







УДК 004.43

ББК 32.973.26-018.1 М15



Mike McGrath

JAVA IN EASY STEPS, 5TH EDITION

By Mike McGrath. Copyright ©2015 by In Easy Steps Limited.

Translated and reprinted under a licence agreement from the Publisher:

In Easy Steps, 16 Hamolton Terrace, Holly Walk, Leamington Spa, Warwickshire, U.K. CV32 4LY.

М15

МакГрат, Майк.

Программирование на Java для начинающих / Майк Мак- Грат ; [пер. с англ. М.А. Райтмана]. – Москва : Издательство

«Э», 2016. – 192 с. – (Программирование для начинающих).

ISBN 978-5-699-85743-2

Книга «Программирование на Java для начинающих» является исчерпыва- ющим руководством для того, чтобы научиться программировать на языке Java. В этой книге с помощью примеров программ и иллюстраций, показываю- щих результаты работы кода, разбираются все ключевые аспекты языка. Уста- новив свободно распространяемый Java Development Kit, вы с первого же дня

сможете создавать свои собственные исполняемые программы!

УДК 004.43

ББК 32.973.26-018.1

ISBN 978-5-699-85743-2

© Райтман М.А., перевод на русский язык, 2016

© Оформление. ООО «Издательство «Э», 2016

**Оглавление**

**Предисловие** 8

**1**

**Введение 9**

Установка JDK 12

Создание первой программы на Java 14

Компиляция и запуск программ 16

Создание переменных. 18

Распознавание типов данных 20

Создание констант 22

Добавление комментариев 23

Проблемы отладки 24

Заключение 26

**2**

**Выполнение операций 27**

Выполнение арифметических операций 28

Присваивание значений 30

Сравнение величин 32

Оценочная логика 34

Проверка условий 36

Приритет операций 38

Управляющие литералы 40

Работа с битами 42

Заключение 44

**3**

**Создание операторов 45**

Ветвление с помощью условного оператора **if** 46

Альтернативное ветвление 48

Ветвление с помощью переключателей 50

Цикл **for** 52

Цикл **while** 54

Циклы **do-while** 56

Выход из циклов 58

Возврат управления 60

Заключение 62

**4**

**Работа с данными 63**

[Преобразование типов 64](#_TOC_250032)

[Создание массивов переменных 66](#_TOC_250031)

[Передача аргументов 68](#_TOC_250030)

[Передача множественных аргументов 70](#_TOC_250029)

[Обход элементов в цикле 72](#_TOC_250028)

[Изменение значений элемента 74](#_TOC_250027)

[Добавление размеров массива 76](#_TOC_250026)

[Перехват исключений 78](#_TOC_250025)

[Заключение 80](#_TOC_250024)

**5**

**Работа с данными 81**

[Изучение классов Java 82](#_TOC_250023)

[Математические вычисления 84](#_TOC_250022)

[Округление чисел 86](#_TOC_250021)

[Генерация случайных чисел 88](#_TOC_250020)

[Управление строками 90](#_TOC_250019)

[Сравнение строк 92](#_TOC_250018)

[Поиск строк 94](#_TOC_250017)

[Обработка символов 96](#_TOC_250016)

[Заключение 98](#_TOC_250015)

**6**

**Создание классов 99**

[Программа как набор методов 100](#_TOC_250014)

[Область видимости 102](#_TOC_250013)

[Использование множественных классов 104](#_TOC_250012)

[Расширение существующего класса 106](#_TOC_250011)

[Создание объектного класса 108](#_TOC_250010)

[Создание экземпляра объекта 110](#_TOC_250009)

[Инкапсуляция свойств 112](#_TOC_250008)

[Создание объектных данных 114](#_TOC_250007)

[Заключение 116](#_TOC_250006)

**7**

**Импортирование функций 117**

[Работа с файлами 118](#_TOC_250005)

[Чтение консольного ввода 120](#_TOC_250004)

[Чтение файлов 122](#_TOC_250003)

[Запись файлов 124](#_TOC_250002)

[Сортировка элементов массива 126](#_TOC_250001)

[Создание списочных массивов 128](#_TOC_250000)

Работа с датой 130

Форматирование чисел 132

Заключение 134

Создание окна 136

**8**

**Построение интерфейсов**

**135**

Добавление кнопок 138

Добавление меток 140

Добавление текстовых полей 142

Добавление элементов выбора 144

Добавление переключателей 146

Изменение внешнего вида интерфейса 148

Размещение компонентов 150

Заключение 152

**9**

**Распознавание событий 153**

«Прослушивание» событий 154

Генерация событий 155

Обработка событий кнопок 156

Обработка событий элементов 158

Реагирование на события клавиатуры 160

Ответ на события мыши 162

Вывод сообщений 164

Запрос пользовательского ввода 166

Воспроизведение звука 168

Заключение 170

**10**

**Развертывание программ**

**171**

Методы развертывания 172

Распространение программ 174

Построение архивов 176

Развертывание приложений 178

Подписывание jar-файлов 179

Использование технологии Web Start 180

Создание апплетов 182

Встраивание апплетов в код веб-страницы 184

Развертывание апплетов 186

Заключение 188

**Предметный указатель 189**

**Предисловие**

В этой книге мои предыдущие публикации по Java-программированию дополнены новинками данной технологии. Все приведенные здесь примеры демонстрируют возможности Java, которые поддерживают- ся современными компиляторами в операционных системах Windows и Linux, а представленные скрин- шоты отражают реальные результаты компиляции и исполнения приведенного кода.

**Некоторые соглашения**

Листинги кода, приведенные в книге, выглядят вот так:

### // Примечание к коду

### String message = "Добро пожаловать в язык Java!" ; System.out.println( message ) ;

Кроме того, для идентификации каждого исходного файла, описанного в пошаговых инструкциях, на

полях рядом с каждым пунктом будет появляться значок и имя соответствующего файла:



**JAVA**



**110100101010**

**000101101011**

**110100101010**

**000101101011**

**110100101011**

**CLASS**



**JAR**



**JNLP**

*App.java App.class App.jar App.jnlp*

**Получение исходных кодов**

Для удобства я поместил файлы исходных кодов всех примеров, представленных в этой книге, в один ZIP-архив. Вы можете получить весь архив, выполнив следующие простые шаги.

1. Откройте браузер и загрузите архив по ссылке [www.eksmo.ru.](http://www.eksmo.ru/)
2. Извлеките содержимое архива в любое удобное место на вашем компьютере.

Я искренне надеюсь, что вам понравится открывать для себя интересные возможности языка про- граммирования Java и вы получите при этом не меньшее удовольствие, чем я во время работы над этой книгой.

**1 Введение**

***Добро пожаловать в удивительный мир программирования на Java. В этой главе***

***демонстрируется, как создавать и запускать на выполнение простые Java-программы и как хранить в них данные.***

* **Установка JDK**
* **Создание первой программы на Java**
* **Компиляция и запуск программ**
* **Создание переменных**
* **Распознавание типов данных**
* **Создание констант**
* **Добавление комментариев**
* **Проблемы отладки**
* **Заключение**

Язык программирования Java разработан в 1990 году Джеймсом Гос- лингом (James Gosling), инженером из компании Sun Microsystems. Он решил создать новый язык, так как был не очень доволен используемым языком программирования C++, и назвал его сначала Oak\* — в честь дуба, который он мог наблюдать из окна своего офиса.

**1. Введение**

По мере роста популярности Всемирной паутины компания Sun пред- положила, что язык, разработанный Гослингом, может быть использо- ван для интернет-разработок. Впоследствии языку дали название Java (просто потому что это звучит лучше), и в 1995 году он стал свобод- но доступным. Разработчики по всему миру быстро адаптировались к прекрасному новому языку и, благодаря его модульному дизайну, получили возможность создавать новую функциональность, обогащая тем самым языковое ядро. В последующих версиях языка было добав- лено множество впечатляющих функций, превративших Java в очень мощный инструмент на сегодня.



Основную сущность языка Java составляют библиотеки файлов, назы- ваемые *классами*, каждый из которых содержит небольшие фрагмен- ты проверенного, готового к выполнению кода. Подобно кирпичам в стене, любые из этих классов можно встраивать в новую программу, и таким образом, для окончательного завершения программы обычно остается написать небольшую часть кода. Такая методика экономит программистам много времени и является одной из основных причин широкой популярности программирования на Java. К тому же такая модульная организация упрощает процесс отладки: ведь найти ошиб- ку в небольшом модуле гораздо проще, чем в одной большой про- грамме.

**10**

Технология Java является одновременно и платформой, и языком про- граммирования. Исходные коды программ языка Java написаны в че- ловекочитаемом виде в обычном текстовом файле с расширением *.java*, который затем компилируется в файлы с расширением *.class* при помо- щи компилятора javac. После этого программа исполняется интерпре- татором java при помощи виртуальной машины Java (Java VM):

**Новинка**

Значок, показанный выше, указывает на добавленные или улучшенные функции, представленные в послед- ней версии Java.

*Program.java Program.class Программа*



**JAVA**



Компилятор



**110100101010**

**000101101011**

**110100101010**

**000101101011**

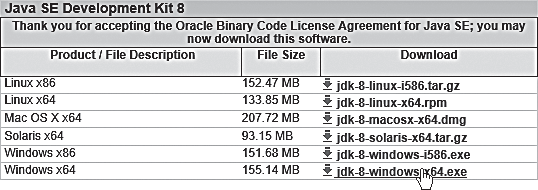
**110100101011**

**CLASS**



**Java VM**

\* В пер. с англ. «Дуб». *Прим. пер.*

Поскольку виртуальная машина Java доступна на различных платфор- мах, одни и те же файлы *.class* можно запускать как в среде Windows и Linux, так и на компьютере Mac. Это и есть основной принцип кросс- платформенности языка — «написано единожды, работает везде».

**11**

Чтобы создавать программы на Java, на вашем компьютере должны быть установлены библиотеки классов и компилятор javac, а чтобы их запускать, нужно установить среду Java Runtime Environment (JRE), ко- торая поддерживает интерпретатор java. Все вышеперечисленные ком- поненты содержатся в пакете Java Platform, Standard Edition Development Kit (JDK), находящемся в свободном доступе.

Программы, представленные в этой книге, используют версию JDK 8, включающую как инструмент разработчика, так и среду исполнения. Загрузить данный пакет вы можете с сайта Oracle по адресу **www.oracle. com/technetwork/java/javase/downloads**.





Пакет JDK 8 доступен в версиях для 32-битной и 64-битной операци- онных систем Linux, OS X, Solaris и Windows — примите лицензионное соглашение Oracle, затем выберите подходящую для вас версию и за- грузите Java Development Kit.



**На заметку**

Страница загрузки сайта Oracle содержит также другие пакеты, но для того чтобы начать программи- рование на Java, требуется только JDK 8.

**Совет**

Слухи о том, что JAVA означает Just Another Vague Acronym (еще одна непонят- ная аббревиатура) не под- тверждены.

**1. Введение**

Возможно, что предыду- щие версии JRE уже уста- новлены у вас в системе и веб-браузер может вы- полнять Java-апплеты, но

во избежание конфликтов

с новой версией JDK8 лучше всего переустановить JRE.

**12**

Для упрощения вы можете установить только мини- мальный набор функций.

**Установка JDK**

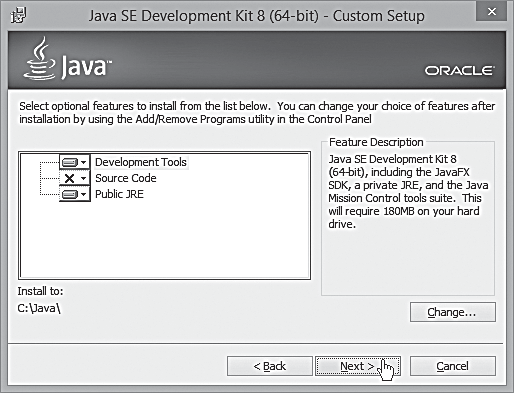
Для того чтобы установить Java на ваш компьютер, выберите подхо- дящий пакет Java Development Kit для вашей операционной системы со страницы загрузки Oracle и выполните следующие шаги.

