r")
{ System.out.println("java:>" + line);}

} catch (IOException ioe) {
ioe.printStackTrace();

} finally {
ri.close();
ri = null;
}

} // main()

} // ReadInput class
```

Как используется программа, покажем на примере команды `ping`:

1. Скомпилируйте класс ReadInput в среде разработки или вручную, при помощи запуска компилятора **javac**, например
   `javac ReadInput.java`.
2. Запустите скомпилированную программу и передайте ей в аргументе команду **ping**, например
   `java ReadInput "ping localhost"`.
3. Программа отобразит на консоли вывод команды `ping` с префиксом `java:>`.

Если у вас на консоли символы кириллицы отображаются некорректно, не удивляйтесь. Это связано с различием кодировок Java (Unicode) и командной строки Windows. Как решить эту проблему, мы сейчас расскажем.

Вывод русских символов на консоль

Когда вы будете учиться программировать на Java, вы обязательно столкнетесь с тем, что русскоязычные символы отображаются в ваших программах на консоли некорректно. Это связано с тем, что среда времени выполнения не всегда поддерживает используемый по умолчанию в Java набор символов Unicode.

Набор символов, используемый при выводе на консоль операционной системы Windows отличается от Unicode. Несмотря на то, что про-

граммы, написанные на языке Java, могут работать в разных системах без необходимости в компиляции для данной среды назначения, это совсем не значит, что обо всем позаботится система а программист не столкнется ни с какими проблемами совместимости. Несовпадение символьных таблиц хорошее тому подтверждение.

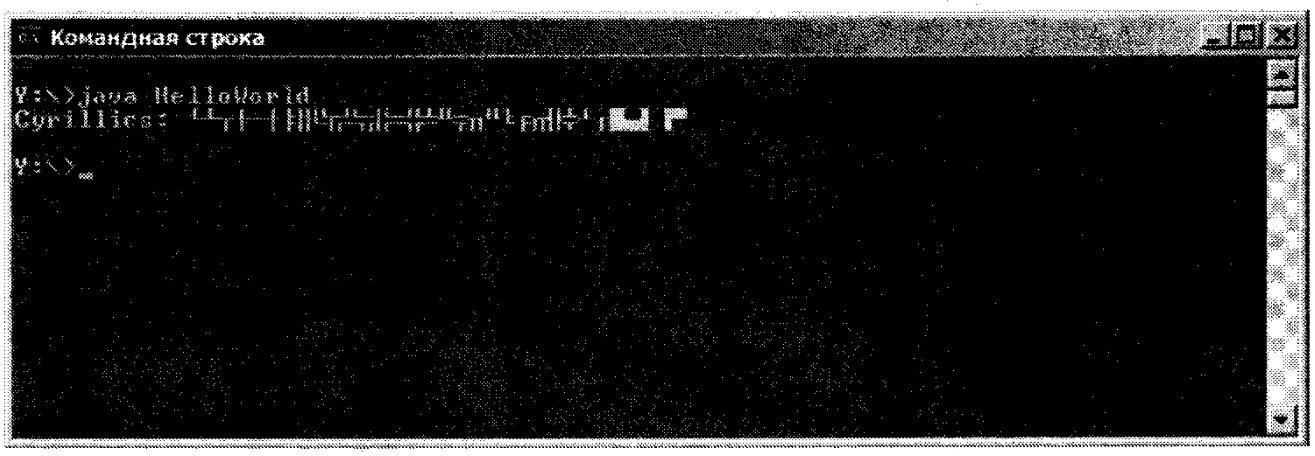


Рис. 6.3. *Некорректный вывод русских символов*

Как решить проблему с символами? Поскольку платформа Java имеет встроенный конвертер, преобразующий текст из одной кодировки в другую, используемую в определенной операционной системе, он должен указать, какой набор символов применен в данной программе. Для консоли операционной системы Windows XP это набор символов `Cp866`.

```
import java.io.*;
public class HelloWorld {
public static void main(String[] args) {

try {
PrintStream ps = new PrintStream(System.out, true, "Cp866");
System.setOut(ps);
} catch (UnsupportedEncodingException uee) {
System.out.println("Environment doesn't
                    support encoding.");
} finally {
System.out.println("Вывод русских символов
поддерживается!");
}
}
}
```

В JDK мы можем использовать для этого модифицированный класс `PrintStream`, который с помощью нового конструктора позволяет ука-

зать необходимый формат кодирования символов в третьем параметре. Далее достаточно лишь вызвать метод `setOut()` класса `System` для перенаправления входного потока.

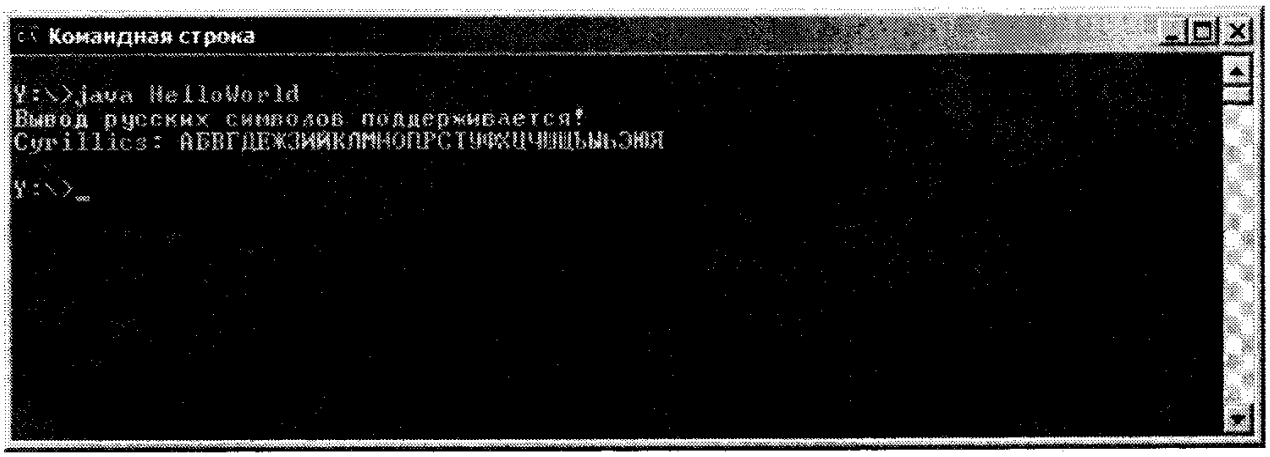


Рис. 6.4. *Теперь вывод программы в полном порядке*

Для того чтобы нам не пришлось указывать формат кодирования в каждой программе, можно создать вспомогательный класс, который за нас выполнит «грязную» работу.

```
/** Класс, запускающий PrintStream
* с русскоязычной кодировкой
*/
public final class RusPrintStream extends PrintStream {

public RusPrintStream()
throws UnsupportedEncodingException {
super(System.out, true, "Cp866");
} // RusPrintStream()

} // RusPrintStream class
```

После создания этого класса достаточно будет в начале каждого вашего консольного приложения добавлять одну строку, которая укажет, что вывод программы осуществляется через наш вспомогательный класс.

```
// Пример использования класса RusPrintStream()
public class HelloWorld {

public static void main(String[] args) {

// включение вывода русскоязычных символов на консоль
try {
System.setOut(new RusPrintStream());
} catch (UnsupportedEncodingException uee) {
```

```
System.out.println("Environment doesn't support
                    CP866 encoding.");
} finally {
System.out.println("Здравствуй, Мир!");
}

} // main(String[])

} // HelloWorld
```

Приведенный класс имеет еще одно преимущество — его можно использовать и для вывода сообщений об ошибках программы. Достаточно будет создать дополнительный экземпляр класса `RusPrintStream` и выводить сообщения об ошибках через него, используя метод `setErr()`.

```
try {

// замещение стандартного вывода
System.setOut(new RusPrintStream());
// замещение вывода об ошибках
System.setErr(new RusPrintStream());

} catch (UnsupportedEncodingException uee) {
// Ошибка вывода: среда не поддерживает кодировку CP866
System.out.println("Environment doesn't support
                    CP866 encoding.");
} finally {
System.out.println("Здравствуй, Мир!");
}
```

Вывод русских символов на консоль в старых версиях JDK

Класс, описанный в предыдущем примере, работает только в JDK версии 1.4 и выше. Очевидно, что русские символы можно использовать и на предыдущих платформах. В этом случае программный код будет несколько сложнее. Это и было причиной, по которой компания Sun модернизировала класс `PrintStream`, реализовав в нем возможность поддержки различных языков вывода консоли.

Русские символы в программе, написанной для JDK 1.3 и для более ранних версий, не выводятся на консоль обычным вызовом методов `printXX()` класса `System.out`. Здесь необходимо создать цепочку классов для обработки выходного потока.

```
public class HelloWorld {

public static void main(String[] args) {
```

```
PrintWriter pw = null;

try {
Writer osw = new OutputStreamWriter(System.out, "Cp866");
BufferedWriter bw = new BufferedWriter(osw);
pw = new PrintWriter(bw, true);
bw = null;
osw = null;
} catch (UnsupportedEncodingException uee) {
System.out.println("Environment doesn't support
                    Cp866 encoding.");
} catch (IOException ioe) {
ioe.printStackTrace();

} finally {
pw.println("Здравствуй, Мир");
}
} // main(String[])
} // HelloWorld class
```

Как видите, текст этой программы код длиннее, чем предыдущей для JDK 1.4, и к тому же, согласитесь, он выглядит несколько неуклюже, хотя бы тем, что вам придется повторять этот фрагмент кода в каждой своей программе. Это относится и к непосредственному выводу текста на экран:

```
RusPrintWriter pw = new RusPrintWriter(System.out);
pw.println("Здравствуй, Мир ");
pw = null;
```

Создадим другой вспомогательный класс `RusPrintWriter`, который будет принимать в конструкторе в качестве параметра объект типа `PrintStream`. Объект `PrintStream` возвращается методами `System.out` и `System.err`. Таким образом мы сможем использовать наше решение и для перенаправления сообщений об ошибках.

```
// класс для вывода русских символов на консоль
public final class RusPrintWriter {

private PrintWriter pw = null;
private PrintStream ps = null;

public RusPrintWriter(PrintStream ps) {
Writer osw = null;
BufferedWriter bw = null;

this.ps = ps;
try {
```

```
osw = new OutputStreamWriter(ps, "Cp852");
bw = new BufferedWriter(osw);
pw = new PrintWriter(bw, true);
} catch (UnsupportedEncodingException uee) {
/* вы можете игнорировать исключение,
** или обрабатывать его в других классах так,
** чтобы оно не влияло на остальной код
** смотрите реализацию метода println(). */
} finally {
osw = null;
bw = null;
}

} // RusPrintWriter(PrintStream) constructor

// вывод строк
public void println(String line) {

if (pw != null) {

pw.println(line);
} else {
ps.println(line);
}
}
} // RusPrintWriter class
```

6.3. Запуск внешнего приложения

Если требуется запустить из вашей программы другое приложение (например, текстовый редактор), для этого достаточно воспользоваться перегружаемым методом `exec()` стандартного класса `Runtime`. Простейшим вариантом его применения является вызов с одним строковым параметром `String`, в котором будет указан путь к запускаемому файлу.

Листинг 6.4. Пример запуска внешнего приложения

```
public class ExecProcess {

public static void main(String[] args) {
try {
```

```
/* запуск программы Блокнот
* в Windows */
Runtime.getRuntime().exec("notepad.exe");

} catch (IOException ioe) {
ioe.printStackTrace();
}
} // main()

} // ExecProcess class
```

Всего существует несколько вариантов метода `exec()`, которые хорошо задокументированы (`http://www.javaworld.com/javaworld/jw-12-2000/jw-1229-traps.html`):

- `public Process exec(String command);`
- `public Process exec(String [] cmdArray);`
- `public Process exec(String command, String [] envp);`
- `public Process exec(String [] cmdArray, String [] envp);`

В каждом из этих вариантов вы можете передать методу в качестве параметров одиночную строку, содержащую путь к программе и ее параметры запуска, либо массив строкового типа, в элементах которого перечислены сначала путь к программе, а затем все ее аргументы по отдельности, либо массив с переменными среды.

6.4. Перехват всех видов ошибок и исключений

Если на определенном этапе выполнения программы вам необходимо обработать все возможные типы ошибок и исключений, укажите для этой цели в блоке `catch` объект класса `Throwable`. Этот класс является базовым для всех типов ошибок и исключений в языке Java.

```
try {
int a[] = new int[4];
a[5] = 6; // эта строка вызывает исключение
} catch (Throwable t) {
System.err.println("Исключение: " + t.getMessage());
t.printStackTrace();
}
```

Код в блоке `catch (Throwable t)` обрабатывает любое исключение. Класс `Throwable` следует использовать, только когда у вас есть веские основания не обрабатывать отдельные виды исключений. Поскольку

в приведенном примере может иметь место только исключение вида ArrayIndexOutOfBoundsException, программа должна была бы выглядеть следующим образом:

```
try {
int a[] = new int[4];
a[5] = 6; // эта строка вызывает определенный тип
исключения
} catch (ArrayIndexOutOfBoundsException e) {
System.err.println("Исключение: " + e.getMessage());
e.printStackTrace();
}
```

6.5. Работа с числами

Преобразование целого числа в строку и обратно

Преобразование числа в строку бывает нужно, например, для записи его в файл, а обратное преобразование — для чтения числа из файла. Задачу такого преобразования поможет решить класс Integer — объектная «оболочка» простого типа int.

Класс Integer содержит два интересных статических метода. Первый из них, parseInt(), служит для извлечения числа из строки; второй, toString(), выполняет обратную операцию:

```
class StringInt {

 public static void main(String[] args) {

    // преобразование строки в целое число
    String stringValue = "101";
    int intValue = Integer.parseInt(stringValue);
    System.out.println(intValue);
 } // main(String[])
} // StringInt class

class IntString {

 public static void main(String[] args) {

    // преобразование целого числа в строку
    int intValue = 101;
    String stringValue = Integer.toString(intValue);
```

```
    System.out.println(stringValue);
  } // main(String[])
} // IntString class
```

Максимальные и минимальные значения числовых типов

Каждый числовой тип данных может использоваться для сохранения чисел только определенного диапазона. Диапазон значений данных числового типа, то есть их максимальное или минимальное значение, задается константами MAX_VALUE и MIN_VALUE класса-оболочки числового типа. Следующая команда `for` повторяется во всем диапазоне значений целого типа данных `int`.

```
// Внимание: цикл выполняется очень долго
for (int i = Integer.MIN_VALUE; i <= Integer.MAX_VALUE;
i++) {
 System.out.println(i);
}
```

Преобразование числа в двоичную, восьмеричную и шестнадцатеричную форму

Если вы хотите преобразовать десятичное целое число в двоичную, восьмеричную или шестнадцатеричную систему или выполнить обратную операцию, используйте для этого метод `parseInt()`. Перегружаемый вариант этого метода, имеющий два параметра, позволит вам передать не только саму строку (в первом параметре), но и указать систему счисления, в которую производится преобразование. Для осуществления противоположного действия используйте перегружаемый метод `toString()`, который работает аналогичным образом.

```
int cislo = 63;
String s;

cislo = Integer.parseInt("111111", 2);   // 63
s = Integer.toString(cislo, 2); // 111111 в двоичной записи

cislo = Integer.parseInt("77", 8);       // 63
s = Integer.toString(cislo, 8); // 77 в восьмеричной записи

cislo = Integer.parseInt("63");          // 63
s = Integer.toString(cislo);   // 63 в десятичной записи

cislo = Integer.parseInt("3F", 16);      // 63
s = Integer.toString(cislo, 16); // 3F в шестнадцатеричной записи
```

6.6. Работа с датой и временем

Получение даты и времени из строки

Существует много ситуаций, когда вам может понадобиться выделить день, месяц, год или другую информацию из полной даты. В Java на этот случай предусмотрен класс `SimpleDateFormat` из пакета `java.text`. Вызывая методы `applyPattern()` и `format()`, можно получить необходимую часть даты (или времени).

Метод `applyPattern()` имеет единственный строковый параметр, который указывает, какую часть даты (или времени) нужно извлечь, например букву «y» для обозначения года. Если затем вызвать метод `format()` с параметром `Date`, он возвратит строку, соответствующую данной части даты. Эту строку можно преобразовать в число уже известным нам способом, вызывая статический метод `parseInt()` класса `Integer`. В приведенном примере наглядно отражены все указанные действия.

Листинг 6.5. Пример чтения даты и времени в программе

```
public class DatePart {

private Date fromDate = null;
private SimpleDateFormat formatter = null;

public DatePart(Date anyDate) {
fromDate = anyDate;
formatter = new SimpleDateFormat("EEE MMM dd hh:mm:ss yyyy",
Locale.getDefault());
} // DatePart constructor

// возвращает день
public int getDay() {
formatter.applyPattern("d");
return Integer.parseInt(formatter.format(fromDate));
} // getDay()

// возвращает месяц
public int getMonth() {
formatter.applyPattern("M");
return Integer.parseInt(formatter.format(fromDate));
} // getMonth()

// возвращает год
public int getYear() {
```

```
formatter.applyPattern("y");
return Integer.parseInt(formatter.format(fromDate));
} // getYear()

// возвращает час
public int getHour() {
formatter.applyPattern("h");
return Integer.parseInt(formatter.format(fromDate));
} // getHour()

// возвращает минуты
public int getMinute() {
formatter.applyPattern("m");
return Integer.parseInt(formatter.format(fromDate));
}

} // DatePart class
```

Данный класс используется следующим образом. Вы вызываете один из приведенных методов, например `getMonth()`, а он возвращает необходимое значение (в приведенном примере метод возвращает месяц).

```
public class DatePartTest {

public static void main(String[] args) {

Date currentDate = new Date();
DatePart dp = new DatePart(currentDate);
int month = dp.getMonth();

System.out.println("Текущий месяц: " + month);
} // main(String[])

} // DatePartTest class
```

Получение даты и времени по григорианскому календарю

Как известно, летоисчисление в наших географических широтах ведется по григорианскому календарю. В Java существует в пакете `java.util` класс `GregorianCalendar`, который помогает оперировать датами, в том числе получать определенные элементы указанного времени и даты.

```
Date d = new Date();
GregorianCalendar gc = new GregorianCalendar();
gc.setTime(d);
```

```
// получение элементов даты и времени
System.out.println("Год: " + gc.get(gc.YEAR));
System.out.println("Месяц: " + gc.get(gc.MONTH));
System.out.println("День:" + gc.get(gc.DAY_OF_MONTH));
System.out.println("Час: " + gc.get(gc.HOUR_OF_DAY));
System.out.println("Минуты: " + gc.get(gc.MINUTE));
System.out.println("Секунды: " + gc.get(gc.SECOND));
```

Все работает довольно просто. Для класса `GregorianCalendar` устанавливаем вызовом метода `setTime()` текущую дату, после чего можно, последовательно вызывая метод `get()`, получить отдельные элементы даты и времени. Положим, программа была запущена 15 ноября 2005 года в 22:15:45. Тогда на экране появится следующая запись:

```
Год: 2005
Месяц: 11
День: 15
Часы: 22
Минуты: 15
Секунды: 45
```

Определение високосности года

Нередко в программе нужно определить, является ли определенный год високосным. Для этого в классе `GregorianCalendar` предусмотрена специальная функция `isLeapYear()`:

```
// пример проверки, является ли указанный год високосным
public class LeapYearDemo {

public static void main(String[] args) {

GregorianCalendar gc = new GregorianCalendar();
for (int year = 1996; year <= 2012; year++)
if (gc.isLeapYear(year) == true) {
System.out.println("Год" + year + " високосный");
}
// for

} // main(String[] args)

} // LeapYearDemo class
```

Как видите, метод `isLeapYear()` класса `GregorianCalendar` возвращает значение `true`, если год високосный, и `false`, если нет.

Приведенная выше программа проверяет последовательно все года с 1996 по 2012, выводя на экран сообщения только о високосных годах.

```
1996 год - високосный
2000 год - високосный
2004 год - високосный
2008 год - високосный
2012 год - високосный
```

6.7. Получение числового кода символа

Если необходимо получить числовой код, соответствующий данному символу в кодировке ASCII, достаточно присвоить значение типа `char` переменной типа `int`, а система сама произведет необходимое преобразование.

```
char bChar = 'B';
int charVal = bChar;
System.out.println("" + charVal);
```

Приведенный фрагмент кода при запуске отображает число 66, которое соответствует в таблице ASCII символу `B`. Если вы хотите осуществить обратное действие, то есть преобразовать число в переменную типа `char`, нужно выполнить команды такого вида:

```
int charVal = 66;
char bChar = (char) charVal;
System.out.println("" + bChar);
```

Результатом такого преобразования является вывод на консоль символа `B`.

6.8. Вывод файлов протокола на консоль

Ниже приведено решение, которое можно использовать, если какая-либо программа выводит сообщения об ошибках в файл, а не непосредственно на консоль. Типичным примером могут быть JRun или интернет-приложения, запущенные в контейнерах (Tomcat и т.д.).

В таких программах очень часто приходится открывать файл протокола в текстовом редакторе и проверять, есть ли там изменения. Было бы очень удобно, если бы у нас была возможность автоматически отображать все изменения в файле протокола, выводя эту информацию на консоль. Именно это побудило написать следующую программу.

Сначала программа открывает нужный файл при помощи класса `BufferedReader`. Он используется нами потому, что способен читать

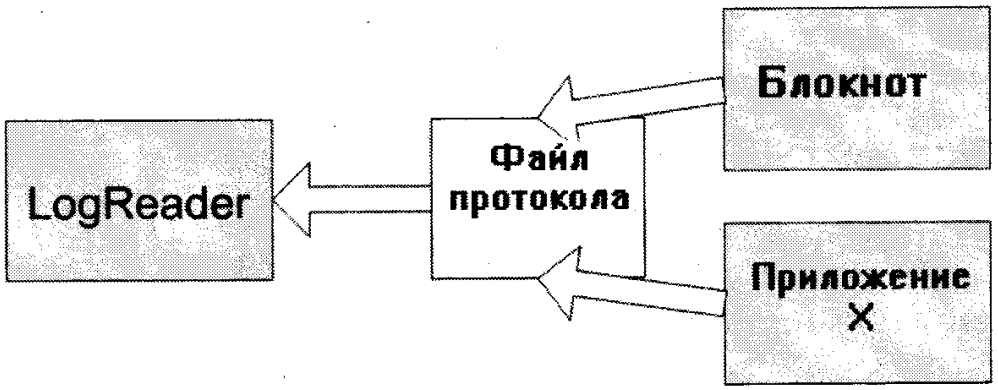


Рис. 6.5. *Структура программы для чтения файла протокола*

входной поток по частям. Если класс в ходе чтения данных обнаружит изменение в потоке вывода приложения, он отобразит его на консоли. Если изменения нет, программа на некоторое время перейдет в спящий режим. И так по кругу.

```
// Пример чтения из файла протокола выводимой
// приложением информации
import java.io.*;
public class LogReader {

public static void main(String[] args) {

/* первый параметр должен содержать имя файла протокола */
if (args.length == 0) {
System.out.println("Вызов: java LogReader <файл_
протокола>");
System.exit(0);
}

/* последовательное чтение файла*/
String logFile = null;
BufferedReader br = null;
try {
logFile = args[0];
br = new BufferedReader(
new InputStreamReader(
new FileInputStream(logFile)));
while (true) {
String line = br.readLine();
if (line != null) {
// отображение строки
System.out.println(line);
} else {
```