1. Удалите все предыдущие версии JDK и/или Java Runtime Environ- ment, установленные на вашей системе.
2. Запустите процесс установки и примите лицензионное согла- шение.



**Внимание**

1. При появлении диалогового окна настроек либо подтвердите предлагаемое расположение установки, либо укажите предпочи- таемое расположение, например *C:\Java* в операционных системах Windows или */usr/Java* — в Linux.



1. Убедитесь, что в списке выбраны пункты **Development Tools** и **Public JRE**. Вы можете также отменить пометку для других опций установки, поскольку они необязательны для работы с примерами этой книги.
2. Нажмите кнопку **Далее** (Next), чтобы установить все необходимые инструменты и библиотеки классов Java в выбранное место.



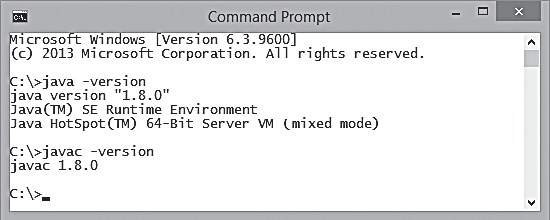
**Совет**

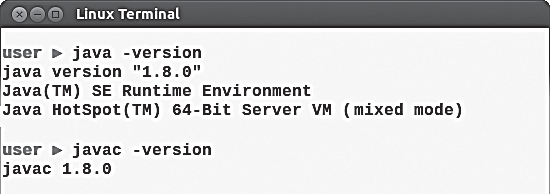
Инструменты для компиляции и запуска Java-программ обычно выпол- няются из командной строки и размещаются в подкаталоге *bin* каталога *Java*. Их можно сделать доступными из любого места вашего компью- тера, если добавить этот каталог в системный путь.

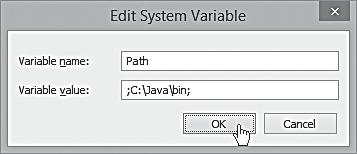
* + В системе Windows 8, 7, Windows Vista или Windows XP нажмите кнопку **Пуск** (Start) и откройте окно **Панель управления**  **Систе- ма**  **Дополнительные параметры системы**  **Переменные среды** (Control Panel  System  Advanced System Settings  Environment Variables). Выберите системную переменную с именем **Path** и на- жмите кнопку **Изменить** (Edit). Добавьте в конец значение пере- менной имя подкаталога **bin** (например, **C:\Java\bin;**) и нажмите кнопку **OK**.
  + В системе Linux добавьте путь к каталогу **bin** в файл *.bashrc* из ваше- го домашнего каталога. Например, добавьте строку **PATH=$PATH:/ usr/Java/bin** и сохраните файл.

Чтобы проверить установленное окружение, откройте командную строку, введите в ней команду **java -version** и нажмите клавишу **Enter**, чтобы увидеть номер версии интерпретатора. Теперь наберите команду **javac -version**, нажмите клавишу **Enter**, и вы увидите номер версии компилятора. Номера должны совпадать — в нашем случае это версия

1.8.0. Теперь вы готовы начать программировать на Java.







Имена каталога, содержащие пробелы, должны быть за- ключены в двойные кавыч- ки, например **"C:\Program Files\Java\bin";**.



**На заметку**

Если файл *.bashrc* не виден в списке файлов каталога, выберите команду меню **Вид**  **Показать скрытые**

**файлы** (View  Show Hidden Files).

**13**



**Внимание**

В ранних версиях Windows инструменты JDK можно сделать доступными с по- мощью редактирования файла **autoexec.bat**, добавив имя подкаталога *bin* в конец строки **SET PATH**.

**1. Введение**



**JAVA**

*Hello.java*

Язык Java чувствителен



**Внимание**

**14**

к регистру, поэтому **Hello**

и **hello** — совершенно разные имена. По традиции, имена программ принято всегда начинать с пропис- ной буквы.

Программы на Java всегда сохраняются в файле с точно таким же именем с добавле- нием расширения .*java*.



**На заметку**

**Создание первой программы на Java**

Все программы на Java обычно начинаются как текстовые файлы, ко- торые впоследствии используются для создания файлов «классов», ко- торые, в свою очередь, являются в действительности исполняемыми программами. Это означает, что программы на Java могут быть напи- саны в любом простейшем текстовом редакторе, таком как, например, Блокнот (Notepad).

Выполните данные шаги, чтобы создать простую Java-программу, кото- рая будет выводить традиционное приветствие.

1. Откройте простейший текстовый редактор, например, Блокнот (Notepad) и наберите в нем следующий код в точности как здесь — вы создадите класс с именем **Hello**.

### class Hello

### {

### }

1. Между двумя фигурными скобками класса **Hello** вставьте следую- щий код для создания метода **main** класса **Hello**.

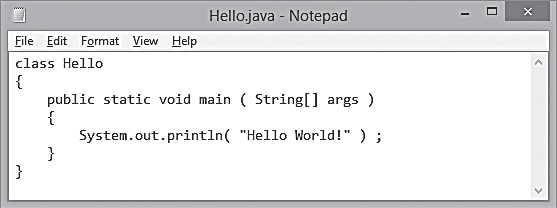
### public static void main ( String[] args )

### {

### }

1. Между фигурными скобками метода **main** добавьте следующую строку кода, который определяет то, что будет делать программа.

### System.out.println( "Hello World!" ) ;

1. Сохраните файл в любом удобном месте и назовите его точно так: *Hello.java* — готовая программа выглядит теперь следующим об- разом:

Для более четкого понимания каждой отдельной части программы рас- смотрим их индивидуально.

**Программный контейнер**

**15**

### class Hello { }

Имя программы объявляется после ключевого слова **class**, а после него следует пара фигурных скобок. Весь код программы, который станет определять класс **Hello**, будет помещен внутри этих фигурных скобок.

**Метод main**

### public static void main ( String[] args ) {}

Эта выглядящая устрашающе строка является стандартным кодом для определения начальной точки, фактически всех программ на Java. В та- ком виде она будет использована почти во всех примерах данной кни- ги, поэтому стоит ее запомнить.

В коде объявляется метод с именем **main**, который будет содержать вну- три фигурных скобок все инструкции программы.

Ключевые слова **public static void**, предваряющие имя метода, опре- деляют, как метод должен использоваться, и объясняются подробнее позже.

Строка кода **( String[] args )** используется при передаче значений методу и также будет подробнее объясняться позже.

**Оператор**

### System.out.println( "Hello World!" ) ;

Операторы представляют собой команды, которые должна выполнить программа и которые всегда должны заканчиваться точкой с запятой. Метод может содержать многочисленные операторы внутри своих фи- гурных скобок, формируя тем самым «блок операторов», определяю- щий набор задач для выполнения. В данном случае одиночный опера- тор дает программе команду вывести строку текста.

Далее вы узнаете, как скомпилировать и запустить эту программу.



**На заметку**

Все самостоятельные про- граммы Java должны содер- жать метод **main** (в отличие от java-апплетов, формат ко- торых объясняется позднее).



**Совет**

Создайте каталог *MyJava*, чтобы сохранять все ваши программные файлы в нем.

**1. Введение**

Если в командной строке (терминальном окне) про- сто набрать команду **javac** и нажать клавишу **Enter**, то можно вывести параметры Java-компилятора.



**Совет**

**16**

Вы можете также запускать компиляцию исходного кода из любого места, в том случае если вы укажете ком- пилятору **javac** полный путь до файла — в данном случае это **C:\MyJava\Hello.java**.



**Совет**

**Компиляция и запуск программ**

Перед тем как запускать Java-программу на выполнение, ее нужно скомпилировать в файл класса при помощи компилятора Java, который располагается в подкаталоге *bin* и имеет имя **javac**. Ранее было описа- но, каким образом нужно добавлять подкаталог *bin* в системный путь, чтобы компилятор **javac** мог быть запущен из любого места системы.

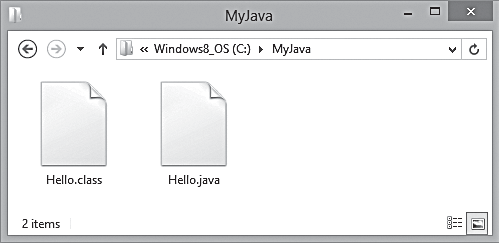
Выполните следующие шаги, чтобы скомпилировать программу с пре- дыдущей страницы.

1. Откройте командную строку (терминальное окно) и перейдите в каталог, в котором вы сохранили файл с исходным кодом *Hello. java*.
2. В строке-подсказке наберите **javac**, затем пробел и имя файла *Hello. java* с исходным кодом и нажмите клавишу **Enter**.



Если компилятор **javac** находит ошибки в тексте программы, он оста- навливается и отображает сообщение, указывающее природу ошиб- ки, — о проблемах отладки смотрите далее.

В случае, если компилятор **javac** не находит никаких ошибок, он созда- ет новый файл с именем программы и расширением *.class*.



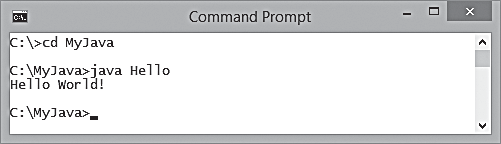
Когда процесс компиляции завершается, фокус возвращается в команд- ную строку без какого-либо предупреждающего сообщения и програм- ма готова к запуску.

Программный интерпретатор Java — это приложение с именем **java**, которое также размещается в подкаталоге *bin* вместе с компилятором **javac**. Поскольку данный каталог уже добавлен в системный путь, то интерпретатор **java** также может быть запущен из любого места.

Выполните следующие, шаги для того чтобы запустить программу, ко- торая была скомпилирована, используя процедуру, описанную на пре- дыдущей странице.

* 1. Откройте командную строку (терминальное окно) и перейдите в каталог, где расположен файл программы *Hello.class*.
  2. Наберите **java** и имя программы **Hello**, а затем нажмите клавишу

### Enter.



Программа **Hello** запускается и исполняет команды, описанные в опе- раторах основного метода, в данном случае выводит строку Hello, world! После завершения фокус снова возвращается к строке-подсказке.

Процесс компиляции и запуска программы на Java обычно объединя- ется в последовательные шаги, и эти шаги одинаковы и независимы от платформы, на которой они выполняются. Снимок экрана, представ- ленный ниже, демонстрирует последовательную компиляцию и запуск программы **Hello** в операционной системе Linux:

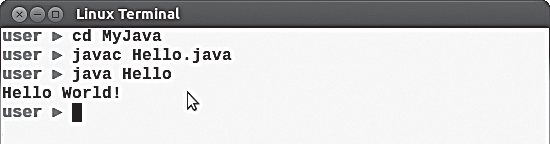
**17**

При запуске программ используйте только имя программы без расширения



**Внимание**

*.class*.



Так же, как и другие опера- торы, каждое объявление переменной должно закан- чиваться точкой с запятой.

**1. Введение**

**Создание переменных**

В Java-программировании «переменная» — это некоторый контейнер, в котором может храниться значение для дальнейшего использования в программе. Сохраненное значение может изменяться по мере испол- нения программы — отсюда и название «переменная».



**Внимание**

Переменная создается с помощью «объявления», в котором указыва- ется тип данных, содержащихся в переменной, и задается для нее имя. Например, чтобы объявить переменную строкового типа **String** с име- нем **message**, которая будет содержать обычный текст, мы пишем:

### String message ;

При назначении имен переменным программисты должны следовать некоторым договоренностям. Имя переменной в Java может начи- наться только с латинской буквы, знака **$**, либо знака **\_**. Последующие символы могут быть латинскими буквами, цифрами, знаками **$** и зна- ками **\_**. Все имена чувствительны к регистру, так что **var** и **Var** явля- ются совершенно разными переменными. Знаки пробелов в именах не допускаются.

Также запрещается использовать в качестве имен переменных пред- ставленные в таблице ниже ключевые слова Java, которые имеют осо- бое значение в языке:

**18**

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

Рекомендуется именование переменной словами либо распознаваемы- ми аббревиатурами, описывающими назначение переменных. Напри- мер, для описания кнопки номер 1 можно использовать имена **button1** или **btn1**. Для однословных имен предпочтительно выбирать буквы нижнего регистра, а для имен многословных обычно используется так называемый «горбатый регистр», когда первая буква второго слова в верхнем регистре заглавная, например, **gearRatio**.

После того как переменная объявлена, ей можно присвоить начальное значение соответствующего типа, используя знак равенства, причем это можно сделать либо во время объявления, либо позднее, в про- грамме. После этого к переменной можно обратиться по ее имени в лю- бое время.

Выполните следующие шаги для создания программы, в которой пере- менная объявляется, тут же инициализируется, а затем изменяется..

1. Создайте новую программу с именем **FirstVariable**, содержащую метод **main**.

### class FirstVariable

### {



**Совет**

Строго говоря, некоторые слова, представленные

в таблице, на самом деле не являются ключевыми словами языка: **true**, **false** и **null** — это литералы, **String** — это специальное имя класса, **const** и **goto** — являются зарезервирован- ными словами (в текущей версии не используются). Они добавлены в таблицу, поскольку при именовании

переменных нужно избегать давать имена, совпадающие с этими словами.

### public static void main ( String[] args ) { }



**JAVA**

### }

**19**

1. Внутри фигурных скобок главного метода добавьте следующий код для создания, инициализации и последующего вывода пере- менной:

### String message = "Начальное значение" ; System.out.println( message ) ;

1. Добавьте следующие строки, в которых значение переменной из-

меняется и снова выводится на экран:

### message = "Измененное значение" ; System.out.println( message ) ;

1. Сохраните программу под именем *FirstVariable.java*, затем скомпи-

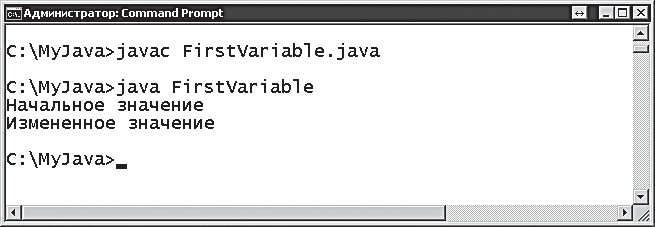
лируйте и запустите.

*FirstVariable.java*



**На заметку**

Если вы столкнулись с про- блемой при компиляции или запуске программы, можете обратиться к концу этой главы, где описаны вопросы отладки.



**Распознавание типов данных**

**1. Введение**

Типы данных, наиболее часто используемые при объявлении перемен- ных в Java, перечислены в таблице ниже с кратким описанием.



**Внимание**

Из-за погрешностей вычис- лений, вызванных использо- ванием типа данных с пла- вающей точкой (**float**), не следует использовать этот тип данных там, где нужна высокая точность (напри- мер, при финансовых рас- четах) — подробнее об этом см. в главе 7.

**20**

|  |  |  |
| --- | --- | --- |
| **Тип данных Описание Пример** | | |
| **char** | Одиночный символ в Юникоде | **'a'** |
| **String** | Любое количество символов в Юникоде | **"my String"** |
| **int** | Целое число в диапазоне от  –2147483648 до 2147483647 | **1000** |
| **float** | Вещественное число с плаваю- щей точкой | **3.14159265f** |
| **boolean** | Логическое значение true или false | **true** |



**На заметку**

Все ключевые слова, обо- значающие типы данных, начинаются с нижнего реги- стра, за исключением **String**, который является специаль- ным классом.

Обратите внимание, что переменные типа **char** выделяются одинарны- ми кавычками, а типа **String** — двойными. Также следует запомнить, что значение типа **float** всегда имеет суффикс **f**.

В дополнение к наиболее распространенным типам данных, представ- ленным выше, Java предоставляет возможность при некоторых обстоя- тельствах работать со специальными типами.

|  |  |
| --- | --- |
| **Тип данных Описание** | |
| **byte** | Целое число в диапазоне от –128 до 127 |
| **short** | Целое число в диапазоне от –32768 до 32767 |
| **long** | Целое число в диапазоне от –9223372036854775808 до 9223372036854775807 |
| **double** | Вещественное число в диапазоне от 1.7е–308 до 1.7е+308 |

Специальные типы данных применяются в более сложных Java-про- граммах, а примеры данной книги используют в основном общие типы, представленные в верхней таблице.

Чтобы написать программу на Java, в которой создаются, инициализи- руются и выводятся переменные всех пяти основных типов програм- мы, выполните следующие шаги.

1. Создайте новую программу с именем **DataTypes**, которая содержит стандартный метод **main**.

### class DataTypes

### {

### public static void main ( String[] args ) { }

### }

1. Внутри фигурных скобок главного метода добавьте объявление пяти переменных, следующее ниже.

### char letter = 'M' ;

### String title = "Java in easy steps" ; int number = 365 ;

### float decimal = 98.6f ; boolean result = true ;

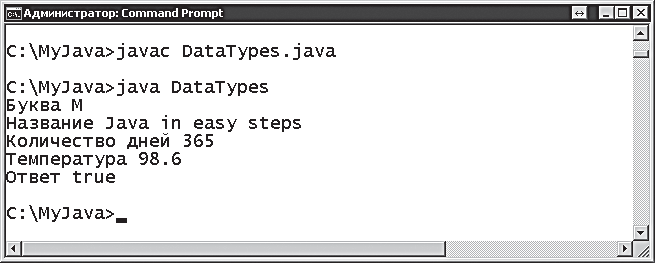
1. Добавьте следующие строки для вывода соответствующих пере-

менных с объединением строк.

### System.out.println( "Буква " + letter ) ; System.out.println( "Название " + title ) ; System.out.println( "Количество дней " + number ) ; System.out.println( "Температура " + decimal) ; System.out.println( "Ответ " + result ) ;

1. Сохраните программу под именем *DataTypes.java*, затем скомпили-

руйте и запустите.





**JAVA**

*DataTypes.java*



**Совет**

Обратите внимание, что сим- вол + здесь используется для объединения (конкатенации) текстовых строк и значений сохраненных переменных.



**Совет**

Компилятор Java будет со- общать об ошибке, если про- грамма попытается присво- ить переменной значение

не того типа — попробуйте изменить значение в приме- ре, а затем скомпилируйте, чтобы увидеть результат.

**21**

**Создание констант**

**1. Введение**

Часто в программах нужно использовать «константы» — фиксирован- ные значения, которые не должны изменяться по мере выполнения программы. Для объявления таких переменных существует ключевое слово **final**.

Константы в программе принято писать символами верхнего регистра, чтобы отличать их от обычных переменных. Если в программе проис- ходит попытка изменить константу, то компилятор **javac** выдает со- общение об ошибке.

Выполните следующие шаги для создания Java-программы, использую- щей константы:



**JAVA**

*Constants.java*

**22**



**Совет**

Символ **\*** используется здесь для перемножения значений констант, а скобки, в которых заключены три слагаемых, особого значения не имеют.

1. Создайте новую программу с именем **Constants**, содержащую стан- дартный метод **main**.

### class Constants

### {

### public static void main ( String[] args ) { }

### }

1. Внутри фигурных скобок главного метода добавьте следующий код для создания и инициализации трех целочисленных констант.

### final int TOUCHDOWN = 6 ; final int CONVERSION = 1 ; final int FIELDGOAL = 3 ;

1. Теперь объявите четыре целочисленные переменные.

### int td , pat , fg , total ;

1. Проинициализируйте обычные переменные, используя операции умножения и константы.

### td = 4 \* TOUCHDOWN ; pat = 3 \* CONVERSION ; fg = 2 \* FIELDGOAL ;

### total = ( td + pat + fg ) ;

1. Добавьте следующую строку для вывода общего количества очков.

### System.out.println( "Очков всего: " + total ) ;

1. Сохраните программу под именем *Constans.java*, затем скомпили- руйте и запустите, чтобы увидеть вывод общего количества очков: Очков всего: 33.

### ( 4 x 6 = 24, 3 x 1 = 3, 2 x 3 = 6, таким образом, 24 + 3 +

### 6 = 33 )

**Добавление комментариев**

При программировании на любом языке рекомендуется добавление комментариев в код программы для объяснения каждого определенно- го раздела. Комментарии делают код более легким для понимания как другими программистами, так и самим разработчиком, когда он воз- вращается к частям программы спустя некоторый период времени.

При программировании на Java комментарии могут добавляться двух ви- дов: многострочные с использованием символов **/\*** и **\*/** и однострочные с использованием символов **//**. Все, что находится между **/\*** и **\*/**, а также на строке после **//**, полностью игнорируется компилятором **javac**.

Если добавить комментарии в программу *Constants.java*, описанную в предыдущем разделе, исходный код может выглядеть примерно так:

### /\*



**JAVA**

### Программа, демонстрирующая использование констант.

### \*/

### class Constants

### {

### public static void main( String args[] )

### {

*Constants.java*

(с комментариями)

**23**

### // Константы для подсчета очков. final int TOUCHDOWN = 6 ;

### final int CONVERSION = 1 ; final int FIELDGOAL = 3 ;

### // Подсчет очков.

### int td , pat , fg , total ;

### td = 4 \* TOUCHDOWN ; // 4x6=24 pat = 3 \* CONVERSION ; // 3x1= 3 fg = 2 \* FIELDGOAL ; // 2x3= 6

### total = ( td + pat + fg ) ; // 24+3+6=33

### // Вывод вычисленной суммы.

### System.out.println( "Очков всего: " + total ) ;

### }

### }

### На заметку

Можете добавить опера- торы, которые попытаются изменить значение констант, а затем попробовать заново скомпилировать программу: вы увидите сообщение об ошибке.



Сохранив программу с комментариями, откомпи- лировав и запустив ее, вы увидите, что она выпол- няется так же, как и раньше, без ошибок.

**Проблемы отладки**

**1. Введение**

Иногда компилятор **javac** или интерпретатор **java** сообщают об ошиб- ках. Поэтому полезно понимать причину этих ошибок и знать, как бы- стро решить данную проблему. Для демонстрации некоторых общих сообщений об ошибках приведем следующий пример:



**JAVA**

### class test

### {

*Test.java*

### public static void main ( String[] args )

### {

### String text ;

### System.out.println( "Тест " + text )

### }

### }

Первая попытка скомпилировать *Test.java* вызовет следующее сообще- ние об ошибке.

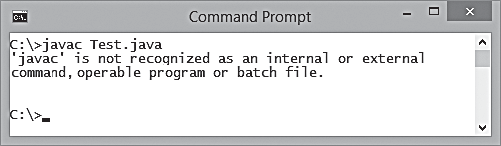
**24**

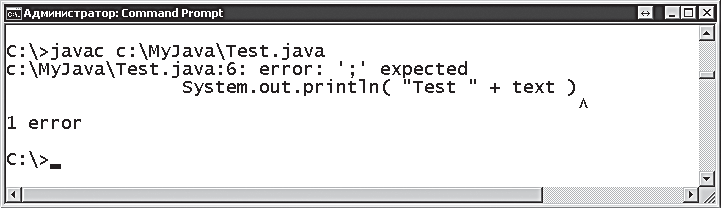
* + Причина — компилятор **javac** не найден.
  + Решение — отредактируйте системную переменную **PATH**, как опи- сано ранее в этой главе, либо используйте полный путь до компи- лятора, чтобы запустить его.



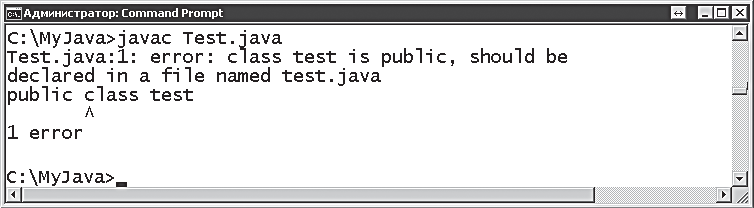
**Совет**

Полный путь к файлу нужно включать в кавычки, если он содержит пробелы, напри- мер, **"C:\Program Files\Java"**.