```
try {
//спящий режим программы
Thread.sleep(500);
} catch (InterruptedException ie) {
ie.printStackTrace();
} // try {Thread.sleep(500)}
} // if (line != null)
} // while(true)
} catch (IOException ioe) {
ioe.printStackTrace();
}
} // main(String[] args)
} // LogReader class
```

Теперь, после разработки программы, опишем ее действие на примере записи файла протокола с помощью стандартной программы Блокнот. Изменения в файле будут сразу же после его сохранения отображаться на консоли:

1. Запустите Блокнот или другой редактор текстовых файлов.
2. Введите какой-либо текст, например «строка 1».
3. Сохраните файл под названием `text.log` в папку программы LogReader.
4. Запустите программу командой `"java LogReader text.log"`.
5. Программа выводит содержимое файла `text.log` на консоль и ожидает последующих изменений.
6. Напишите в редакторе еще одну строку текста и сохраните файл.
7. Практически мгновенно (не позднее, чем через полсекунды) изменение будет отображено на экране.
8. Вы можете повторять действия пунктов 6 и 7, пока вам это не надоест.
9. Завершите программу LogReader при помощи команды «Ctrl»+«C».

Конечно, данная программа не совсем идеальна и ее можно и нужно было бы улучшить. Например, она отображает только добавленные в файл записи и не справляется с ситуацией, когда часть текста была удалена. Мы оставляем реализацию этих функций за вами.

Эта маленькая и простая программа может стать очень полезной, особенно если ее использовать с предыдущим инструментом `ReadInput`. Их действие вы наверняка оцените, когда нужно будет отслеживать вывод программы на экран, если ее исходный код вам не доступен.

На основе полученных в ходе наблюдения за работой программы данных вы можете относительно быстро реагировать на события в ней (например, можно изменить действия в ней, запустить или перезапустить ее и т.д.).

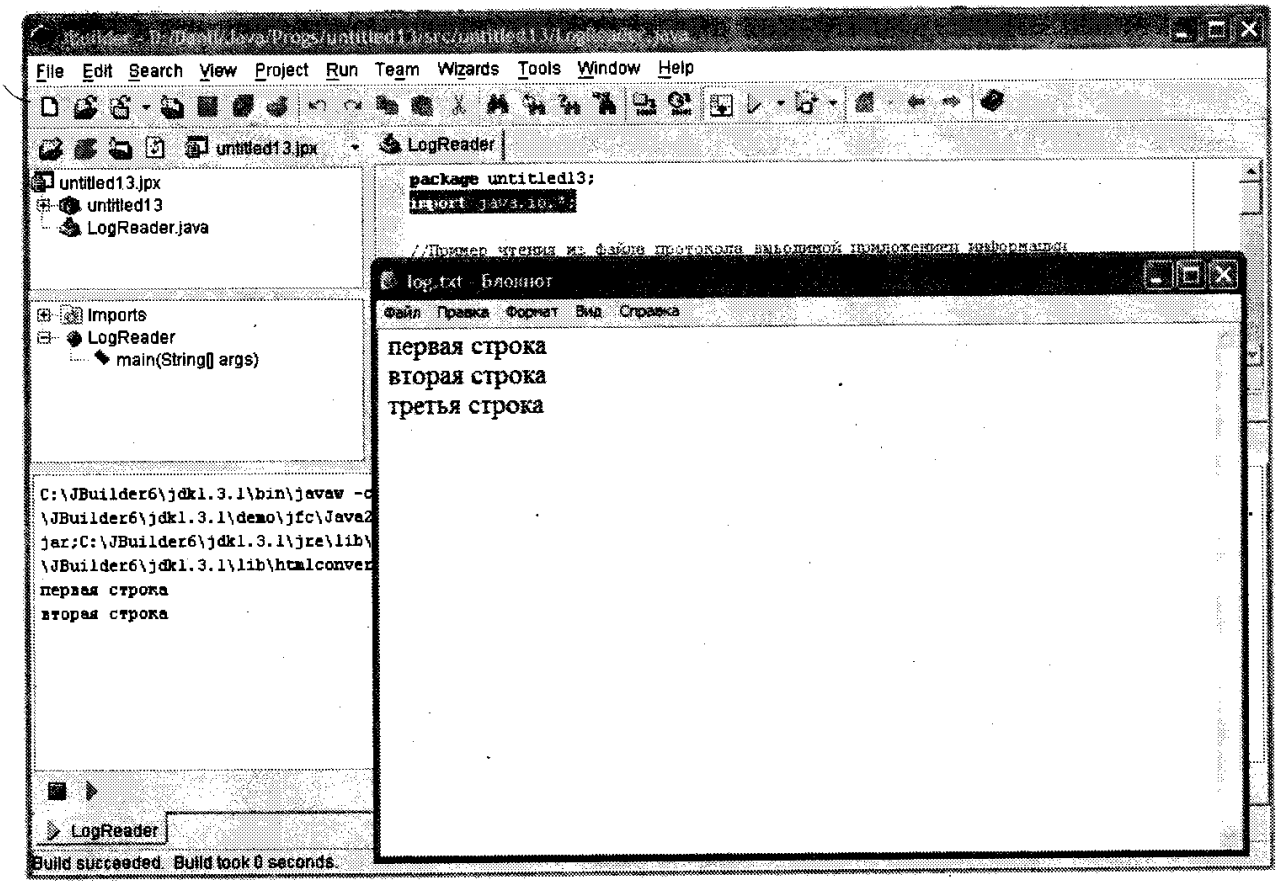


Рис. 6.6. *Пример использования инструмента LogReader*

6.9. Безопасность

Шифрование пароля

Рассмотренный ранее способ кодирования по алгоритму Base64 очень ненадежен, о чем мы уже неоднократно говорили. Что делать, если необходимо использовать настоящую шифровку (например, чтобы скрыть пароль)? Очень частой ошибкой в этой области является отправка пароля (даже в зашифрованном виде) адресату через сеть или сохранение его в файл (базу данных).

Сохранять так пароль очень небезопасно, поскольку его шифрованную форму очень легко взломать. Чтобы себя обезопасить, лучше применить так называемую хэш-функцию (hash). Хэш-функции формируют ключ фиксированной длины на базе переданного им произвольного массива байтов. Каждая хэш-функция удовлетворяет следующим двум условиям:

- Алгоритм преобразования исключает возможность наличия одинаковых ключей для двух разных строк.
- Итоговый ключ не содержит никакой информации из преобразуемых данных.

Хэш-ключи иногда считаются аналогом отпечатка пальцев.

Примечание

Используемый здесь алгоритм SHA-1 обозначен как однонаправленный. Это значит, что результат шифровки пароля невозможно использовать для получения его первоначального вида.

В пакете `java.security` вы найдете класс `MessageDigest`, который предлагает для использования две функции, генерирующие уникальные ключи. Эти две функции реализуют алгоритм MD-5 или SHA-1.

```
public class PasswordEncrypter {

// зашифрованный пароль
public static byte[] encrypt(String password) {

try {
MessageDigest d = MessageDigest.getInstance("sha-1");
d.update(password.getBytes());
return d.digest();
} catch (NoSuchAlgorithmException nsae) {
// молча игнорируем исключения..
}

// возвращает null если шифрование не удалось
return null;
} // encrypt(String)

// остальной код

} // PasswordEncrypter
```

Для шифровки пароля на основе приведенного выше исходного кода необходимо выполнить три обязательных шага:

1. Получить экземпляр класса `MessageDigest` при помощи статического метода `getInstance()`, которому вы передаете название используемого алгоритма, в нашем случае это алгоритм SHA-1.
2. Передать пароль с помощью метода `update()`. На основе пароля будет генерирован хэш-ключ.
3. Получить хэш-ключ методом `digest()`. Возвращаемое методом значение — это массив байтов.

Поскольку в результате использования некоторых алгоритмов шифровки и хэширования данных мы можем получить ключ, в котором будут также и непечатаемые символы, например символ конца строки, табуляции и т.д.,

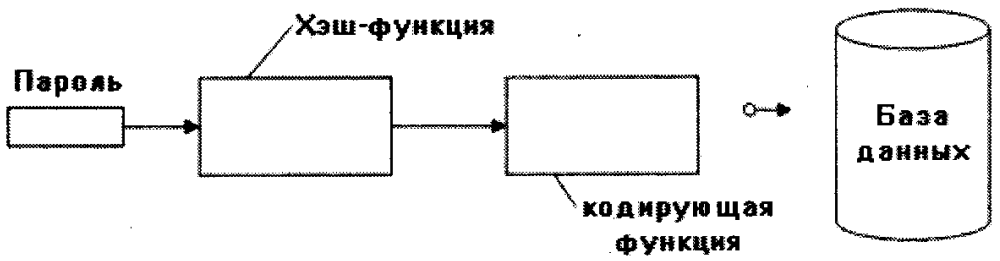


Рис. 6.7. *Обычная схема для сохранения хэш-ключа в базу данных*

можно преобразовать результат кодировки (хэш-ключ) по алгоритму Base64. В результате кодированный хэш-ключ будет иметь символьный вид.

```
public static String encryptAndEncode(String password) {
BASE64Encoder enc = new BASE64Encoder();
String encoded = enc.encode(encrypt(password));
enc = null;

return encoded;
}
```

Теперь у вас есть полнофункциональная реализация функции шифровки пароля, результат применения которой вы можете сохранить в файл на диске, вывести на консоль или переслать по сети. Пример использования класса `PasswordEncrypter` может быть следующим:

```
// пример использования класса PasswordEncrypter
// для шифровки пароля
public class PasswordTest {

public static void main(String[] args) {

String pwd = "пароль";
String encrypted = new String(PasswordEncrypter.encrypt(pwd));
String encoded = PasswordEncrypter.encryptAndEncode(pwd);

System.out.println("Зашифрованный пароль:" + encrypted);
System.out.println("Кодированный шифр:" + encoded);
}
}
```

Для повышения надежности программы имеет смысл использовать оба предложенных алгоритма (SHA-1 и MD-5). С этой целью изменим класс `PasswordEncrypter` так, чтобы пользователь мог выбрать нужный алгоритм. Внося изменения, порядок статических методов оставим прежним, однако на этот раз не будем игнорировать исключение `NoSuchAlgorithmException`.

```
// окончательный вид класса для шифрования пароля
public class PasswordEncrypter {

// пользователь может указать не только пароль,
// но и алгоритм
public static byte[] encrypt(String password,
                             String algorithm)
throws IllegalArgumentException {

try {

MessageDigest d = MessageDigest.getInstance(algorithm);
d.update(password.getBytes());
return d.digest();

} catch (NoSuchAlgorithmException nsae) {

throw new IllegalArgumentException("Illegal algorithm value.");

}
} // encrypt(String, String)

// упрощенный вызов без указания алгоритма
public static byte[] encrypt(String password)
throws IllegalArgumentException {
return encrypt(password, "sha-1");
} // encrypt(String)

// шифровка хэш-ключа
public static String encryptAndEncode(String password,
String algorithm)
throws IllegalArgumentException {
BASE64Encoder enc = new BASE64Encoder();
String encoded = enc.encode(encrypt(password, algorithm));
enc = null;

return encoded;
} // encryptAndEncode(String, String)

// упрощенная версия для формирования хэш-ключа по алгоритму SHA-1
public static String encryptAndEncode(String password)
throws IllegalArgumentException {
return encryptAndEncode(password, "sha-1");
}
} // PasswordEncrypter class
```

Проверка пароля

Алгоритмы SHA-1 и MD-5, реализованные в классе `MessageDigest` пакета `java.security`, являются однонаправленными. Как уже было сказано, результат шифровки в них нельзя расшифровать, то есть восстановить по хэш-ключу исходный текст.

Примечание

Расширенная версия алгоритма MD-5 даже генерирует в каждом случае ее использования для одного и того же текста разный результат. Она также добавляет к результату так называемый `random prefix` (`salt`), благодаря которому защита становится более надежной.

На первый взгляд может показаться проблемой отсутствие самого пароля в базе данных: как же его сверить, ведь в базе его нет? Однако сам пароль в случае использования однонаправленных алгоритмов нам и не потребуется, достаточно преобразовать то, что ввел пользователь в качестве пароля, зашифровать введенный текст по тому же алгоритму и сравнить результат с хранимым в базе данных хэш-ключом. Если пользователь ввел правильный пароль, оба ключа будут одинаковыми.

Преимущество такого приема заключается в том, что расшифрованный пароль (имеется в виду пароль, полученный из базы данных, файла и т.п.) не используется во время работы программы. Это создаст определенные трудности тому, кто хочет его узнать.

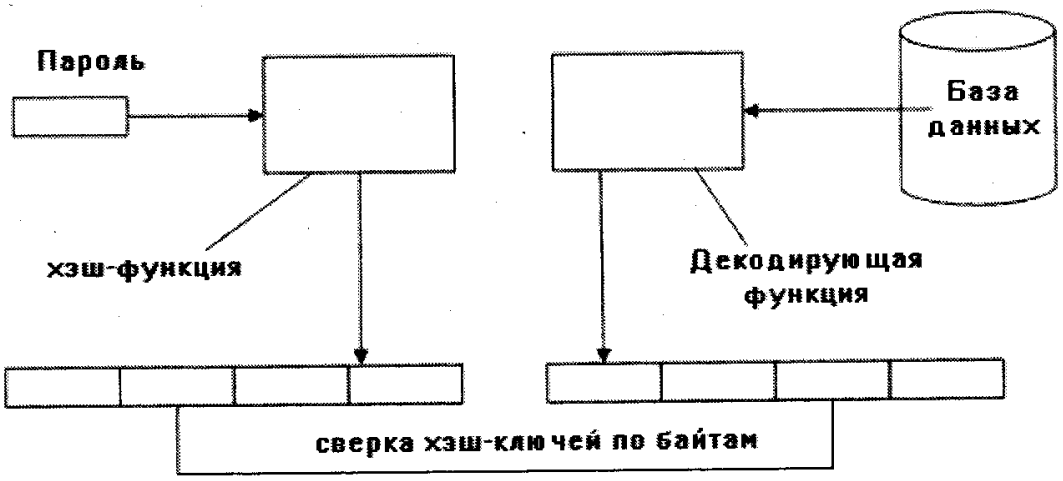


Рис. 6.8. *Схема сверки хэш-ключей паролей*

При сверке хэш-ключей у вас может возникнуть желание использовать, например, метод `compareTo()` класса `String`, как в следующем примере.

```
// пример использование метода compareTo()
// для сравнения зашифрованных паролей
public static void main(String[] args) {

// первый аргумент программы — пароль
if (args.length == 0) return;

// шифровка пароля пользователя
String encrypted = new String(PasswordEncrypter.
encrypt("ivík"));
String user_enc = new String(PasswordEncrypter.
encrypt(args[0]));

// операция сравнения
if (encrypted.compareTo(user_enc) == 0) {

System.out.println("Пароль введен верно.");
} else {
System.out.println("Пароль указан неправильно ");
}
}
```

Обратите внимание на то, что в программе мы сначала зашифровываем исходный пароль. Разумеется, в дальнейшем мы этого делать не будем, так как зашифрованный пароль возьмем из файла или базы данных.

Однако способ сравнения с использованием метода `compareTo()` имеет небольшую проблему: он предназначен для сравнения строк, фактическое представление которых зависит от применяемого вида кодирования (Unicode, ISOxxxx и т.д.).

Правильнее было бы сравнить зашифрованные ключи по байтам. Для этого в классе `MessageDigest` имеется метод `isEqual()`, который сравнивает два массива байтов и возвращает в качестве результата значение типа `boolean`.