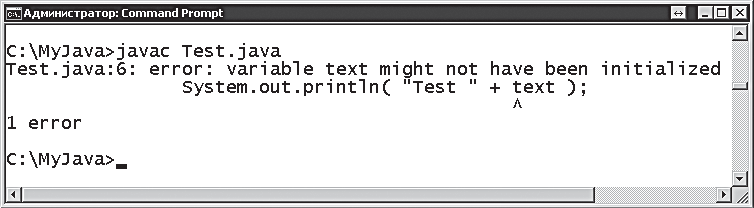
* Причина — файл *Test.java* не найден.
* Решение — либо перейдите в каталог, где размещен файл, либо ис- пользуйте полный путь к файлу в самой команде.



* + Причина — оператор завершен некорректно.
  + Решение — в исходном коде добавьте точку с запятой в конец опе- ратора, затем сохраните файл, чтобы принять изменения.



* + Причина — имя программы и имя класса не совпадают.
  + Решение — измените в исходном коде имя класса с **test** на **Test**, затем сохраните файл для применения изменений.



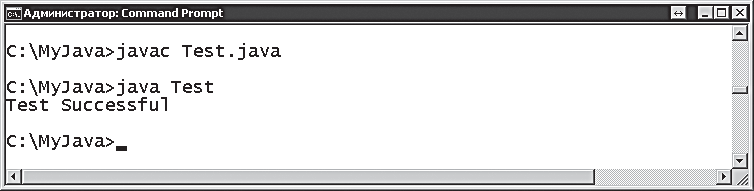
**25**



**Внимание**

* + Причина — переменная **text** не имеет значения.
  + Решение — в объявлении переменной присвойте переменной **text** корректное строковое значение, например, **text = "Successful"**, затем сохраните файл.

Вы должны запускать про- грамму из того каталога, где она находится, — нельзя использовать полный путь к файлу, поскольку ин- терпретатор **java** требует не имя файла, а имя про- граммы.



**Заключение**

**1. Введение**

* + - Java является как языком программирования, так и исполняемой платформой.
    - Программы на Java можно писать в виде обычного текста и сохра- нять в файлах с расширением *.java*.
    - Задача компилятора **javac** заключается в компиляции файлов про- грамм *.class* из первоначальных файлов исходного кода *.java*.
    - Интерпретатор **java** исполняет скомпилированные программы, используя экземпляр виртуальной машины Java (Java VM).
    - Виртуальная машина Java доступна на многих операционных си- стемах.
    - Добавление подкаталога *bin* в системную переменную **PATH** позво- ляет запускать компилятор **javac** из любого расположения.
    - Язык Java чувствителен к регистру.
    - Стандартный метод **main** является точкой входа в программы на Java.
    - Оператор **System.out.println()** предназначен для вывода текста.
    - Имя файла программы на Java должно полностью совпадать с име- нем класса.

**26**

* + - Переменные в Java должны быть названы в соответствии с опреде- ленными соглашениями по именованию, и при этом не должны ис- пользоваться ключевые слова языка.
    - Каждый оператор в Java должен быть завершен символом точки с запятой.
    - Наиболее распространенные типы данных в Java — это **String** (строковые), **int** (целочисленные), **char** (символьные), **float** (с пла- вающей точкой) и **boolean** (логические).
    - Значения строкового типа должны быть заключены в двойные ка- вычки, символьные — в одинарные кавычки, а значения с плаваю- щей точкой должны содержать суффикс **f**.
    - Для создания переменных, являющихся константами, использует- ся ключевое слово **final**.
    - В любую Java-программу можно добавлять многострочные ком- ментарии, находящиеся между символами **/\*** и **\*/**, либо одностроч- ные, указываемыми после символов **//**.

Проблемы компиляции и исполнения идентифицируются с помощью сообщения об ошибках.

**Выполнение операций**

**2**

***Данная глава демонстрирует применение различных операторов, использующихся для создания выражений***

***в Java-программах.***

* **Выполнение арифметических операций**
* **Присваивание значений**
* **Сравнение величин**
* **Оценочная логика**
* **Проверка условий**
* **Расстановка акцентов**
* **Управляющие литералы**
* **Работа с битами**
* **Заключение**

**Выполнение арифметических операций**

**2. Выполнение операций**

Арифметические операции, перечисленные в таблице ниже, использу- ются для создания в Java-программах выражений, которые возвращают одно результирующее значение. Например, выражение **4 \* 2** возвра- щает значение **8**:

|  |  |
| --- | --- |
| **Оператор Операция** | |
| **+** | Сложение (и конкатенация строк) |
| **-** | Вычитание |
| **\*** | Умножение |
| **/** | Деление |
| **%** | Деление по модулю (Остаток от деления) |
| **++** | Инкремент |
| **--** | Декремент |

**28**

Операторы «инкремент» и «декремент» обычно ис- пользуются для подсчета итераций в конструкциях с циклами, которые рас-

сматриваются в следующей главе.

Операторы инкремент (**++**) и декремент (**--**) возвращают результат из- менения единственного операнда на единицу. Например, **4++** возвра- щает значение **5**, а **4--**возвращает значение **3**.

Все другие арифметические операторы возвращают результат опера- ции над двумя заданными операндами и работают обычным образом. Например, выражение **5 + 2** возвращает **7**.



**Совет**

Оператор деления по модулю возвращает остаток от операции деления первого операнда на второй. Например, **32 % 5** возвращает **2** — так как 32 делится на 5: получается 6 и в остатке 2.

Результат операции, выполняемой оператором сложения (**+**), зависит от типа его операндов. Если операндами являются числовые значения, то возвратится значение суммы этих чисел, но когда в качестве операн- дов выступают строки, то в результате получится одна объединенная строка, включающая текст из обеих строк. Например, выражение **"Java " + "Arithmetic"** возвратит **"Java Arithmetic"**.

Выполните следующие шаги для создания Java-программы, выполняю- щей некоторые арифметические операции.

1. Создайте новую программу с именем **Arithmetic**, содержащую стандартный метод **main**.

### class Arithmetic

### {

### public static void main( String[] args ) { }

### }

1. Внутри фигурных скобок главного метода объявите и проинициа- лизируйте три целочисленные переменные.

### int num = 100 ; int factor = 20 ; int sum = 0 ;

1. Добавьте следующие строки для выполнения операций сложения

и вычитания, а также вывода соответствующих результатов.

### sum = num + factor ;// 100 + 20

### System.out.println( "Результат сложения: " + sum ) ; sum = num - factor ;// 100 - 20

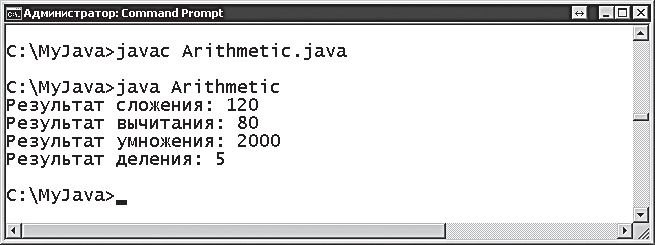
### System.out.println( "Результат вычитания: " + sum ) ;

1. Теперь добавьте строки, выполняющие операции умножения и де- ления, а также выводящие результаты.

### sum = num \* factor ;// 100 x 20

### System.out.println( "Результат умножения: " + sum ) ; sum = num / factor ;// 100 ÷ 20

### System.out.println( "Результат деления: " + sum ) ;

1. Сохраните программу под именем *Arithmetic.java*, затем скомпили- руйте и запустите.



**JAVA**

*Arithmetic.java*

**29**



**На заметку**

В операторах, выводящих результат, используется оператор + для конкатена- ции в одну строку текстовой строки и целого числа, со- ставляющего результат.

**Присваивание значений**

**2. Выполнение операций**

Операторы присваивания, перечисленные в таблице ниже, использу- ются, чтобы занести результат выражения в переменную. Все из них, за исключением оператора **=**, являются сокращенной формой от более длинного эквивалентного выражения.

|  |  |  |
| --- | --- | --- |
| **Оператор Пример Эквивалент** | | |
| **=** | **a = b** | **a = b** |
| **+=** | **a += b** | **a = a + b** |
| **-=** | **a -= b** | **a = a - b** |
| **\*=** | **a \*= b** | **a = a \* b** |
| **/=** | **a /= b** | **a = a / b** |
| **%=** | **a %= b** | **a = a % b** |

**30**



**На заметку**

Оператор равенства **==** срав- нивает значения и подробно описан далее в этой главе.

Следует отличать оператор **=**, означающий присваивание, от оператора равенства **==**.

В выражении **a = b** значение, хранящееся в переменной с именем **b**, присваивается переменной с именем **a**. Таким образом, в **a** заносится новое значение, замещая любое, которое там содержалось раньше.

Оператор **+=** используется для добавления какого-либо значения к зна- чению, хранящемуся в переменной, сохраняя в этой переменной «теку- щую сумму».

В выражении **a += b** сначала вычисляется общая сумма значений, хранящихся в переменных **a** и **b**, затем результат присваивается пере- менной **a**. В программе могут после этого содержаться и дальнейшие присваивания, например **a += c**, которые вычисляют сумму значений, хранящихся в **a** и **с**, а затем присваивают эту сумму переменной **a**, до- бавляя значение переменной **c** к предыдущему значению, хранящемуся в **a**.

Все остальные операторы присваивания работают по тому же прин- ципу, выполняя сначала арифметическое вычисление над двумя хра- нящимися значениями, а затем, присваивая результат первой перемен- ной, заносят в нее новое значение.

Выполните следующие шаги для создания Java-программы, в которой выполняются различные операторы присваивания.

1. Создайте новую программу с именем **Assignment**, содержащую стандартный метод **main**.



**JAVA**

### class Assignment

### {

### public static void main( String[] args ) { }

### }

*Assignment.java*

1. Внутри фигурных скобок главного метода добавьте следующие строки кода, в которых складывается и присваивается строковое значение.

**31**

### String txt = "Fantastic " ; String lang = "Java" ;

### txt += lang ; // Присваивание с объединением строк System.out.println( "Складываем и присваиваем строки: " + txt ) ;

1. Добавьте эти строки для сложения и присваивания целых чисел.

### int sum = 10 ; int num = 20 ;

### sum += num ; // Присваиваем результат ( 10 + 20 = 30 ) System.out.println( " Складываем и присваиваем целые числа: " + sum ) ;

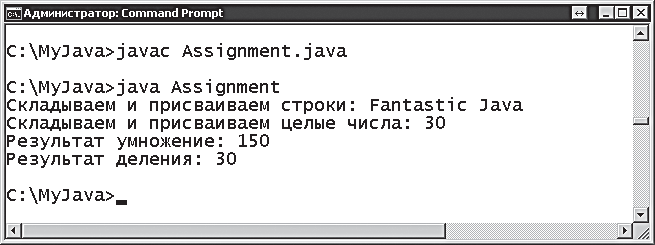
1. Теперь добавьте строки, в которых производится умножение и при- сваивание целых чисел.

### int factor = 5 ;

### sum \*= factor ; // Присваиваем результат ( 30 x 5 = 150 ) System.out.println( "Результат умножения " + sum ) ;

1. Добавьте строки, выполняющие деление и присваивание целых чисел.

### sum /= factor ; // Присваиваем результат ( 150 ÷ 5 = 30 ) System.out.println( " Результат деления: " + sum ) ;

1. Сохраните программу под именем *Assignment.java*, затем скомпи- лируйте и запустите.



**Внимание**

Присваивание переменной значения неверного типа приводит к сообщению об ошибке.

**Сравнение величин**

**2. Выполнение операций**

Операторы сравнения, перечисленные в таблице ниже, используются для оценки двух значений в выражении и возвращают логическое зна- чение **true** или **false**, описывающее результат этого сравнения.

|  |  |
| --- | --- |
| **Оператор Операция сравнения** | |
| **==** | Равно |
| **!=** | Не равно |
| **>** | Больше |
| **>=** | Больше или равно |
| **<** | Меньше |
| **<=** | Меньше или равно |

**32**

### Совет

Оператор меньше **<** обычно используется для проверки значения счетчика в цикле – пример вы можете найти

в главе 3.

Оператор равенства **==** оценивает два операнда и возвращает **true**, если значения операндов равны. Если операнды являются одними и теми же числами, то они равны; если они являются строками, содержащими одни и те же символы в одном и том же порядке, то они тоже равны. Операнды логического типа равны, если оба содержат **true** или оба со- держат **false**.

Оператор неравенства **!=** наоборот, возвращает **true**, если значения операндов не равны, используя те же правила, что и оператор равен- ства.

Операторы равенства и неравенства часто применяются для выполне- ния так называемого условного ветвления в программе, когда в зави- симости от условия выбирается дальнейшее направление работы про- граммы.

Оператор «больше» (**>**) сравнивает два операнда и возвращает **true**, если значение первого больше, чем значение второго.



Оператор «меньше» (**<**) делает то же самое сравнение, но возвращает

**true** в случае, если значение первого операнда меньше.

Операторы «больше либо равно» и «меньше либо равно» возвращают также значение **true**, если значения обоих операндов равны.

Выполните следующие шаги для создания Java-программы, выполняю- щей различные операции сравнения.

1. Создайте новую программу с именем **Comparison**, содержащую стандартный метод **main**.



**JAVA**

### class Comparison

### {

### public static void main( String[] args ) { }

### }

*Comparison.java*

1. Внутри фигурных скобок главного метода добавьте следующие строки кода для сравнения двух строковых переменных на равен- ство.

**33**

### String txt = "Fantastic " ; String lang = "Java" ;

### boolean state = ( txt == lang ) ; // Присваиваем результат проверки System.out.println( "Проверка строк на равенство: " + state ) ;



1. Добавьте следующие строки для проверки на неравенство.

### state = ( txt != lang ) ; // Присваиваем результат System.out.println( " Проверка строк на неравенство: " + state ) ;

1. Теперь добавьте строки для сравнения двух целочисленных пере- менных.

### int dozen = 12 ; int score = 20 ;

### state = ( dozen > score ) ; // Присваиваем результат System.out.println( "Проверка на больше: " + state ) ;

1. Добавьте еще две строки для сравнения целочисленных перемен- ных еще раз.

### state = ( dozen < score ) ; // Присваиваем результат System.out.println( " Проверка на меньше: " + state ) ;



1. Сохраните программу под именем *Comparison.java*, затем скомпи- лируйте и запустите.

**Совет**

Обратите внимание, что про- веряемое выражение можно заключить в скобки для лучшей читаемости.

**На заметку**

В данном случае неверно (**false**), что строковые значе- ния равны, но верно (**true**), что они не равны.

**Оценочная логика**

**2. Выполнение операций**

Логические операторы, перечисленные в таблице ниже, используются чтобы объединить несколько выражений, каждое из которых возвра- щает логическое значение, в одно сложное выражение, которое будет возвращать единственное логическое значение.

|  |  |
| --- | --- |
| **Оператор Операция** | |
| **&&** | Логическое И |
| **||** | Логическое ИЛИ |
| **!** | Логическое НЕ (отрицание) |

Логические операторы работают с операндами, имеющими значение логического (булевого) типа, то есть **true** или **false**, либо со значения- ми, которые преобразуются в **true** или **false**.

Оператор «логическое И» (**&&**) оценивает два операнда и возвращает значение **true**, только если оба операнда сами имеют значение **true**, в противном случае оператор возвращает значение **false**. Этот опера- тор обычно используется при условном ветвлении, когда направление работы программы определяется проверкой двух условий: если оба они верны, программа идет в определенном направлении, в противном случае — в другом.

**34**

В отличие от оператора **and**, которому необходимо, чтобы оба операн- да имели значение **true**, оператор «логическое ИЛИ» (**||**) оценивает два операнда и возвращает **true**, если хотя бы один из них сам возвраща- ет значение **true**. В противном случае оператор **||** возвратит значение **false**. Это полезно использовать при программировании определенных действий в случае выполнения одного из двух проверяемых условий.

Оператор «логическое НЕ» (**!**) является унарным и используется с од- ним операндом. Он возвращает противоположное значение от того, какое имел операнд. Так, если переменная **a** имела значение **true**, то

**!a** возвратит значение **false**. Он может использоваться, например, для

переключения значения переменной в последовательных итерациях цикла при помощи выражения **a=!a**. Это значит, что на каждой итера- ции цикла логическое значение меняется на противоположное, подоб- но выключению и включению лампочки.

Выполните следующие шаги для создания Java-программы, использую- щей три логических оператора.

* 1. Создайте новую программу с именем **Logic**, содержащую стандарт- ный метод **main**.

### class Logic



**JAVA**

### {

### public static void main( String[] args ) { }

### }

*Logic.java*

* 1. Внутри фигурных скобок главного метода добавьте следующие строки с объявлением и инициализацией двух переменных логиче- ского типа.

**35**

### boolean yes = true ; boolean no = false ;

* 1. Добавьте строки, содержащие проверки двух условий на их истин- ность.

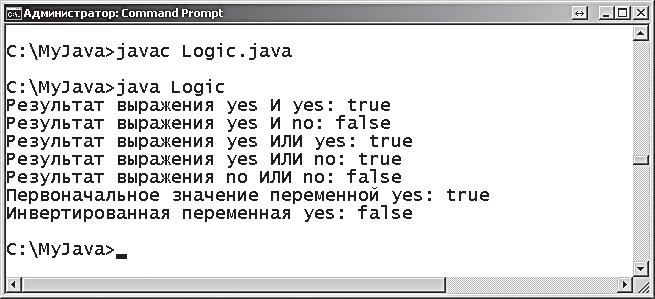
### System.out.println( "Результат выражения yes И yes: " + ( yes && yes ) ) ; System.out.println( " Результат выражения yes И no: " + ( yes && no ) ) ;

* 1. Добавьте строки для проверки, что одно из двух условий истинно.

### System.out.println( " Результат выражения yes ИЛИ yes: " + ( yes || yes ) ) ; System.out.println( " Результат выражения yes ИЛИ no: " + ( yes || no ) ) ; System.out.println( "Результат выражения no ИЛИ no: " + ( no || no ) ) ;

* 1. Добавьте строки, показывающие первоначальное и противополож- ное значения логической переменной.

### System.out.println( "Первоначальное значение переменной yes: " + yes ) ; System.out.println( "Инвертированная переменная yes: " + !yes ) ;

* 1. Сохраните программу под именем *Logic.java*, затем скомпилируйте и запустите.



**Совет**

Логический тип данных (**boolean**) назван в честь ма- тематика Джорджа Буля — создателя логической алгебры.

**36**

Условный оператор также называют «тернарным».

**2. Выполнение операций**

**Проверка условий**

Наверное, одним из самых любимых Java-программистами операторов является условный оператор, который делает объемную конструкцию очень краткой. Хотя его необычный синтаксис кажется на первый взгляд слишком запутанным, все равно стоит познакомиться с этим полезным оператором.



**Совет**

Условный оператор вначале оценивает выражение на предмет значе- ний **true** или **false**, а затем возвращает один из двух своих операндов в зависимости от результатов оценки. Его синтаксис выглядит следую- щим образом:

#### ( *логическое-выражение* ) ? *если-истина-возвращаем-это* : *если-ложь-возвращаем-это*;

Каждый из указанных операндов является альтернативной веткой ис- полнения программы в зависимости от логической величины, возвра- щаемой проверочным выражением. Например, в качестве вариантов могут возвращаться строковые значения:

### status = ( quit == true ) ? "Готово!" : "В процессе..." ;

В данном случае, когда переменная **quit** содержит значение **true**, услов- ный оператор присваивает значение своего первого операнда (**Готово!**) переменной **status**, а в противном случае оператор присваивает ей зна- чение второго операнда (**В процессе...**).

Когда проверяется простая логическая величина, то доступна краткая запись проверочного выражения, в этом случае можно опустить сим- волы **== true**. Таким образом, вышеприведенный пример может быть записан в виде:

### status = ( quit ) ? " Готово!" : " В процессе..." ;

Условный оператор может возвращать значение любого типа данных и использовать любое действительное проверочное выражение. На- пример, для оценки двух числовых значений может использоваться оператор сравнения **>** и возвращаться логическое значение в зависи- мости от результата:

### busted = ( speed > speedLimit ) ? true : false ;

Аналогично в проверочном выражении может использоваться опера- тор неравенства **!=** для оценки строковой величины и возвращать, на- пример, числовое значение в зависимости от результата:

### bodyTemperature = ( scale != "Celsius" ) ? 98.6 : 37.0 ;

Выполните следующие шаги для создания Java-программы, использую- щей условный оператор.

1. Создайте новую программу с именем **Condition**, содержащую стан- дартный метод **main**.

### class Condition

### {

### public static void main( String[] args ) { }

### }

1. Внутри фигурных скобок главного метода добавьте следующие строки, объявляющие и инициализирующие две целочисленные переменные.

### int num1 = 1357 ; int num2 = 2468 ;

1. Далее объявите строковую переменную для хранения результата

проверки.

### String result ;

1. Добавьте строки для определения, является ли значение первой переменной четным или нечетным числом, также добавьте вывод результата.

### result = ( num1 % 2 != 0 ) ? "Нечетное" : "Четное" ; System.out.println( num1 + " - " + result ) ;

1. Добавьте следующие строки для аналогичной проверки значения

второй целочисленной переменной.

### result = ( num2 % 2 != 0 ) ? " Нечетное " : " Четное " ; System.out.println( num2 + " - " + result ) ;

1. Сохраните программу под именем *Condition.java*, затем скомпили-

руйте и запустите.



**JAVA**

*Condition.java*

**37**



**На заметку**

В данном случае выражение будет истинным, когда есть какой-либо остаток.

**2. Выполнение операций**

Операторы, имеющие одинаковый приоритет, об- рабатываются в порядке их появления в выражении — слева направо.

**38**

**Приритет операций**

Сложные составные выражения, которые содержат несколько операто- ров и операндов, могут быть двусмысленными до тех пор, пока не опре- делить, в каком порядке выполнять операции. Такая неясность может привести к тому, что одно и то же выражение даст разные результаты. Например, выражение:

### num = 8 + 4 \* 2 ;

Если выполнять операции слева направо, то **8 + 4 = 12** и **12 \* 2 = 24**. Таким образом, **num = 24**. Если действовать справа налево, то **2 \* 4 = 8**, и **8 + 8 = 16**. Таким образом, **num = 16**.

Java-программисты могут с помощью скобок явно указывать, какая из операций будет выполняться первой, заключая в скобки оператор, ко- торый имеет более высокий приоритет. В таком случае **(8 + 4) \* 2** означает, что сложение будет выполняться перед умножением, и в ре- зультате будет получаться 24, а не 16. И наоборот, **8 + (4 \* 2)** выпол- няет вначале умножение и результатом будет 16.

В случае, когда порядок операторов явно не указан, используется при- оритет операторов, заданный по умолчанию, который представлен в таблице ниже. Здесь представлены операторы в порядке убывания приоритетов.



**Совет**

|  |  |
| --- | --- |
| **Оператор Описание** | |
| **++ -- !** | Инкремент, декремент, логическое НЕ |
| **\* / %** | Умножение, деление, деление по модулю |
| **+ -** | Сложение, вычитание |
| **> >= < <=** | Больше, больше или равно, меньше, меньше или равно |
| **== !=** | Равно, не равно |
| **&&** | Логическое И |
| **||** | Логическое ИЛИ |
| **? :** | Тернарная условная операция |
| **= += -= \*= /= %=** | Присваивание |

Выполните следующие шаги для создания Java-программы, работаю- щей с различным приоритетом операторов.