```
// пример использования метода isEqual() класса MessageDigest
MessageDigest d = MessageDigest.getInstance("sha-1");

// сравнение байт за байтом
if (d.isEqual(encrypted.getBytes(),
user_enc.getBytes()) == true) {
System.out.println("Пароль задан правильно.");
}
```

Для того, чтобы сберечь усилия программистов, предложим собственный класс PasswordComparer, который будет сравнивать зашифрованные ключи.

```
// пример простого использования класса PasswordComparer
if (args.length == 0) return;
String userPassword = args[0];
String encrypted = new String(PasswordEncrypter.
encrypt("ivik"));

PasswordComparer pc = new PasswordComparer(encrypted);
if (pc.isEqual(userPassword) == true) {
System.out.println("Пароль задан правильно .");
}
```

Класс PasswordComparer будет очень простым. В конструкторе он примет уже зашифрованный ключ, который мы получили перед этим из файла или базы данных и который соответствует правильному паролю. Этот ключ будет сохранен в закрытой переменной, которую нельзя изменить или прочитать извне.

Далее класс вызывает метод isEqual(), который проводит сверку со значением, сохраненным в экземпляре класса.

```
// исходный пример класса для сравнения паролей
public class PasswordComparer {

private MessageDigest d = null;
private String encrypted = null;

private PasswordComparer() {
// конструктор по умолчанию
}

public PasswordComparer(String encrypted) {
this.encrypted = encrypted;
try {
d = MessageDigest.getInstance("sha-1");
} catch (NoSuchAlgorithmException nsae) {
}
} // PasswordComparer(String)

public boolean isEqual(String password) {
String encPassword = new
String(PasswordEncrypter.encrypt(password));
return d.isEqual(encrypted.getBytes(),
encPassword.getBytes());
```

```
} // isEqual(String).

} // PasswordComparer class
```

Метод `isEqual()` использует для получения хэш-ключа класс `EncryptPassword` из предыдущего примера.

Проблемы в применении указанного класса могут быть следующими: класс не позволяет задать тип используемого алгоритма (например, MD-5); в хэш-ключе могут содержаться непечатаемые символы или символы записи команды в несколько строк. В результате ключ нельзя будет записать в текстовый файл. Именно по этой причине хэш-ключи закодированы в символьный формат (например, при помощи алгоритма Base64), как уже рекомендовалось выше.

Было бы неплохо устранить приведенные выше проблемы. Достичь этого можно с помощью дополнительного конструктора. Конструктор будет принимать три параметра. Первым параметром будет значение хэш-ключа, вторым — значение типа `boolean`, определяющее, закодирован хэш-ключ при помощи алгоритма Base64 или нет, и последний, третий параметр будет задавать применяемый алгоритм.

```
// пример расширения возможностей класса при помощи конструктора
public class PasswordComparer {

private String value = null;
private MessageDigest d = null;

private PasswordComparer() {
// конструктор по умолчанию
}

// расширяющий возможности конструктор
public PasswordComparer(String value,
boolean decode,
String algorithm) {

if (decode == true) {

// декодирование хэш-ключа
try {
BASE64Decoder dec = new BASE64Decoder();
this.value = new String(dec.decodeBuffer(value));
dec = null;
} catch (IOException ioe) {
// игнорирование ошибок ввода/вывода
}
```

```
} else {
this.value = value;
} // if (decode == true)

// получение имени алгоритма преобразования
try {
d = MessageDigest.getInstance(algorithm);
} catch (NoSuchAlgorithmException nsae) {
// алгоритм задан неверно
}

} // PasswordComparer(String, boolean, String)

//установка параметров конструктора по умолчанию
public PasswordComparer(String encrypted) {
this(encrypted, false, "sha-1");
}

public PasswordComparer(String value, boolean decode) {
this(value, decode, "sha-1");
}

// остальной код класса опущен

} // PasswordComparer class
```

Чтобы данный класс не мог использоваться для определения пароля «силой» (brute force attack), мы можем встроить в класс механизм, позволяющий использовать экземпляр класса ровно столько раз, сколько мы укажем. Так мы ограничим максимально допустимое число неудачных попыток, для чего следует добавить несколько вспомогательных методов и внести некоторые изменения в метод `isEqual()`.

```
// пример добавления дополнительных методов
public final class PasswordComparer {

private String value = null;
private MessageDigest d = null;

private int failedCount = 0; // счетчик неудачных попыток
private static int maxFailedCount = 3; // макс. число
неудачных попыток

// конструктор опущен, добавьте его из предыдущего примера

// возвращает число неудачных попыток
```

```
public int getFailedCount() {
return failedCount;
}

// увеличивает число неудачных попыток
private synchronized void incrementFailedCount() {
failedCount++;
}

// сбрасывает значение счетчика неудачных попыток на 0.
private synchronized void resetFailedCount() {
failedCount = 0;
}

public synchronized boolean isEqual(String password) {
// тело метода опущено, добавьте его из предыдущих
примеров
}

} // PasswordComparer class
```

Прежде всего добавим закрытые переменные класса и обрабатывающие их методы. Первая переменная `failedCount` будет хранить число неудачных попыток. Вторая статическая переменная `maxFailedCount` служит для задания максимально допустимого количества неудачных попыток (в данном случае оно равно трем).

Метод `incrementFailedCount()` служит для увеличения максимально допустимого числа неудачных попыток ввода пароля. Метод `resetFailedCount()` имеет обратное действие — позволяет сбросить это число на ноль. Обратите внимание, что класс `PasswordComparer` обозначен как окончательный (спецификатор `final`). Это сделано для того, чтобы предотвратить добавление класса-потомка, действия которого выходят за определенные нами рамки.

Также хотим обратить ваше внимание на то, что оба дополнительных метода, устанавливающих значение максимального числа неудачных попыток, обозначены ключевым словом `synchronized`. Это исключает вызов данного метода из нескольких задач одновременно, за счет чего можно было бы обойти ограничение числа неудачных попыток.

Конечный этап работы над созданием класса для проверки паролей — реализация механизма подсчета количества неудачных попыток в методе `isEqual()`.

```
public final class PasswordComparer {

// декларирование переменных, конструкторов
// и дополнительных методов
// опущено, возьмите их из предыдущих примеров

public synchronized boolean isEqual(String password)
throws IllegalAccessException {

boolean success = false;

// генерация исключения, если достигнуто максимальное
// число неудачных попыток ввода пароля
if (getFailedCount() == maxFailedCount) {
throw
new IllegalAccessException(
"Достигнуто максимальное число неудачных попыток ввода
пароля.");
}

// сравнение хэш-ключей
try {
String encPassword =
new String(PasswordEncrypter.encrypt(password,
d.getAlgorithm()));
success = d.isEqual(value.getBytes(),
encPassword.getBytes());
} catch (IllegalArgumentException iae) {
// игнорирование исключения
}

// в зависимости от результата проверки хэш-ключей
// увеличивается значение счетчика неудачных попыток
// или он сбрасывается на ноль
if (success == true) {
resetFailedCount();
} else {
incrementFailedCount();
}

// возвращает результат сравнения
return success;
} // isEqual(String)

} // PasswordComparer class
```

В начале метода `isEqual()` проверяется, достигнуто ли максимальное число неудачных попыток. Если да, то выбрасывается исключение `IllegalAccessException`, и выполнение метода на этом прекращается.

В середине метода производится уже описанное нами побайтовое сравнение хэш-ключей, и результат проверки записывается в локальную переменную `success`.

В зависимости от результата проверки хэш-ключей увеличивается значение счетчика неудачных попыток или он сбрасывается на ноль. Только потом возвращается результат проверки в возвращаемом значении.

Шифрование строки

Для шифровки произвольного текста существует много алгоритмов, одним из которых является, например, алгоритм DES. В JDK 1.4 имеется пакет `javax.crypto`, который позволяет шифровать и дешифровать тексты при помощи данного алгоритма.

Следующий класс показывает, как могут быть реализованы функции шифровки и дешифровки текстов с использованием данного пакета.

```
//класс для реализации функций шифровки и дешифровки
import javax.crypto.*;
import java.io.*;
import sun.misc.*;
import  java.security.*;
public class DesEncrypter {
Cipher ecipher = null;
Cipher dcipher = null;

public DesEncrypter(SecretKey key) {
try {
ecipher = Cipher.getInstance("DES");
dcipher = Cipher.getInstance("DES");

ecipher.init(Cipher.ENCRYPT_MODE, key);
dcipher.init(Cipher.DECRYPT_MODE, key);
} catch (NoSuchPaddingException nspe) {
// игнорируем исключение
} catch (NoSuchAlgorithmException nsae) {
// игнорируем исключение
} catch (InvalidKeyException ike) {
// игнорируем исключение
}
} // DesEncrypter(SecretKey)
public String encrypt(String str) {
```

```
try {
byte[] utf8 = str.getBytes("utf8");
byte[] enc = ecipher.doFinal(utf8);
return new BASE64Encoder().encode(enc);
} catch (UnsupportedEncodingException uee) {
// игнорируем исключение
} catch (BadPaddingException bpe) {
// игнорируем исключение
} catch (IllegalBlockSizeException ibse) {
// игнорируем исключение
}
return null;
} // encrypt(String)

public String decrypt(String str) {
try {
byte[] dec = new BASE64Decoder().decodeBuffer(str);
byte[] utf8 = dcipher.doFinal(dec);
return new String(utf8, "utf8");
} catch (IOException ioe) {
// игнорируем исключение
} catch (BadPaddingException bpe) {
// игнорируем исключение
} catch (IllegalBlockSizeException ibse) {
// игнорируем исключение
}
return null;
}//decrypt(String)

} // DesEncrypter class
```

Класс `DesEncrypter` создает два экземпляра класса `Cipher` из пакета `javax.crypto`. Оба настроены в конструкторе для использования алгоритма DES, первый из них для шифрования, другой — для дешифрования текста. Классу передается ключ типа `SecretKey` — ключ для алгоритма шифрования.

Класс имеет два метода, которые обеспечивают процесс шифровки/дешифровки. Метод `encrypt()` принимает в своем единственном параметре строку, которую необходимо зашифровать. Строка сначала преобразуется в массив байтов, а затем шифруется при помощи метода `doFinal()` класса `Cipher`. За расшифровку отвечает метод `decrypt()`.

Использование данного класса также очень простое, см. код ниже.

```
import javax.crypto.*;
import java.io.*;
```

```
import sun.misc.*;
import  java.security.*;

// пример шифровки произвольного текста

public class DesEncrypterTest {

public static void main(String[] args) {

try {
SecretKey key = KeyGenerator.getInstance("DES").generateKey();

DesEncrypter encrypter = new DesEncrypter(key);
String encrypted = encrypter.encrypt("Проверка");
String decrypted = encrypter.decrypt(encrypted);

System.out.println("Расшифровано: " + decrypted);

} catch (NoSuchAlgorithmException nsae)
{// игнорирование исключений
}

} // main(String[])

} // DesEncrypterTest class
```

Сначала создается ключ шифрования с помощью класса `KeyGenerator`. Затем предадим его экземпляру нашего класса `DesEncrypter` в процессе его формирования. Далее шифруем слово «Проверка» при помощи метода `encrypt()`, а зашифрованный текст передаем методу `decrypt()` для дешифрования. Результат дешифровки отображаем на консоли. Если все нормально, то результат должен быть следующим:

```
Расшифровано: Проверка
```

Смысл всего кода очевиден из первой строки блока `try`, где создается экземпляр класса `KeyGenerator` — ключ для шифрования.

6.10. Работа с потоками

Переведение задачи в спящий режим на определенный срок

Если вы хотите на некоторое время передать управление системе и перевести вашу программу в спящий режим, используйте статический метод `sleep()` класса `Thread`. В качестве параметра он принимает количество

миллисекунд (тип данных — long). При необходимости вы можете также вызывать данный метод с двумя параметрами, где второй параметр соответствует наносекундам.

```
// пятисекундная пауза в программе
long mili = 5000L;
Thread.sleep(mili);
```

Атомарные операции

В современных версиях Java чтение или запись 32-битной и меньшей по емкости переменной выполняется как атомарная операция. **Атомарная операция — эта элементарная (наименьшая по количеству действий) операция, особенность которой состоит в том, что она не может быть прервана.**

Если вы используете атомарные операции в многопоточных приложениях, они не вызовут конфликтов с другими потоками (атомарные операции еще называют thread-safe — безопасные в отношении других потоков) и их не нужно синхронизировать. Это удобно, так как синхронизация требует определенных системных ресурсов, ее нужно использовать только в случае необходимости.

Приведем пример атомарных операций:

```
// пример атомарных операций
public class ThreadSafePoint {
private int x; // координата x
private int y; // координата y
public int getX() {
return this.x;
}
public void setX(int x) {
this.x = x;
}
public int getY() {
return this.y;
}
public void setY(int y) {
this.y = y;
}
} // ThreadSafePoint
```

Проанализируем отдельные части приведенного выше кода. Метод setX(), служащий для записи координаты точки по оси x, выполняет только одно действие, а именно запись числа во внутреннюю перемен-

ную. Поэтому метод считается атомарным. Точно так же другие методы класса являются атомарными:

```
public void getX() {
return this.x; // только одна операция
}
```

Попробуйте догадаться, является ли операция инкремента определенной переменной атомарной.