1. Создайте новую программу с именем **Precedence**, содержащую стандартный метод **main**.

### class Precedence

### {

### public static void main( String[] args ) { }

### }

1. Внутри фигурных скобок главного метода добавьте следующий код для объявления и инициализации целочисленной переменной, со- держащей результат выражения, используя приоритет операторов по умолчанию.

### int sum = 32 - 8 + 16 \* 2 ; // 16 x 2 = 32, + 24 = 56 System.out.println( "Порядок действий по умолчанию: " + sum ) ;

1. Добавьте строки для присваивания переменной результата того же

самого выражения, но с заданием приоритета для операций вычи- тания и сложения.

### sum = ( 32 - 8 + 16 ) \* 2 ; // 24 + 16 = 40, x 2 = 80

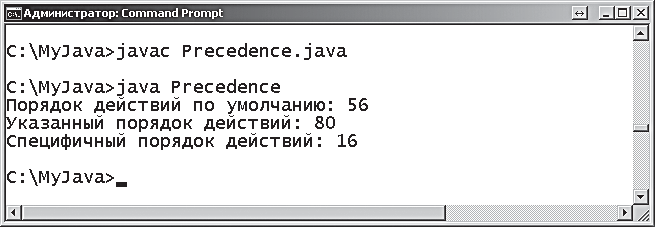
### System.out.println( " Указанный порядок действий: " + sum ) ;

1. Добавьте следующие строки, в которых в переменную заносится результат выражения, где задан приоритет операций по убыванию: сначала сложение, затем вычитание, потом умножение.

### sum = ( 32 - (8 + 16) ) \* 2 ; // 32 - 24 = 8, \* 2 = 16

### System.out.println( " Специфичный порядок действий: " + sum ) ;

1. Сохраните программу под именем *Precedence.java*, затем скомпили- руйте и запустите.





**JAVA**

*Precedence.java*

**39**



**На заметку**

При использовании вложен- ных скобок в выражениях высший приоритет имеют внутренние скобки.

**Управляющие литералы**

**2. Выполнение операций**

Числовые и текстовые значения в Java-программах известны как «лите- ралы» — на самом деле они ничего собой не представляют, это просто символы, которые вы видите.

Литералы обычно отделяются от ключевых слов Java-языка, но там, где требуются двойные или одинарные кавычки внутри строковой пере- менной, необходимо указать, что символ кавычек должен как-то отде- ляться, чтобы избежать аварийного завершения строки. Это достига- ется путем добавления перед каждым символом кавычек управляющей последовательности (или управляющего оператора) **\**. Например, что- бы включить кавычки в строковую переменную, пишем следующее:

### String quote = " \"Удача благоволит храбрым.\" сказал Вергилий ";

Для форматирования простейшего вывода можно использовать раз- личные управляющие последовательности:

**40**

|  |  |
| --- | --- |
| **Управляющий символ Описание** | |
| **\n** | Перевод строки |
| **\t** | Табуляция |
| **\b** | Шаг назад (забой) |
| **\r** | Возврат каретки |
| **\f** | Перевод страницы |
| **\\** | Обратный слеш |
| **\'** | Одиночная кавычка (апостроф) |
| **\"** | Двойная кавычка |

В качестве альтернативы управляющей последова- тельности символов внутри двойных кавычек можно использовать одинарные кавычки.



**Совет**

Управляющая последовательность для новой строки (**\n**) часто приме- няется для вывода нескольких строк. Аналогично, управляющая после- довательность табуляции (**\t**) используется для вывода содержимого в столбцах. Использование комбинаций новой строки и табуляции по- зволяет форматировать вывод как в строках, так и в столбцах — в виде некоторой таблицы.

Выполните следующие шаги для создания Java-программы, использую- щей управляющие последовательности для форматирования вывода.

**41**

1. Создайте новую программу с именем **Escape**, содержащую стан- дартный метод **main**.

### class Escape

### {

### public static void main( String[] args ) { }

### }

1. Внутри фигурных скобок главного метода добавьте следующие строки кода для создания строковой переменной, содержащей форматированные названия таблицы и заголовки столбцов.

### String header = "\n\tНЬЮ-ЙОРК ПРОГНОЗ НА 3 ДНЯ:\n" ; header += "\n\tДень\t\tМакс\tМин\tОсадки\n" ;

### header += "\t---\t\t----\t---\t----------\n" ;

1. Добавьте следующие строки для создания строковой переменной, содержащей форматированные данные и ячейки таблицы.

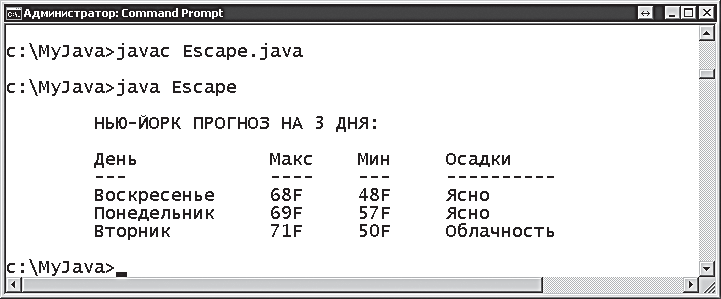
### String forecast = "\tВоскресенье\t68F\t48F\tЯсно\n" ; forecast += "\tПонедельник\t69F\t57F\tЯсно\n" ; forecast += "\tВторник\t\t71F\t50F\tОблачность\n" ;

1. Добавьте строку для вывода обоих значений форматированных

строк.

### System.out.print( header + forecast ) ;

1. Сохраните программу под именем *Escape.java*, затем скомпилируй- те и запустите.





**JAVA**

*Escape.java*



**На заметку**

В данном случае управляю- щая последовательность добавляет символ новой строки, поэтому здесь ис- пользуется метод **print()**, а не **println()**, который добавляет символ новой строки после вывода.

**2. Выполнение операций**

Не путайте логический оператор И (**&&**) с побитовым оператором (**&**), а также ло- гический оператор ИЛИ (**||**) с побитовым оператором (**|**).



**Внимание**

**42**

**Работа с битами**

В дополнение к обычным операторам, описанным в этой главе, язык Java предлагает специальные операторы для работы с бинарными зна- чениями. Они, конечно, используются, в отличие от обычных опера- торов, гораздо реже, но для общего знакомства приведем краткий их обзор.

«Побитовые» операторы в языке Java можно использовать с данными целочисленного типа для того, чтобы манипулировать битами дво- ичного значения. Для этого нужно понимать, как десятичные числа от 0 до 255 представляются в виде 8 бит.

Например, десятичное число 53 представляется двоичным **00110101**

( **0**×128,**0**×64,**1**×32,**1**×16,**0**×8,**1**×4,**0**×2,**1**×1 ).

Операции двоичного сложения выполняются подобно арифметиче- ским операциям над десятичными числами.

### 53 = 00110101

### + 7 = 00000111

### 60 = 00111100

Кроме того, побитовые операторы предлагают специальные операции над двоичными числами.

|  |  |  |  |
| --- | --- | --- | --- |
| **Оператор Операция Пример Результат** | | | |
| **&** | И | **a & b** | 1, если оба бита 1 |
| **|** | ИЛИ | **a | b** | 1, если хотя бы один бит 1 |
| **^** | Исключающее ИЛИ | **a ^ b** | 1, если биты отличаются |
| **~** | НЕ (отрица- ние) | **~a** | Изменяет значение бита на противоположное |
| **<<** | Сдвиг влево | **n << p** | Сдвигает биты значения **n**  на **p** позиций влево |
| **>>** | Сдвиг вправо | **n >> p** | Сдвигает биты значения **n**  на **p** позиций вправо |

Например, используя побитовый оператор И в двоичной арифметике, мы запишем:

### 53 = 00110101

### & 7 = 00000111

### 5 = 00000101

Очень распространенное применение побитовых операторов включа- ется в возможности использовать в одной переменной несколько зна- чений. Например, программа с несколькими целочисленными пере- менными (флагами), имеющими значение 1 или 0 (представляющие состояние «включено» и «выключено»), обычно требует 8 бит памяти. Эти значения требуют одного бита, однако в одной переменной можно скомбинировать до 8 флагов, используя 1 бит для каждого флага. Состо- яние каждого флага можно вызвать с помощью побитовой операции:

1. Создайте новую программу с именем **Bitwise**, содержащую стан- дартный метод **main**.



**JAVA**

### class Bitwise

### {

### public static void main( String[] args ) { }

### }

**43**

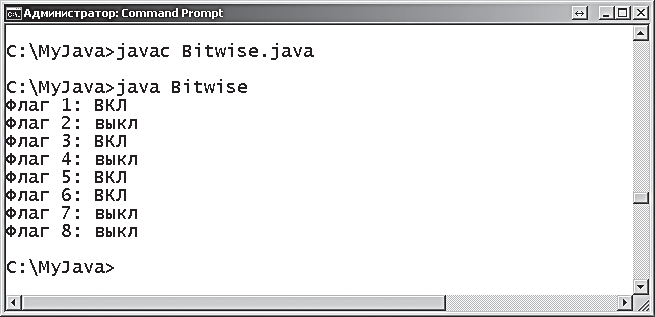
*Bitwise.java*

1. Внутри фигурных скобок главного метода добавьте следующий код, объявляющий и инициализирующий целочисленную переменную со значением, представляющим общее состояние восьми флагов.

### int fs = 53 ; // Двоичное представление 00110101

### Добавьте следующие строки для вызова состояния каждого флага. System.out.println("Флаг 1: "+(( (fs&1)>0) ? "ВКЛ" : "выкл")); System.out.println("Флаг 2: "+(( (fs&2)>0) ? "ВКЛ" : "выкл")); System.out.println("Флаг 3: "+(( (fs&4)>0) ? "ВКЛ" : "выкл")); System.out.println("Флаг 4: "+(( (fs&8)>0) ? "ВКЛ" : "выкл")); System.out.println("Флаг 5: "+(( (fs&16)>0)? "ВКЛ" : "выкл")); System.out.println("Флаг 6: "+(( (fs&32)>0)? "ВКЛ" : "выкл")); System.out.println("Флаг 7: "+(( (fs&64)>0)? "ВКЛ" : "выкл")); System.out.println("Флаг 8: "+(( (fs&128)>0)?"ВКЛ": "выкл"));

1. Сохраните программу под именем *Bitwise.java*, затем скомпилируй-

те и запустите.



**На заметку**

В данном случае побитовая операция возвращает значе- ние 1 или 0, определяющее состояние каждого флага.

**Заключение**

**2. Выполнение операций**

* + Арифметические операторы можно использовать для формирова- ния выражений с двумя операндами для сложения (**+**), вычитания (**-**), умножения (**\***), деления (**/**) и деления по модулю (**%**).
  + Операторы инкремента (**++**) и декремента (**--**) изменяют един- ственный операнд на единицу.
  + Оператор присваивания (**=**) можно комбинировать с арифметиче- скими операторами, чтобы выполнять арифметические вычисле- ния, а затем присваивать результат.
  + Операторы сравнения можно использовать для формирования вы- ражений, оценивающих два операнда на равенство (**==**), неравен- ство (**!=**), больше (**>**) или меньше (**<**).
  + Оператор присваивания может быть скомбинирован с оператора- ми **>** и **<**, и он будет возвращать значение **true** при равенстве опе- рандов.
  + Логические операторы И (**&&**) и ИЛИ (**||**) формируют выражения, оценивающие два операнда и возвращающие логические значения **true** или **false**.

**44**

* + Логический оператор **!** возвращает обратную логическую величи- ну от значения единственного операнда.
  + Условный оператор **? :** оценивает заданное логическое выражение и возвращает один из двух операндов в зависимости от результа- та.
  + При оценке логических выражений на предмет истинности можно опускать знак **== true**.
  + В сложных арифметических выражениях важно явно указывать порядок операторов с помощью добавления скобок.
  + Во избежание синтаксических ошибок кавычки в строковых пере- менных нужно предварять управляющим оператором **\**.
  + Управляющие последовательности **\n** (новая строка) и **\t** (табуля- ция) организуют простое форматирование вывода.
  + В особых ситуациях для выполнения двоичных арифметических операций полезно использовать побитовые операторы.