```
public void increment() {
++this.value;
}
```

Если вы считаете, что это атомарная операция, то ошибаетесь. Вся операция, несмотря на то, что она записана всего на одной строке, содержит три действия:

- Чтение текущего значения переменной `value`.
- Увеличение значения переменной в памяти.
- Запись нового значения обратно в переменную `value`.

Поскольку здесь речь идет больше чем об одной инструкции, операция может быть прервана и управление может перейти к другому потоку. В большинстве многопоточных приложений это может вызвать весьма серьезные затруднения. В Java создание многопоточных приложений имеет следующие особенности:

- Ошибки, касающиеся потоков, трудно выявить, они требуют тщательного анализа кода.
- Еще хуже то, что операция может быть атомарной на одном процессоре и не атомарной на другом. Это может быть определено только путем проверки на разных процессорах или виртуальных машинах.

Чтобы выполнять определенные методы как атомарные, существует ключевое слово `synchronized`, которое дает сигнал Java выполнять метод, к которому оно относится, как атомарную операцию.

```
public synchronized void increment() {
++this.value;
}
```

Поэтому всегда, когда вы пишете какой-нибудь метод, который предполагается использовать в многопоточном приложении, уделяйте особое внимание простым операторам, которые могут быть и не атомарными.

Предотвращение неожиданного выхода из программы

Обычно программа работает от начала и до конца, когда достигается определенный результат. Однако она может быть закончена и до выполнения всех предусмотренных в ней команд: из-за сбоя в системе, действий пользователя или по другой причине.

Чтобы вы имели возможность отреагировать на попытку закрытия, Java предлагает создать специальный поток, который будет вызван и выполнен в момент завершения работы приложения. Этот поток регистрируется вызовом метода класса `Runtime`:

```
Runtime.addShutdownHook(Thread hook);
```

В начале программы следует передать данному методу инициализированный класс, который является производным от `Thread`. Дальше ни о чем беспокоиться не придется, при завершении работы программы (любом) будет выполнен метод `run()` этого класса, реализующего поток завершения.

В примере мы создаем открытый класс `ShutDownHook` с методом `main()` и закрытый класс `ShutDownThread`, который будет обрабатывать команду завершения работы программы.

```
class ShutDownHook {

public ShutDownHook() {
}

public static void main(String[] args) {

Runtime rtm = Runtime.getRuntime();
rtm.addShutdownHook(new ShutDownThread());
rtm = null;

try {
System.out.println("Программа будет ждать 100 секунд
                    перед тем, как закрыться.");
System.out.println("Нажатие клавиш Ctrl+C не завершит
                    работу программы,");
System.out.println("а вызовет переход к следующей
                    ее части.");
Thread.sleep(100000);
} catch (InterruptedException ie) {
System.out.println(ie.getMessage());
}
}
}
```

```
class ShutDownThread extends Thread {

public void run() {
System.out.println("\tПрограмма завершена.");
System.out.println("\tНо можно еще что-то изменить.");
}
}
```

При использовании потока, вызываемого при завершении работы программы, уделяйте особое внимание коду, который находится в теле метода `run()`. Старайтесь, по возможности, защитить его от внешнего доступа.

В результате выполнения программы на экране возникнет следующая запись:

```
Программа будет ждать 100 секунд перед тем, как закрыться
Нажатие клавиш Ctrl+C не завершит работу программы,
а вызовет переход к следующей ее части.
Программа завершена.
Но можно еще что-то изменить.
```

Запрет повторной регистрации пользователей

Представьте себе, что одной из задач вашего приложения является исключить возможность регистрации двух одинаковых пользователей одновременно. Такое приложение наверняка будет иметь класс, который регистрирует пользователей, обрабатывая список в базе данных.

```
public class Login {

public void makeLogin() {
// операции регистрации пользователей
} // makeLogin()

} // Login class
```

Несмотря на то, что в методе `makeLogin()` вы выполняете процедуру регистрации, может возникнуть ситуация, когда одновременно два пользователя захотят зарегистрироваться под одним и тем же именем. Типичный пример — интернет-приложения и сервлеты.

Для того, чтобы убедиться, что метод `makeLogin()` не запущен в один и тот же момент дважды (или несколько раз) различными элементами вашего приложения, нужно обеспечить взаимное исключение (mutual exclusion). Это значит, что данный метод не может быть вызван, если не завершен предыдущий его вызов. То есть метод будет возможно вызвать только поочередно.

```
public class Login {

public synchronized void makeLogin() {
// операции регистрации пользователей
} // makeLogin()

} // Login class
```

Простым решением в данном случае будет использование синхронизации потоков. В представленном примере с интернет-приложением или сервлетами каждое отдельное требование зарегистрироваться (метод `makeLogin`) может запускаться в самостоятельном потоке, то есть существует вероятность многократного запуска кода метода. Используя синхронизацию, вы исключите эту возможность.

6.11. Использование обработчика протоколов (log handler)

По различным причинам, например для наблюдения за работой приложения, имеет смысл записывать сведения о внутреннем состоянии программы. Раньше эта проблема решалась программистами самостоятельно, но с версии JDK 1.4 Java имеет стандартное средство для реализации такого рода задач.

В JDK 1.4 включен пакет `java.util.logging` (Logging API), который, кроме всего прочего, имеет собственные обработчики протоколов, которые в значительной степени облегчают процесс написания программ с файлами протокола. Logging API предоставляет мощные средства для создания «снимков» приложений и записи информации из них в файл.

Но давайте уже перейдем к практике. Рассмотрите следующий пример:

```
Logger logger =
        Logger.getLogger("rus.lifebeing.myapp.computing");
logger.setLevel(Level.ALL);
```

Здесь создается экземпляр класса `Logger` (из пакета `java.util.logging`). Для этого вызывается стандартный метод `getLogger` данного класса, которому в параметре передается название (путь) протокола. Это название строится в соответствии с путем к протоколу (каждая точка соответствует поддиректории), что, с одной стороны, позволяет работать с несколькими протоколами одновременно (каждый для своей цели), с другой — может вызвать конфликты имен протоколов в различных приложениях. Вызов метода `setLevel` с параметром `Level.ALL` указывает

на то, что мы будем протоколировать все события независимо от степени их важности (severity).

После создания объекта типа `Logger` достаточно вызвать один из поддерживаемых им методов, которые занесут программное событие в протокол. Приведенный ниже код показывает несколько вариантов работы с Logging API, которые помогут вам понять суть протоколирования.

```
// Пример записи протокола
logger.info("Сообщение");
logger.warning("Предупреждение");
try {
int i = 0 / 0;
} catch (Exception e) {
logger.log(Level.WARNING, "Деление на ноль.", e);
}
```

Если вы хотите сформировать собственный обработчик протоколов, для этого необходимо создать свой класс на основе абстрактного класса `Handler`. Затем нужно добавить собственную реализацию трех самых важных методов: `close()`, `flush()` и `publish()`. С помощью этих методов вы сможете получить всю необходимую информацию для создания протокола.

Кроме того, вы, можете реализовать и другие методы, которые вам требуются — вам наверняка понадобится конструктор, в котором вы будете передавать своему классу `Handler` нужные параметры (название обрабатываемого файла, базы данных и т.п.).

Одним из самых важных методов является `close()`. Он служит для закрытия всех элементов программы, которые были инициализированы в конструкторе. Если данный метод не был запущен, существует вероятность потери данных.

Связанный с методом `close()`, метод `flush()` используется, если нужно записать всю полученную информацию о работе программы на определенный носитель (обычно в файл на жестком диске). Однако в некоторых ситуациях нет необходимости в собственной реализации тела данного метода. В этом случае желательно создать хотя бы пустой метод `flush()`.

Последний из рассматриваемых нами методов — это метод `publish()`, реализация которого является наиболее трудоемкой. Он вызывается каждый раз, когда программное событие необходимо задокументировать. Метод принимает только один параметр — объект типа `LogRecord`, содержащий все нужные сведения. Метод `publish()` последовательно вызывает все методы типа `get` и записывает на диск возвращаемые ими значения. Класс `LogRecord`, например, содержит свойство `level`, которое устанавливает степень важности события. Это позволит отфильтровать события, которые вы не считаете важными.

ГЛАВА 7.

РАБОТА С ФАЙЛАМИ В JAVA

7.1. Работа с файлами и папками в Java

В Java работа с файлами и папками организована на высоком уровне, начиная с самой первой версии JDK. Особенность Java заключается в поддержке нескольких операционных систем, что с точки зрения операций с файловой системой является большим преимуществом: на Java легко написать программу, которая копирует файл из раздела диска, отформатированного одной файловой системой, в раздел с другой файловой системой.

Как всегда, для работы с файловой системой Java имеет собственную объектную модель, в которой каждая папка (каталог) представлена как файл. Для большинства операций с файлами и папками достаточно будет использовать класс `File` пакета `java.io`.

В этой главе мы не только уделим много внимания работе с файловой системой, но и разберем весьма важные вопросы ввода-вывода (I/O). Поскольку операции ввода-вывода чаще всего используются при работе с файлами, удобно будет рассматривать эти две задачи вместе. Примеры из этой главы вы сможете в дальнейшем использовать при написании серверных приложений или при работе с файлами в интернете.

Операции ввода-вывода на Java реализованы через потоки (stream). Пакет `java.io` содержит множество классов, имена которых заканчиваются на `Stream`, например `BufferedInputStream`, `DataInputStream` и т. п. Эти классы обслуживают отдельные разновидности потоков: `BufferedInputStream` используется для последовательного чтения потока, `DataInputStream` — для чтения потока данных. В именах классов часто встречаются слова Input и Output. Классы со словом Input в имени «считывают» (принимают) данные, а классы со словом Output «записывают» (передают) их. Пакет `java.io` также содержит коллекцию абстрактных классов, которые вы можете использовать для написания собственной реализации операций ввода-вывода.

Java поддерживает три базовые технологии для работы с XML-файлами. Первая из них соответствует спецификации DOM (Document Object Model). Согласно ей XML-документ обобщенно представляется как древовидная структура, состоящая из узлов, атрибутов и секций. Спецификация DOM создана консорциумом World Wide Web Consortium (W3C), разрабатывающим интернет-стандарты, поэтому пакет для работы с XML-файлами носит название `org.w3c.dom`.

Вторая технология, применяемая для работы с XML, — это интерфейс SAX (Simple API for XML Parsing). SAX — это интерфейс последовательного чтения XML-документа специальной программой-парсером (parser), которая в процессе чтения файла генерирует события, соответствующие изменению структуры XML-файла (переход к следующему разделу, узлу или секции). Классы для работы с интерфейсом SAX находятся в пакете `org.xml.sax` и его подпакетах.

Третья технология — это трансформация согласно стандарту XML Transformation, описание которого можно найти по адресу `http://www.w3.org/TR/1999/REC-xslt-19991116`. Трансформация — это преобразование XML-документа к другому виду (например, HTML) с помощью специального языка XSLT (Extended Stylesheet Transformation). Классы, реализующие трансформацию, находятся в пакете `javax.xml.transform`.

7.2. Пути к файлам и папкам

Полный путь к файлу

В разных операционных системах пути к файлам и папкам записываются по-разному. В Unix-системах в качестве разделителя используется косая черта — слэш (/), в Windows — обратный слэш (\). Для того, чтобы избавить программиста от необходимости писать метод, определяющий платформу, на которой выполняется программа, и соответственно разделитель путей, в языке Java предусмотрена статическая переменная `File.separator`:

```
// пример формирования строки, содержащей путь к файлу
/*
** строка path будет содержать
** в Windows: root\folder\file.txt
** в UNIX: root/folder/file.txt
*/
```

```
String path = "root" + File.separator + "folder" +
              File.separator + "file.txt";
```

Файл или папка?

В классе `File` есть метод `isDirectory()`, позволяющий определить, является ли заданный путь путем к файлу или папке:

```
File spec = new File("folder");
boolean isDir = spec.isDirectory(); // файл или папка

if (isDir == true) {
 System.out.println("папка "); // папка
} else {
 System.out.println("файл"); // файл
}
```

Получение абсолютного пути к файлу

Метод `getPath()` класса `File` возвращает тот путь, который был указан при создании экземпляра класса `File`. Это может быть как абсолютный путь, то есть путь, отсчитывающийся от корневой папки диска, так и относительный (по отношению к текущей папке). Абсолютный путь к данному файлу можно получить вызовом метода `getAbsolutePath()` класса `File`:

```
File spec = new File("file.txt");
System.out.println("относительный путь: " +
                   spec.getPath());
System.out.println("абсолютный путь: "
                  + spec.getAbsolutePath());
```

Этот фрагмент кода выведет следующие строки:

```
относительный путь: file.txt
абсолютный путь: Y:\\file.txt
```

Метод `getAbsoluteFile()` класса `File` возвращает новый экземпляр класса `File`, для которого метод `getPath()` будет возвращать абсолютный путь:

```
File spec1 = new File("file.txt");
File spec2 = spec1.getAbsoluteFile();
System.out.println("путь 1: " + spec1.getPath());
System.out.println("путь 2: " + spec2.getPath());
```

Этот фрагмент кода выведет следующие строки:

```
путь 1: file.txt
путь 2: Y:\\file.txt
```

Пути разные — файл один?

Бывает, что две различных строки пути указывают на самом деле на один и тот же файл. На платформе Windows это может произойти, когда относительный путь к файлу содержит специальные имена папок «.» и «..», в Unix-системах — из-за символических ссылок. Поэтому существует однозначно определяемая каноническая форма пути.

Метод `getCanonicalPath()` класса `File` возвращает канонический путь в виде строки, а метод `getCanonicalFile()` возвращает новый экземпляр класса `File`, в котором путь хранится в канонической форме. Если вам нужно проверить, являются ли файлы, заданные двумя разными путями, одним и тем же файлом, вы можете сравнить их канонические объекты `File` с помощью метода `equals()`:

```
boolean result;
File f1 = new File("./file.txt"); // абсолютный путь:
                                  // "." - это текущая папка
File f2 = new File("file.txt"); // относительный путь
result = f1.equals(f2);         // вернет false
if (!result) System.out.println("Файлы разные");

try {
 f1 = f1.getCanonicalFile();
 f2 = f2.getCanonicalFile();
} catch (IOException ioe) {}
result = f1.equals(f2);         // вернет true
if (result == true)
System.out.println("Файлы одинаковые");
```

Для файлов `f1` и `f2` вы могли бы сравнить строки, возвращенные методом `File.getCanonicalPath()`: в классе `String` тоже существует метод `equals()`.

Извлечение пути к родительской папке

Чтобы получить из пути к файлу путь к папке, в которой находится файл, воспользуйтесь методами `getParent()` или `getParentFile()` класса `File`. Будьте внимательны: оба метода могут возвратить значение `null`, если переданный им путь к файлу будет относительным либо если путь состоит только из корневой папки диска.

```
// указываем файл по относительному пути
File f = new File("file.txt");
String parentPath = f.getParent(); // ВНИМАНИЕ: возвращает null
File parentFolder = f.getParentFile();
                                   // ВНИМАНИЕ: возвращает null
```

```
// указываем файл по абсолютному пути
f = new File("c:\\java\\file.txt");
parentPath = f.getParent();         // вернет c:\java
parentFolder = f.getParentFile();  // вернет c:\java

// аргументом может быть путь к папке
parentPath = parentFolder.getParent();         // вернет c:\
parentFolder = parentFolder.getParentFile(); // вернет c:\

// аргументом может быть путь к корневой папке
parentPath = parentFolder.getParent();
                              // ВНИМАНИЕ: возвращает null
parentFolder = parentFolder.getParentFile();
                              // ВНИМАНИЕ: возвращает null
```

Преобразование путей в URL и обратно

Преобразование между путями и URL очень просто: для преобразования пути в URL служит метод `toURL()` класса `File`, для обратного преобразования — метод `getFile()` класса `URL` из пакета `java.net`:

```
import java.io.*
import java.net.*

File f = new File("file.txt");
URL url = null;
try {
 url = f.toURL(); // file://y:/ file.txt
} catch (MalformedURLException mue) {}

f = new File(url.getFile()); // y:\file.txt
```

7.3. Действия над файлами и папками

7.3.1. Проверка существования файла

Чтобы определить, существует ли тот или иной файл, вы можете использовать метод `exists()` класса `File`. Этот метод возвращает значение `true`, если файл существует, иначе — `false`. Использование этого метода может вызвать исключение `SecurityException`, если у вас недостаточно прав на чтение файла:

```
public static boolean fileExists(String fileName) {
 boolean result = false;
```

```
 File f = null; // файл
 try {
    f = new File(fileName);
    result = f.exists();
 } catch (SecurityException se) {
    // если возникло исключение, значит, понадобились права,
    // значит, файл существует
    result = true;
 } finally {
    if (f != null) f = null;
 }
 return result;
} // fileExists(String)
```

7.3.2. Создание и удаление файла

Почти каждая программа работает с файлами — читает или записывает данные. Наверняка и вам когда-нибудь понадобится создать файл, чтобы в дальнейшем использовать его в приложении. Для этого служит метод `createNewFile()`, возвращающий значение `true` в случае успешного создания файла `false`, если файл уже существует.

Независимо от того, существует ли уже на диске файл, который вы пытаетесь создать методом `createNewFile()`, вызов этого метода создает в программе объект `File`, через который вы можете работать с файлом.

```
try {
 File f = new File("file.txt");
 boolean exists = f.createNewFile();
 if (exists) {
    // был создан новый файл
 } else {
    // операции с существующим файлом
 }
} catch (IOException ioe) {
 // обработка исключений
}
```

Метод `delete()` класса File, служащий для удаления файла, не требует комментариев:

```
File fileDel = new File("file.txt");
boolean deleted = fileDel.delete();
if (!deleted) {
 System.out.println("Файл не удален.");
}
```

7.3.3. Временные файлы

Нередко в программе нужно создать временный файл, чтобы в дальнейшем использовать его в качестве буфера или других целей. Класс `File` предлагает два варианта статического метода `createTempFile()`, различающиеся количеством параметров.

Метод `createTempFile()`, вызываемый с двумя параметрами, служит для создания файла в общей для всех программ папке для временных файлов. Размещение этой папки зависит от настроек операционной системы. Как правило, в Windows это папка `c:\temp`, в Unix-системах такой папкой, скорее всего, будет являться `/tmp` или `/var/tmp`.

Если вызывать метод `createTempFile()` с тремя параметрами, то в третьем параметре можно прямо указать папку, где будет создан временный файл.

В первом параметре методу передается префикс имени файла (минимум три буквы), во втором — его расширение. Если второй параметр имеет значение `null`, то временный файл будет иметь расширение `*.tmp`.

```
// пример создания временного файла
public static File createTempFile(String prefix, String
suffix) {
 File f = null;
 try {
    f = File.createTempFile(prefix, suffix);
 } catch (IOException ioe) {}
 return f;
} // createTempFile(String, String)
```

Временный файл, как и любой другой, можно удалить методом `delete()` класса `File`. Но удобнее сделать так, чтобы он удалялся сам по завершении работы программы. Для этого в классе `File` существует метод `deleteOnExit()`, добавляющий файл в список файлов, которые нужно удалить при выходе из программы. После этого удаление файла будет заботой не вашей, а виртуальной машины Java.

```
// пример автоматически удаляемого временного файла
public static File createTempFile(String prefix, String
suffix) {

 File f = null;
 try {
    f = File.createTempFile(prefix, suffix);
    f.deleteOnExit();
 } catch (IOException ioe) {
    // обработка исключения
```

```
    } catch (SecurityException se) {
        // обработка исключения
    }
    return f;
  } // createTempFile(String, String)
```

Метод `deleteOnExit()` используйте с особой осторожностью, так как после его вызова у вас уже не будет никакой возможности отменить удаление. Список файлов для удаления нельзя ни просмотреть, ни редактировать. Файлы из этого списка виртуальная машина Java удаляет безвозвратно и не запрашивая подтверждения сразу после завершения работы программы, которое могло быть и аварийным (в результате неперехваченного исключения). Таким образом вы можете потерять все промежуточные результаты работы программы.

7.3.4. Просмотр свойств файла

Размер файла

Размер файла в байтах возвращает метод `length()` класса `File`. Возвращаемое значение имеет тип `long`:

```
File f = new File("file.txt");
long l = f.length(); // длина файла в байтах
```

Если файл не существует, метод `length()` вернет значение `0L`, то есть этот метод не позволяет отличить несуществующий файл от пустого.

Время последнего изменения

Для определения времени изменения файла предназначен метод `lastModified()` класса `File`. Он возвращает значение типа `long`, представляющее собой количество миллисекунд, прошедших с 1 января 1970 года 0:00:00 GMT до момента изменения файла. Это число вы должны преобразовать в тип `Date`, из которого уже извлечете все необходимые сведения:

```
File aFile = new File("c:\\windows\\notepad.exe");
long modified = aFile.lastModified();
Date d = new Date(modified);
```

Время последнего изменения можно установить программно с помощью метода `setLastModified()` класса `File`. Аргументом этого метода является количество миллисекунд между 1 января 1970 года 0:00:00 GMT и желаемым временем:

```
public static void touch(String filePath) {

 // определяем текущее системное время
 long timeNow = System.currentTimeMillis();
 File f = new File(filePath);
 // устанавливаем текущее время в качестве времени
 // последнего изменения
 boolean touched = f.setLastModified(timeNow);
 if (!touched) {
   System.err.println("Не удалось установить время: " +
filePath);
 }
} // touch(String)
```

7.3.5. Переименование файла

Для переименования файла необходимо создать два экземпляра класса `File`: первый — для существующего файла, второй — для нового, который будет отличаться от старого именем. Действие по переименованию выполняет метод `renameTo()`, аргументом которого является объект типа `File`, соответствующий файлу с новым именем. В случае успешного переименования метод `renameTo()` возвращает значение `true`, иначе — `false`:

```
File oldFile = new File("old.txt"); // существующий файл
File newFile = new File("new.txt"); // имя нового файла

// переименовываем файл
boolean renamed = oldFile.renameTo(newFile);
if (!renamed) {
 System.out.println("Файл не был переименован.")
}
```

7.3.6. Перемещение файла

Метод переименования файла `renameTo()` легко можно использовать и для перемещения файла. Для этого достаточно, чтобы файл (то есть соответствующий ему объект класса `File`), именем которого мы называем исходный файл, находился в другой папке. Папку можно указать, если вызвать перегруженный метод `renameTo()` с двумя параметрами. Во втором параметре типа `File` передается папка назначения:

```
File origin = new File("origin.txt"); // исходный файл
File folder = new File("c:\\temp"); // папка назначения
```

```
// создание объекта для файла с тем же именем
// в целевой папке
File toname = new File(folder, origin.getName());

// перемещение файла в целевую папку
boolean moved = origin.renameTo(toname);
if (!moved) {
 System.out.println("Файл не был перемещен.");
}
```

7.3.7. Рекурсивное удаление папок

Если необходимо удалить определенную папку, нужно предварительно удалить все хранящиеся в ней файлы и папки. Именно поэтому для удаления папки удобно применять рекурсию: метод получает список файлов и подпапок заданной папки и последовательно удаляет его элементы; если текущий элемент является папкой, то метод вызывает сам себя для этой подпапки и т. д.

Учтите, что для удаления папки требуются права доступа ко всей иерархии ее подпапок на удаление.

Листинг 7.1. Рекурсивное удаление папок и файлов

```
public static void deleteDir(String dirPath) {

 File walkDir = new File(dirPath); // удаляемая папка

 String[] dirList = walkDir.list(); // список элементов в папке

 for (int i = 0; i < dirList.length; i++) {

    File f = new File(dirList[i]);
    if (f.isDirectory()) {
       // если текущий элемент - папка
       deleteDir(f.getPath());
    }
    // текущий элемент - файл или уже пустая папка
    f.delete();
 }
 walkDir.delete();

} // deleteDir(String)
```

7.4. Чтение и запись файлов

Чтение текстового файла

Чтение содержимого текстового файла лучше всего организовать следующим образом: вы создаете экземпляр класса-«читателя» `FileReader`. Этот экземпляр вы потом передаете объекту типа `BufferedReader`, который последовательно читает строки файла в цикле `while` с помощью метода `readLine()`.