**Создание операторов**

**3**

***В данной главе демонстрируется использование различных ключевых слов для создания ветвлений***

***в Java-программах.***

* **Ветвление с помощью условного оператора if**
* **Альтернативное ветвление**
* **Ветвление с помощью переключателей**
* **Цикл for**
* **Цикл while**
* **Циклы do-while**
* **Выход из циклов**
* **Возврат управления**
* **Заключение**

**Ветвление с помощью условного оператора if**

**3. Создание операторов**

Ключевое слово **if** выполняет условную проверку, оценивая логиче- ское значение выражения. Оператор, следующий за этим выражением, будет выполняться, только если данное значение равно **true** (истина). В противном случае программа переходит на последующие строки кода, выполняя альтернативное ветвление. Синтаксис оператора **if** вы- глядит следующим образом:

#### if ( *проверочное-выражение* ) *код-для-исполнения-если-результат-истина* ;

Исполняемый код может содержать несколько операторов, при этом они образуют блок операторов и заключаются в фигурные скобки.

1. Создайте новую программу с именем **If**, содержащую стандартный метод **main**.



**JAVA**

### class If

### {

*If.java*

### public static void main ( String[] args) { }

### }

1. Внутри фигурных скобок главного метода добавьте следующую простую проверку, в которой, при условии, что одно число больше другого, выполняется один оператор (в данном случае выполняет- ся вывод на печать).

**46**

### if ( 5 > 1 ) System.out.println( "Пять больше чем один." ) ;

1. Добавьте вторую проверку, которая, при условии, что одно число меньше другого, будет выполнять блок операторов.

### if ( 2 < 4 )

### {

### System.out.println( "Два меньше четырех." ) ; System.out.println( "Проверка выполнена успешно." ) ;

### }

В данном случае ключевые слова **true** и **false** опущены, поскольку проверочное выражение **(2 < 4)** — это

краткая запись для **(2 < 4**

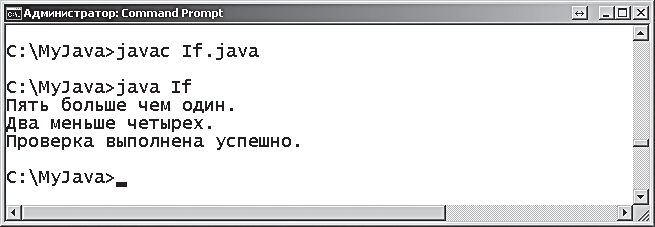
### == true).

1. Сохраните программу под именем *If.java*, затем скомпилируйте и запустите. Поскольку обе проверки дадут в результате значение **true**, то в обоих случаях все операторы выполнятся.

В проверяемом выражении может находиться сложное составное выражение. При этом проверяется несколько условий на предмет логической величины. Можно отделять скобками отдельные вы- ражения для установления порядка вычислений — выражения в скобках будут оцениваться первыми.



**Совет**



Логический оператор И (**&&**) будет возвращать **true**, только если оба проверяемых выражения имеют значение **true**:

**47**

### if ( ( условие-1 ) && ( условие-2 ) ) исполнить-этот-код ;

Логический оператор ИЛИ (**||**) будет возвращать значение **true**, если хотя бы одно из проверочных выражений имеет значение **true**:

### if ( (условие-1 ) || ( условие-2) ) исполнить-этот-код ;

Комбинации вышеуказанных выражений могут составлять более сложные и длинные проверочные выражения.

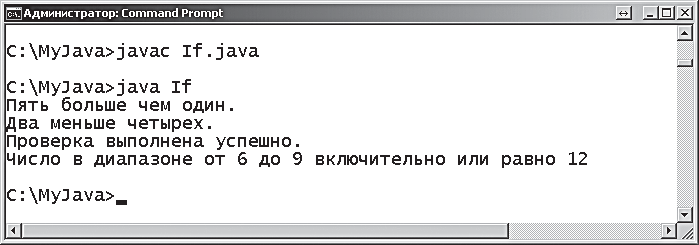
1. Добавьте следующую строку внутрь главного метода, для того что- бы объявить и проинициализировать целочисленную переменную с именем **num**.

### int num = 8 ;

1. Добавьте третье проверочное выражение, которое будет выпол- нять оператор вывода в том случае, если значение переменной **num** находится внутри указанного диапазона или в точности равно ука- занному значению.

### if ( ( ( num > 5 ) && ( num < 10 ) ) || ( num == 12 ) ) System.out.println( "Число в диапазоне от 6 до 9 включительно или равно 12" ) ;

1. Заново скомпилируйте программу и запустите ее еще раз на вы- полнение, чтобы увидеть, что оператор выполняется после про- хождения сложной проверки.



1. Измените значение, присвоенное переменной **num**, таким образом, чтобы оно не входило в диапазон 6–9 и не было равно 12. Ском- пилируйте заново программу и запустите ее снова, чтобы увидеть, что теперь оператор после сложной проверки не выполняется.



**Совет**

Диапазон может быть расши- рен, если включить верхние и нижние границы, исполь- зуя операторы **>=** и **<=**.



**На заметку**

В проверочном выражении используется оператор равенства **==**, а не оператор присваивания **=**.

Обратите внимание, что первый оператор заверша- ется точкой с запятой перед ключевым словом **else**.

**48**

**3. Создание операторов**

**Альтернативное ветвление**

В дополнение к ключевому слову **if** можно использовать ключевое сло- во **else**, которое вместе с **if** образует оператор **if else**, обеспечиваю- щий альтернативные ветви для продолжения программы в соответ- ствии с результатом оценки проверочного выражения. В простейшем случае он просто предлагает альтернативный оператор для исполне- ния, а когда проверка неуспешна — выдает значение **false**:



**Внимание**

#### if (*проверочное-выражение*)

#### *код-для-исполнения-если-результат-истина*; else

#### *код-для-исполнения-если-результат-ложь* ;

Каждая альтернативная ветвь может являться либо отдельным опе- ратором, либо блоком операторов, заключенным внутри фигурных скобок.

При помощи оператора **if else** можно строить более сложные кон- струкции для осуществления дополнительных проверок внутри каж- дой из ветвей **if** и **else**. При этом получаются вложенные операторы **if**. Когда программа обнаруживает выражение, оцененное как **true**, она исполняет операторы, связанные с этой ветвью, а затем выходит из оператора **if else**.

*Else.java*

1. Создайте новую программу с именем **Else**, содержащую стандарт- ный метод **main**.

### class Else



**JAVA**

### {

### public static void main ( String[] args ) { }

### }

1. Внутри главного метода добавьте следующую строку с объявлени- ем и инициализацией целочисленной переменной **hrs**.

### int hrs = 11 ;

1. Добавьте простое проверочное выражение, которое будет испол- нять один оператор, в случае если значение переменной **hrs** мень- ше 13.

### if ( hrs < 13 )

### {

### System.out.println( "Доброе утро " + hrs ) ;

### }

1. Сохраните программу под именем *Else.java*, затем скомпилируйте и запустите для просмотра исполняемого оператора.



1. Присвойте переменной **hrs** значение **15**, а затем добавьте альтерна- тивную ветвь сразу после оператора **if**.

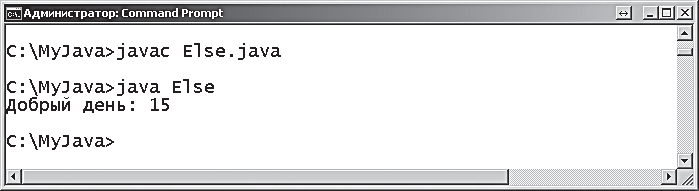
### else if ( hrs < 18 )

### {

### System.out.println( "Добрый день " + hrs ) ;

### }

1. Сохраните изменения, скомпилируйте и запустите заново програм- му, чтобы увидеть, что исполняется альтернативный оператор.

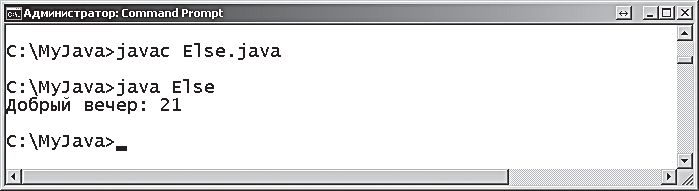


Иногда полезно ставить последнюю ветвь **else** без вложенного оператора **if**, чтобы определить оператор по умолчанию, выпол- няемый в случае, если никакое проверочное выражение не будет истинно.

**49**

1. Присвойте переменной **hrs** значение **21**, затем добавьте следующую ветвь, заданную по умолчанию, в конец оператора **if else**.

### else System.out.println( "Добрый вечер: " + hrs ) ;

1. Сохраните изменения, перекомпилируйте и запустите программу еще раз, чтобы увидеть, что выполняется оператор, заданный по умолчанию.



**На заметку**

Условное ветвление — это фундаментальный процесс, с помощью которого ком- пьютер выполняет про- граммы.

**Ветвление с помощью переключателей**

**3. Создание операторов**

Конструкции с операторами **if else**, предлагающими большое коли- чество условных ветвлений программы, могут стать довольно громозд- кими. В тех случаях, где нужно повторять проверку одного и того же значения переменной, более элегантное решение предлагает оператор **switch** (переключатель). Типичный синтаксис блока оператора **switch** выглядит следующим образом:

**switch ( *проверяемая-переменная* )**

### {

#### *case значение-1 : код-для-исполнения-если-истина* ; break ; *case значение-2 : код-для-исполнения-если-истина* ; break ; *case значение-3 : код-для-исполнения-если-истина* ; break ; default : *код-для-исполнения-если-лож*ь;

### }

**50**

Отсутствие ключевого слова **break** не является синтакси- ческой ошибкой, поэтому, во избежание неожиданных для вас результатов, вы должны убедиться, что все операторы **break** проставле- ны там, где это необходимо.

Оператор **switch** работает довольно необычно. Он проверяет значе- ния, указанные переменной, на соответствие значениям своих опций **case**, а затем выполняет оператор, связанный со значением определен- ной опции.

Последняя опция **default** является необязательной и может добав- ляться в оператор **switch** для указания операторов, которые будут ис- полняться в том случае, если ни одно из указанных значений не соот- ветствует проверяемой переменной.



**Внимание**

Каждая опция оператора начинается с ключевого слова **case** и значе- ния для проверки, для которого стоит символ двоеточия и операторы, исполняемые в случае соответствия значению.

Важно помнить, что оператор и блок операторов, связанный с опреде- ленной опцией **case**, должны завершаться ключевым словом **break**. В противном случае программа будет продолжать исполнять операто- ры другой опцией **case** после той, которая прошла проверку. Иногда это и полезно, чтобы указать несколько опций **case**, для которых нуж- но исполнить один и тот же блок операторов. Например, один оператор для каждой из трех опций выглядит следующим образом:

**switch ( *проверяемая-переменная* )**

### {

**case *значение-1* : case *значение-2* : case *значение-3* :**

***код-А-для-исполнения-если-истина* ; break ;**

**case *значение-4* : case *значение-5* : case *значение-6* :**

***код-B-для-исполнения-если-истина* ; break ;**

### }

1. Создайте новую программу с именем **Switch**, содержащую стан- дартный метод **main**.

### class Switch

### {

### public static void main ( String[] args ) { }



**JAVA**

*Switch.java*

### }

**51**

1. Внутри главного метода добавьте объявление и инициализацию трех целочисленных переменных.

### int month = 2, year = 2016, num = 31 ;

1. Добавьте блок оператора **switch** для проверки значений, присвоен- ных переменной **month**.

### switch ( month )

### {

### }

1. Внутри блока **switch** добавьте опции **case**, присваивая новые зна- чения переменной **num** для месяцев 4, 6, 9 и 11.