```
try {
 BufferedReader in = new BufferedReader(new
                     FileReader("file.txt"));

 // построчное чтение файла
 String str;
 while ((str = in.readLine()) != null) {
   // обработка полученной строки
 }
 in.close();
} catch (IOException e) {
 System.out.println("Ошибка при обработке файла.");
}
```

Если файл содержит символы кириллицы, нужно немного изменить команду создания объекта класса `BufferedReader`:

```
BufferedReader in = new BufferedReader(new
                    InputStreamReader(new
                    FileInputStream("file.txt"), "8859_5"));
```

Сначала вы создаете класс для чтения потока ввода из файла `FileInputStream` и затем передаете ему, кроме самого пути к файлу, формат кодирования. Русские символы описаны в формате кодирования ISO-8859-5 (или в Windows в CP-1251), поэтому используйте строковый параметр «8859_5».

Аналогично, если вы будете читать сведения из файла с форматом кодирования UTF-8, вам будет достаточно вызвать метод с параметром «UTF8».

Чтение двоичного файла

Для чтения файла байт за байтом служит класс `InputStream`. Покажем, как прочитать содержимое двоичного файла в байтовый массив.

Листинг 7.2. Пример чтения двоичного файла

```
/* если будете копировать код в свою программу,
** не забудьте снабдить его блоком try...catch
** для перехвата исключений IOException */

File file = new File("file.bin");
InputStream is = new FileInputStream(file);
// получаем размер файла
long length = file.length();

/*
** Нельзя создать байтовый массив с количеством
** элементов большим, чем значение
** Integer.MAX_VALUE. Если размер файла в байтах больше
** этого значения, файл нельзя прочитать
*/
if (length > Integer.MAX_VALUE) {
 /* если размер файла слишком большой для чтения,
 ** будет сгенерировано исключение. */
 throw new IOException("Файл " + file.getName() +
                       "слишком длинный!");
}

// создание массива для содержимого файла
byte[] bytes = new byte[(int)length];

// чтение байтов
int offset = 0;
int numRead = 0;
while (offset < bytes.length &&
    (numRead=is.read(bytes, offset,
                     bytes.length-offset)) >= 0) {
 offset += numRead;
}

// Проверка, пройден ли файл до конца
if (offset < bytes.length) {
 throw new IOException("Не удалось прочитать файл " +
    file.getName() + " целиком.");
}
// закрытие потока чтения из файла
is.close();
```

В примере обратите внимание на то, что нужно избегать чтения файлов, размер которых больше, чем значение `Integer.MAX_VALUE`, так как это предельное значение размера байтового массива. На практике двоичные файлы читаются по частям небольшого размера (например, по 2, 4 или 8 Кб).

Запись в текстовый файл

Для последовательной записи данных в текстовый файл создайте экземпляр класса `BufferedWriter` и передайте ему объект `FileWriter`. По окончании записи не забудьте закрыть поток вывода методом `close()`:

```
try {
 String fileName = "file.txt";
 BufferedWriter out = new BufferedWriter(new
FileWriter(fileName));
 out.write("Hello World!");
 out.close();
} catch (IOException e) {
 System.out.println("Ошибка в процессе записи файла.");
}
```

Если файл `file.txt` еще не существует, он будет создан, а если существует — перезаписан.

Чтобы дописать строки к существующему файлу, создавайте объект `FileWriter` другим конструктором, передав ему `true` в качестве второго аргумента:

```
try {
 String fileName = "append.txt";
 BufferedWriter out = new BufferedWriter(new
                          FileWriter(fileName, true));
 out.write("We shall overcome!");
 out.close();
} catch (IOException e) {
 System.out.println("Ошибка в процессе записи файла.");
}
```

Чтобы можно было записывать в файл символы кириллицы, вызывайте конструктор `BufferedWriter` с такими же аргументами, как конструктор `BufferedReader` из примера «Чтение текстового файла»:

```
BufferedWriter out = new BufferedWriter(new
OutputStreamWriter(new
 FileOutputStream("file.txt"), "8859_5"));
```

Чтение и запись файла в произвольном порядке

Для того, чтобы читать или записывать данные из любого места файла, нужно создать объект типа `RandomAccessFile` и передать ему объект `File`. Конструктор класса `RandomAccessFile` принимает строковый параметр, указывающий способ доступа к файлу (чтение «`r`», запись «`w`», чтение и запись «`rw`»):

```
// пример записи текста в произвольную позицию в файле
try {
 File f = new File("file.txt");
 RandomAccessFile raf = new RandomAccessFile(f, "rw");

 // чтение первого символа файла
 char ch = raf.readChar();

 // переход к концу файла
 raf.seek(f.length());

 // запись в конец файла
 raf.writeChars("The End");

 // закрытие файла
 raf.close();
} catch (IOException e) {
 System.out.println("Ошибка при чтении или записи файла.");
}
```

Фильтрация потока ввода

Иногда требуется извлекать из потока ввода только некоторые строки, а именно те, которые содержат заданный образец; все остальные строки нужно отклонять. Так действует известная в мире Unix утилита-фильтр **grep**. Напишем собственный класс `GrepInputFilter` на базе `FilterInputStream`, решающий задачу фильтрации потока ввода (см. листинг 7.3).

Листинг 7.3. Пример фильтрации потока ввода

```
class GrepInputStream extends FilterInputStream {

 String substring = null; // образец для поиска
 BufferedReader br;

 public GrepInputStream(InputStream in, String substring)
{
    super(in);
```

```
    this.br = new BufferedReader(new
InputStreamReader(in));
    this.substring = substring;
  } // конструктор GrepInputStream

  public final String readLine() throws IOException {

    String line;

    do {
      line = br.readLine();
    } while ((line != null) && line.indexOf(substring) == -1);
    return line;
  } // readLine()

} // GrepInputStream class
```

Поток ввода обрабатывается при помощи класса `BufferedReader`, предназначенного для чтения символьных данных. В JDK 1.1 эта функция была реализована методом `readLine()` класса `DataInputStream`. Начиная с версии JDK 1.2 метод `DataInputStream.readLine()` был объявлен устаревшим и запрещен для использования (deprecated) с целью устранения возможных конфликтов. Вместо него используется метод `BufferedReader.readLine()`.

Метод `readLine()` класса `GrepInputStream` последовательно перебирает строки из потока ввода, пока не достигнет конца файла или строки, содержащей заданный образец `substring`. Найденную строку (или `null` в случае конца файла) метод возвращает.

Листинг 7.4. Демонстрация класса GrepInputStream

```
public class Grep {

  public static void main(String[] args) {

    if ((args.length == 0) || (args.length > 2)) {
      System.out.println("Usage: java Grep <образец> <имя_
файла>");
      System.exit(0);
    }

    try {
      FileInputStream fis = new FileInputStream(args[1]);
      GrepInputStream gis =
        new GrepInputStream((InputStream) fis, args[0]);
```

```
        String line;
        for (;;) {
          line = gis.readLine();
          if (line == null) break;
          System.out.println(line);
        } // for (;;)
        gis.close();

      } catch (IOException ioe) {
        System.err.println(ioe.getMessage());
      }
    } // main(String[])
  } // Grep class
```

Рекурсивная упаковка папки в ZIP-архив

В JDK давно имеется пакет `java.util.zip`, классы которого предназначены для работы с ZIP-архивами — их создания, чтения, добавления и удаления элементов и т. п. Напишем утилиту, которая будет архивировать содержимое папки с ее подпапками.

В пакете `java.util.zip` есть класс `ZipOutputStream`, «на лету» архивирующий поток вывода. При создании объект этого класса должен получить как параметр экземпляр стандартного выходного потока в файл — `FileOutputStream`:

```
archiveName = "c:\archiv.zip";
ZipOutputStream zos = new ZipOutputStream(new
                          FileOutputStream(archiveName));
```

Чтобы поместить файл в архив, нужно создать экземпляр класса `ZipEntry`, представляющий элемент архива, передав его конструктору имя архивируемого файла:

```
fileName = "c:\windows\explorer.exe";
ZipEntry ze = new ZipEntry(fileName);
zos.putNextEntry(ze);
```

Теперь поток ввода из файла `fileName` связан с потоком вывода в архив. Осталось последовательно читать содержимое файла в байтовый массив определенного размера (буфер) и записывать содержимое буфера в поток вывода:

```
byte[] readBuffer = new byte[2048];
int bytesReaded = 0;
FileInputStream fis = new FileInputStream(fileName);

while ((bytesRead = fis.read(readBuffer)) == -1) {
```

```
 zos.write(readBuffer, 0, bytesRead);
}
fis.close();
```

Мы пишем утилиту, которая архивирует папку со всеми ее файлами и подпапками, поэтому нам не обойтись без метода рекурсивного просмотра папки:

```
private static void walkingDir(String walkedDir) {

 // получение списка элементов папки
 File walkDir = new File(walkedDir);
 String[] dirList = zipDir.list();

 // перебор списка элементов
 for (int i = 0; i < dirList.length; i++) {

   File f = new File(dirList[i]);
   if (f.isDirectory()) {
     // найдена папка: просматриваем ее
     walkingDir(f.getPath());
     continue;
   } else {
     // здесь будет код, обрабатывающий файл
     System.out.println(f.getPath());
   }

 } // for

} // walkingDir(String)
```

Теперь объединим действия по архивированию файлов и рекурсивному просмотру папок. Утилита будет называться ZipDir.

Листинг 7.5. Пример создания архива ZIP

```
import java.util.zip.*;
import java.io.*;

class ZipDir {

 public static void exec(String zipFile, String zippedDir)
     throws FileNotFoundException, IOException {

   ZipOutputStream zos = new ZipOutputStream(
     new FileOutputStream(zipFile));
```

```
      walkingDir(zos, zippedDir);
      zos.close();
    } // exec(String, String)

    public static void walkingDir (ZipOutputStream zos,
   String zippedDir) {

      File zipDir = new File(zippedDir);
      String[] dirList = zipDir.list();

      for (int i = 0; i < dirList.length; i++) {

        File f = new File(dirList[i]);
        if (f.isDirectory()) {
          String filePath = f.getPath();
          walkingDir (zos, filePath);
          continue;
        } else {

          try {
            byte[] readBuffer = new byte[2048];
            int bytesReaded = 0;

            String fullPath = zippedDir + "\\" + f.getPath();
            System.out.println("\t архивируется " + fullPath);
            FileInputStream fis =
              new FileInputStream(fullPath);
            ZipEntry ze = new ZipEntry(fullPath);
            zos.putNextEntry(ze);

            while ((bytesReaded = fis.read(readBuffer)) != -1){
              zos.write(readBuffer, 0, bytesReaded);
            }
            fis.close();

          } catch (FileNotFoundException fnfe) {
            System.out.println(fnfe.getMessage());
          } catch (IOException ioe) {
            System.out.println(ioe.getMessage());
          }
        } // ! isDirectory
      }    // for
    }      // walkingDir (ZipOutputStream, String)
  }        // ZipDir class
```

```
public class ZipDirTest {

 public static void main(String[] args) {

    // утилита ZipDirTest принимает два аргумента:
    // имя_архива и имя_архивируемой_папки
    if (args.length != 2) {
      System.out.println("Usage: java ZipDirTest
                         <имя_архива> <имя_папки>");
      return;
    }

    String zipFile = args[0];
    String zippedDir = args[1];

    try {
      System.out.println("Начало архивации папки: "
                         + zippedDir);
      ZipDir.exec(zipFile, zippedDir);
      System.out.println("Архив был записан успешно: "
                         + zipFile);
    } catch (FileNotFoundException fnfe) {
      System.out.println(fnfe.getMessage());
    } catch (IOException ioe) {
      System.out.println(ioe.getMessage());
    }
 } // main(String[])
} // ZipDirTest class
```

Основным элементом класса `ZipDir` является статический метод `exec()`. Он принимает два параметра: первый — `zipFile` — содержит название нового ZIP-архива, а второй — `zippedDir` — папку, содержимое которой должно быть помещено в архив.

Метод `exec()` открывает поток вывода в архив и передает его экземпляр методу `walkingDir()`, выполняющему основную работу. Метод `walkingDir()` рекурсивно перебирает дерево папок, начиная с папки, имя которой получает в параметре `zippedDir`, и копирует в архив все обычные файлы.

7.5. Файлы XML

Расширяемый язык разметки (eXtensible Markup Language, XML) — это язык описания документов, напоминающий язык разметки гипертекста

HTML, но гораздо более универсальный. Как и HTML, XML представляет собой систему тэгов, описывающих компоненты документа. Если HTML — это набор стандартных тэгов, которые должен понимать любой веб-браузер, то XML дает пользователям возможность определять свои собственные тэги. В результате в формате XML можно представить документ любой структуры, содержащий информацию практически любой природы, но разработка программного обеспечения для чтения таких документов представляет собой самостоятельную задачу.

Вывод содержимого XML-документа по модели DOM

Как уже сказано, спецификация DOM (Document Object Model) представляет файл, размеченный тэгами XML, в виде документа, состоящего из дерева узлов. Каждый из узлов может содержать кроме дочерних узлов также именованные атрибуты. Классы, работающие с документами DOM, находятся в пакете `org.w3c.dom`.

Документ DOM — это внутреннее представление файла XML, генерируемое грамматическим анализатором (разборщиком, или парсером) `DocumentBuilder` из объекта `File`, соответствующего файлу на диске. Экземпляр класса `DocumentBuilder` создает «фабрика классов» — класс `DocumentBuilderFactory`. После этого для получения объекта `Document`, представляющего документ DOM, достаточно вызвать метод `parse()` объекта `DocumentBuilder`, передав ему указатель на файл:

```
File docFile = new File("y:\test.xml");
try {
 DocumentBuilderFactory factory =
                    DocumentBuilderFactory.newInstance();
 factory.setIgnoringComments(true);

 // создание экземпляра парсера
 DocumentBuilder builder = factory.newDocumentBuilder();

 // чтение файла
 Document doc = builder.parse(docFile);

} catch (Exception e) {
 System.out.println(ex);
}
```

Необходимо отметить, что у метода `parse()` есть несколько перегружаемых вариантов. Кроме использованного в примере варианта с параметром `File`, можно передать любой поток ввода, созданный на базе

класса `InputStream`, либо URL в форме строки (`String`), либо класс `InputSource` интерфейса SAX (см. далее).

Документ DOM в памяти представляет собой древовидную структуру узлов и атрибутов, с которой вы можете «играть» с момента получения доступа к ее корневому узлу:

```
Document doc = builder.parse(docFile);
Node root = doc.getDocumentElement(); // доступ к
корневому узлу
```

Напишем утилиту, выводящую структуру и содержимое файла XML на экран (см. листинг 7.6).

Листинг 7.6. Пример чтения XML-документа с помощью DOM

```
import java.io.*;
import javax.xml.parsers.*;
import java.io.*;
import org.w3c.dom.*;

class DomPrint {

 // вывод отступа для разделения уровней
 private static void printIndentation(int level) {
    // отступ равен 3 пробелам * номер уровня
    for (int i = 0; i < level; i ++) {
      System.out.print("   ");
    }
 } // printIndentation(level)

 // печать содержимого узла
 private static void print(Node node, int level) {
    // для каждого последующего уровня выводится
    // пустая строка и отступ
    if (level > 0) {
      System.out.println("");
      printIndentation(level);
    }
    // вывод имени узла
    System.out.println(node.getNodeName().toString() + ":");

    // вывод атрибутов узла, если они есть
    if (node.hasAttributes()) {

      NamedNodeMap attributes = node.getAttributes();
```

```
      if (attributes.getLength() > 0) {
        // атрибуты - это следующий уровень
        level ++;

        for (int i = 0; i < attributes.getLength(); i++) {
          Node attribute = attributes.item(i);
          // вывод после отступа имени и
          // значения атрибута
          printIndentation(level);
          String s = "." + attribute.getNodeName() + "=";
          System.out.println(s + attribute.getNodeValue());
        } // for
        // уровень закончился
        level --;
      } // if (attributes.getLength() > 0)
    } // if (node.hasAttributes())

    // вывод значения узла, если оно непустое
    String value = node.getNodeValue();
    value = (value == null ? "" : value.trim());
    if (value.length() > 0) {
      printIndentation(level);
      System.out.println(value);
    }

    // печать дочерних узлов, если они есть
    if (node.hasChildNodes()) {
      // дочерние узлы - это следующий уровень
      level ++;
      NodeList children = node.getChildNodes();

      for (int i = 0; i < children.getLength(); i ++) {
        Node child = children.item(i);
        print(child, level);
      } // for
    } // if (node.hasChildNodes())
  } // print(Node, level) method

  public static void main(String[] args) {

    // утилита принимает один параметр: имя XML-файла
    if (args.length < 1) {
      System.out.println("Usage: java DomPrint имя_файла_xml");
      System.exit(0);
    }
```

```
        File docFile = new File(args[0]);
        try {
          DocumentBuilderFactory factory =
            DocumentBuilderFactory.newInstance();
          factory.setIgnoringComments(true);

          DocumentBuilder builder = factory.newDocumentBuilder();

          // разбор входного файла
          Document doc = builder.parse(docFile);
          // печать разобранного документа.
          print(doc.getDocumentElement(), 0);
        } catch(Exception ex) {
          System.out.println(ex);
        }
      } // main(String[]) method
    } // DomPrint class
```

Вывод содержимого XML-документа по модели SAX

Если грамматический анализатор, работающий по модели DOM, читает из XML-файла древовидную структуру, отражающую вложенность элементов документа, то работа парсера модели SAX (Simple API for XML) основана на событиях. Событием при чтении XML-файла считается появление тэга, открывающего или закрывающего узел, или текста в теле узла. Результатом разбора документа XML является при этом не дерево в памяти, а отдельный его узел. Модель SAX удобнее для последовательной обработки XML-файла, потому что требует меньшего объема оперативной памяти. Набор классов и интерфейсов SAX находится в пакетах `javax.xml.parsers`, `org.xml.sax` и подпакетах последнего: `org.xml.sax.helpers` и `org.xml.sax.ext`.

Чтобы анализировать XML-файл по модели SAX, нужно передать парсеру экземпляр класса — наследника `DefaultHandler`. В классе `DefaultHandler` объявлены методы-обработчики событий: `startDocument()` и `endDocument()`, вызываемые в начале и конце чтения документа; `startElement()` и `endElement()`, вызываемые в начале и конце обработки элемента, то есть при появлении в файле открывающего и закрывающего тэгов; `characters()`, вызываемый для чтения текстового содержимого элемента. От вас требуется написать собственную реализацию этих методов.

Парсер SAX, как и парсер DOM, получается с фабрики классов, после чего вы вызываете его метод `parse()`, которому передаете объект `File`, соответствующий XML-файлу на диске, и объект класса-наследника `DefaultHandler`:

```
class SAXPrint extends DefaultHandler {

 // здесь должна быть реализация методов класса DefaultHandler,
 // см.листинг 7.7

 public static void main(String[] args) {
   try {
     File docFile = new File("y:\test.xml");

     // Получение экземпляра фабрики классов
     SAXParserFactory factory =
                  SAXParserFactory.newInstance();

     // получение парсера из фабрики классов
     SAXParser parser = factory.newSAXParser();

     // грамматический анализ входного файла;
     // экземпляр класса SAXPrint нельзя передать как this,
     // потому что в статических методах ключевое слово this
     // запрещено (гл.3)
     parser.parse(docFile, new SaxPrint());
   } catch(Exception ex) {
     System.out.println(ex);
   }
 } // main(String[]) method
} // SAXPrint class
```

Продемонстрируем вывод на экран содержимого XML-файла по модели SAX. Встретив тэг XML, парсер SAX сам вызывает методы класса `DefaultHandler` с соответствующими аргументами, представляющими элементы структуры документа. Нам осталось написать только реализацию этих методов так, чтобы они выводили элементы структуры на экран (см. листинг 7.7).

Листинг 7.7. Печать содержимого файла XML по модели SAX

```
import java.io.*;
import javax.xml.parsers.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;

class SaxPrint extends DefaultHandler {

 // реализация методов-обработчиков событий, объявленных
 // в классе DefaultHandler
```

```
public void startDocument() throws SAXException {
   System.out.println("Начало документа");
} // startDocument() method

public void endDocument() throws SAXException {
   System.out.println("");
   System.out.println("Конец документа");
} // endDocument() method

public void startElement(String uri, String lName, String qName,
         Attributes attribs)
        throws SAXException {
   // вывод имени узла
   System.out.println("");
   System.out.println("   Начало " + qName);

   // вывод атрибутов, если они есть
   if (attribs.getLength() > 0) {
      for (int i = 0; i < attribs.getLength(); i ++) {
         System.out.println("      ." + attribs.getQName(i) +
            "=" + attribs.getValue(i));
      }
   }
} // startElement(uri, lName, qName, Attributtes)

public void endElement(String uri, String lName, String qName)
      throws SAXException {
   System.out.println(" Конец " + qName);
} // endElement(uri, lName, qName) method

public void characters(char[] data, int start, int length)
      throws SAXException {

   StringBuffer buf = new StringBuffer(length);
   buf.append(data, start, length);
   if (buf.toString().trim().length() > 0) {
      System.out.println(" " + buf);
   }
} // characters(char[], start, length)

public static void main(String[] args) {

   // утилита принимает один параметр: имя XML-файла
   if (args.length < 1) {
      System.out.println("Usage: java SaxPrint имя_файла_xml");
```

```
        System.exit(0);
      }

      try {
        File docFile = new File("y:\test.xml");

        SAXParserFactory factory =
                          SAXParserFactory.newInstance();
        SAXParser parser = factory.newSAXParser();

        parser.parse(docFile, new SaxPrint());

      } catch(Exception ex) {
        System.out.println(ex);
      }

  } // main(String[]) method
} // SAXPrint class
```

Трансформация документа XML

Как известно, для документов в формате HTML общие правила оформления (цвет и стиль шрифта, отступы и интервалы и т. п.) задаются обычно независимо от содержательной части документа в отдельном файле — так называемой таблице стилей.

Такой же подход оказался удобен и для представления документов в формате XML, поскольку тэги XML предназначены для структурирования документа, но не для задания правил его отображения. Для документов XML разработан собственный язык таблиц стилей XSL (XML Stylesheet Language). Применив таблицу стилей — файл на языке XSL — к XML-структурированному документу, можно преобразовать его в страницу HTML, документ PDF или документ другого формата. Такое преобразование называется XSL-трансформацией, и для него используется язык преобразований XSLT (XML Stylesheet Language for Transformation).

Продемонстрируем, как можно вывести все тот же файл `test.xml` на консоль средствами XSLT. Простейшая таблица стилей для вывода плоского текста в кодировке CP866 выглядит так:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.
w3.org/1999/XSL/Transform">
<xsl:output method="text" encoding="CP866">
</xsl:output>
</xsl:stylesheet>
```

Сохраните этот файл под именем `test.xsl`.

Классы и интерфейсы, реализующие трансформацию XSLT, собраны в пакете `javax.xml.transform` и его подпакетах. Объект-трансформатор класса `Transformer` получается с фабрики классов `TransformerFactory`. Его конструктор получает входной поток типа `SourceStream`, полученный из файла XSL. И, наконец, вызывается метод объекта трансформатора `transform()`, которому передается входной поток `SourceStream` из файла XML и поток вывода на консоль `StreamResult`.

Листинг 7.8. Пример XSLT трансформации

```
class Transform {
 public static void main(String[] args) {

    // утилита принимает два параметра:
    // имя XML-файла и имя файла с таблицей стилей
    if (args.length < 1) {
      System.out.println("Usage: java
                Transform имя_файла_xsl имя_файла_xml");
      System.exit(0);
    }

    File xslFile = new File(args[0]); // XSL-таблица
    File xmlFile = new File(args[1]);
                              // исходный XML-документ
    // создание входных и выходного потока
    StreamSource xslSource = new StreamSource(xslFile);
    StreamSource xmlSource = new StreamSource(xmlFile);
    StreamResult outResult = new StreamResult(System.out);

    try {
      // получение фабрики и трансформатора
      TransformerFactory factory =
                    TransformerFactory.newInstance();
      Transformer transformer =
                    factory.newTransformer(xslSource);

      // трансформация XML-документа.
      transformer.transform(xmlSource, outResult);
    } catch(Exception ex) {
      System.out.println(ex);
    }
 } // main(String[]) method
} // Transform class
```

Список использованной литературы

1. www.wikipedia.org

2. Васильев А. Н. Самоучитель Java с примерами и программами. 2-е издание. — СПб.: Наука и Техника, 2013. — 368 с.: ил.

3. www.oracle.com

Группа подготовки издания:

Зав. редакцией компьютерной литературы: *М. В. Финков*

Редактор: *Е. В. Финков*

Корректор: *А. В. Громова*

ООО «Наука и Техника»

Лицензия №000350 от 23 декабря 1999 года.

198097, г. Санкт-Петербург, ул. Маршала Говорова, д. 29.

Подписано в печать 09.12.2015. Формат 70x100 1/16.

Бумага офсетная. Печать офсетная. Объем 15 п. л.

Тираж 1800 экз. Заказ 9402.

Отпечатано с готовых файлов заказчика
в АО «Первая Образцовая типография»,
филиал «УЛЬЯНОВСКИЙ ДОМ ПЕЧАТИ»
432980, г. Ульяновск, ул. Гончарова, 14

ИЗУЧАЕМ

Java

на примерах и задачах

Эта книга является превосходным базовым учебным пособием для изучения языка программирования Java с нуля. По своей сути Java — популярная современная платформа, позволяющая писать программы, работающие почти на всех мыслимых и немыслимых операционных системах и практически любом оборудовании.

В книге содержатся рецепты и практические указания по решению задач, часто встречающихся при программировании на языке Java. Большинство авторов книг в своих трудах рассматривают теоретические основы языка и уделяют основное внимание базовому синтаксису языка, не рассматривая при этом практи-ческую сторону его применения. Эта же книга старается восполнить недостаток практического материала, содержит множество примеров с комментариями, которые вы сможете использовать в качестве основы своих программных решений, изучения Java.

Материал книги излагается последовательно и сопровождается большим количеством наглядных примеров, разноплановых практических задач и детальным разбором их решений.

НиТ
ИЗДАТЕЛЬСТВО

www.nit.com.ru

ISBN 978-5-94387-993-7

9 785943 879937

Россия: Санкт-Петербург,
пр. Обуховской обороны, 107
для писем: 192029, Санкт-Петербург, а/я 44
(812) 412 7025, 412 7026
e-mail: nit@mail.wplus.net

Украина: 02166, Киев, ул. Курчатова, 9/21
(044) 516 3866
e-mail: nits@voliacable.com