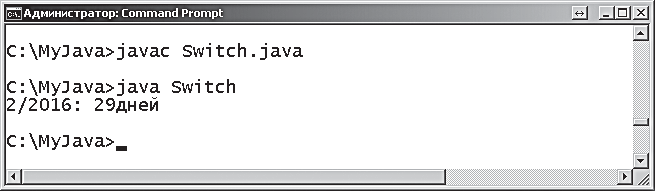
### case 4 : case 6 : case 9 : case 11 : num = 30 ; break ;

1. Добавьте опцию **case** с присваиванием нового значения перемен- ной **num** для месяца 2 в соответствии со значением переменной **year**.

### case 2 : num = ( year % 4 == 0 ) ? 29 : 28 ; break ;

1. После блока **switch** в конце главного метода добавьте следующую строку для вывода всех трех значений переменных.

### System.out.println( month+"/"+year+": "+num+"дней" ) ;

1. Сохраните программу под именем *Switch.java*, затем скомпилируй- те и запустите.



**Совет**

Обратите внимание, что все три целочисленные пере- менные объявлены и про- инициализированы в одной строке, причем использова- но удобное сокращение.



**На заметку**

На шаге 5 используется условный оператор для эф- фективной проверки. Чтобы вспомнить, как он работает, можете вернуться к главе 2.

**Цикл for**

**3. Создание операторов**

Циклом называется блок кода, в котором с определенной периодич- ностью исполняются содержащиеся в нем операторы до тех пор, пока не выполнится определенное условие; затем цикл завершается, и про- грамма переходит к своей следующей задаче.

Наиболее часто используемая структура циклов в программировании на Java использует ключевое слово **for**, и синтаксис данной структуры следующий:

#### for ( *инициализация ; проверочное-выражение ; итерация* )

### {

#### *операторы-для-выполнения-на-каждой-итерации* ;

### }

**52**

Итерацию часто называют счетчиком приращений, поскольку чаще всего она используется для увели- чения значения, а не для уменьшения.

Скобки после ключевого слова **for** должны содержать три управляю- щих выражения, которые и определяют действие цикла.

* **Инициализация** — присваивает начальное значение переменной- счетчику, который будет подсчитывать число итераций цикла. Переменная для этих целей может быть объявлена прямо здесь, и обычно это самая простая целочисленная переменная с именем **i**.
* **Проверочное выражение** — данное выражение оценивается в на- чале каждой итерации цикла на предмет логического значения **true**. Когда оценка возвращает значение **true**, итерация продолжается, а при возвращении значения **false** цикл немедленно прекращает свою работу, не завершая текущую итерацию.
* **Итерация** — изменяет текущее значение переменной-счетчика, храня в себе общее число итераций, сделанных циклом. Обычно здесь используется выражение **i++** для увеличения либо **i--** для уменьшения значения счетчика.



**На заметку**

Исполняемый на каждой итерации цикла код может быть как одиноч- ным оператором, так и блоком операторов и даже вложенным циклом.

Каждый цикл в какой-то определенной точке должен привести зна- чение проверочного выражения в значение **false**, иначе будет создан так называемый бесконечный цикл. В общем случае проверочное вы- ражение обычно оценивает текущее значение переменной-счетчика для выполнения определенного количества итераций. Например, если проинициализировать счетчик **i** значением 1 и увеличивать его на еди- ницу с каждой итерацией, то после 10 итераций значение выражения **i < 11** станет ложным. Таким образом, цикл, исполнившись 10 раз, завершится.

1. Создайте новую программу с именем **For**, содержащую стандарт- ный метод **main**.

### class For

### {

### public static void main ( String[] args ) { }

### }

1. Внутри главного метода объявите и проинициализируйте целочис- ленную переменную для подсчета общего числа итераций.

### int num = 0 ;

1. Добавьте цикл **for**, выполняющий три итерации и отображающий текущее значение своего счетчика-переменной **i** на каждой ите- рации.

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

### {



**JAVA**

*For.java*

### System.out.println( "Внешний цикл i=" + i ) ;

### }

1. Внутри блока цикла **for** добавьте вложенный цикл **for**, который также выполняет три итерации, отображая текущее значение счетчика-переменной **j** и общее число итераций.

**53**

### for ( int j = 1 ; j < 4 ; j++ )

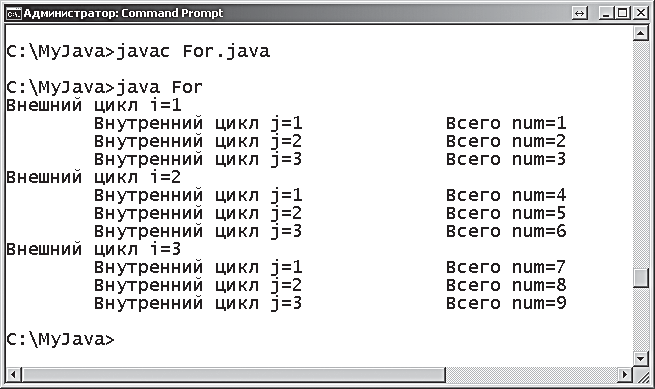
### {

### System.out.print( "\tВнутренний цикл j=" + j ) ; System.out.println( "\t\tВсего num="+ (++num) ) ;

### }



**Внимание**

1. Сохраните программу под именем *For.java*, затем скомпилируйте и запустите, чтобы увидеть вывод.

Операторы инкремента

и декремента могут быть как префиксными, так и пост- фиксными. В первом случае они вначале увеличивают значение переменной, а за- тем происходит обращение к ней, во втором случае — наоборот.

**Цикл while**

**3. Создание операторов**

Альтернативной структурой для цикла **for** является структура цикла, использующая ключевое слово **while** и имеющая следующий синтак- сис:

**while ( *проверочное-выражение* )**

### {

#### *операторы-для-выполнения-на-каждой-итерации* ;

### }

Бесконечный цикл заблоки- рует программу, поскольку итерации продолжают выполняться — в системе Windows, чтобы выйти из этой ситуации, нажмите со- четание клавиш **Ctrl+C**.

Подобно циклу **for** цикл **while** тоже с периодичностью выполняет со- держащиеся в нем операторы до тех пор, пока проверочное условие не будет иметь значение **true**. В этом случае цикл завершает свою работу, и программа переходит к следующей задаче.

В отличие от цикла **for**, в скобках после ключевого слова **while** не содержится ни инициализатора, ни модификатора, изменяющего переменную-счетчик. Это означает, что в проверочном выражении должна находиться какая-то величина, которая будет изменяться по мере выполнения итераций цикла, иначе будет создан бесконечный цикл, выполняющий свои операторы.

Проверочное выражение оценивается в начале каждой итерации цикла на предмет логического значения **true**. Если результат проверки воз- вращает **true**, итерация продолжается, в противном случае цикл не- медленно завершается, не выполняя итерацию.

Обратите внимание, что если проверочное выражение возвращает зна- чение **false** при самой первой оценке, то операторы цикла никогда не исполнятся.



**Внимание**

При помощи цикла **while** можно смоделировать структуру цикла **for**, при этом в проверочном выражении будет оцениваться переменная- счетчик, которая станет создаваться вне цикла, а изменять свое значе- ние внутри цикла на каждой его итерации. Например, внешний цикл **for** из предыдущего примера можно воссоздать при помощи цикла **while** следующим образом:

### int i = 1 ; while ( i < 4 )

### {

### System.out.println( "Внешний цикл i=" +i ) ; i++ ;

**54**

### }

Таким образом, мы переместили инициализатор снаружи цикла перед структурой **while**, а модификатор — внутри блока операторов самого цикла.

1. Создайте новую программу с именем **While**, содержащую стандарт- ный метод **main**.

### class While

### {

### public static void main ( String[] args ) { }

### }

1. Внутри главного метода объявите и проинициализируйте целочис- ленную переменную с именем **num**.

### int num = 100 ;

1. Добавьте цикл **while**, который будет отображать текущее значение переменной **num**, пока оно остается больше нуля.

### while ( num > 0 )

### {



**JAVA**

*While.java*

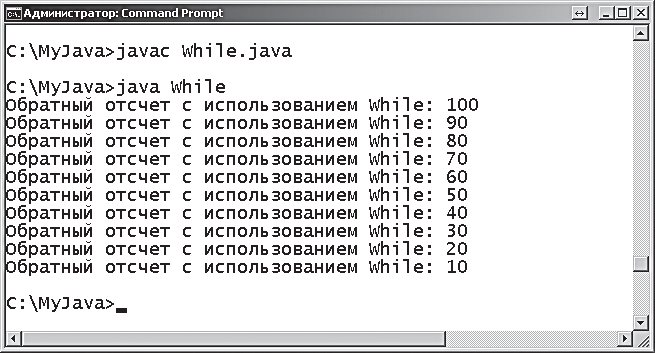
### System.out.println( "Обратный отсчет с использованием While: " + num ) ;

### }

**55**

1. В конец блока **while** добавьте модификатор, чтобы уменьшать зна- чение переменной **num** на 10 на каждой итерации, тем самым избе- гая бесконечного цикла.

### num -= 10 ;

1. Сохраните программу под именем *While.java*, затем скомпилируйте и запустите.



**На заметку**

Присваивание в данном модификаторе — это со- кращенная запись для **num = (num - 10)**.

**Циклы do-while**

**3. Создание операторов**

Одной из вариаций структуры цикла **while**, описанной на предыдущей странице, является цикл, использующий ключевое слово **do** и имею- щий следующий синтаксис:

### do

### {

#### *операторы-для-выполнения-на-каждой-итерации* ;

### }

**56**

Операторы, исполняемые в цикле, всегда заключайте в скобки — это внесет яс- ность и улучшит обслужи- ваемость кода.

**while ( *проверочное-выражение* ) ;**

Подобно циклам **for** и **while**, цикл **do-while** с периодичностью выпол- няет содержащиеся в нем операторы до тех пор, пока проверочное вы- ражение не примет значение **true** — затем цикл завершает свою работу и программа переходит к следующей задаче.

В отличие от циклов **for** и **while**, в цикле **do-while** проверочное выра- жение стоит после блока, содержащего выполняющиеся в цикле опера- торы. Проверочное выражение оценивается в конце каждой итерации цикла на предмет логического значения. Когда проверочное выраже- ние в результате возвращает значение **true**, продолжается следующая итерация, в противном случае цикл немедленно завершается. Это озна- чает, что операторы внутри цикла **do-while** исполнятся, по крайней мере, один раз. Обратите внимание, что если проверочное выражение возвратит значение **false** при первой его оценке, то операторы цикла при этом уже один раз выполнены.

С помощью цикла **do-while** можно также смоделировать структуру цикла **for**, оценивая переменную-счетчик в его проверочном выраже- нии, при этом инициализатор будет находиться вне цикла, а модифика- тор — внутри блока операторов, точно так же, как в цикле **while**.

Для всех видов циклов **for**, **while** и **do-while**, если они содержат только один оператор для исполнения, можно опускать фигурные скобки во- круг этого оператора. Но если вы захотите добавить дополнительные операторы, то обойтись без скобок не получится.



Какой из циклов — **for**, **while** или **do-while** — выбрать при програм- мировании, зависит от персональных предпочтений и от того, для чего используется цикл. Структура цикла **for** удобно размещает ини- циализатор, проверочное выражение и модификатор (после ключевого слова **for**). Структура цикла **while** может быть более лаконичной, но вы не должны забывать включить модификатор в операторы цикла во избежание его зацикливания. Цикл **do-while** просто добавляет воз- можность исполнения своего блока операторов хотя бы один раз перед

**Совет**

оценкой проверочного выражения — как демонстрируется в примере ниже.

1. Создайте новую программу с именем **DoWhile**, содержащую стан- дартный метод **main**.

### class DoWhile

### {

### public static void main ( String[] args ) { }

### }

1. Внутри главного метода добавьте объявление и инициализацию целочисленной переменной с именем **num**.

### int num = 100 ;

1. Добавьте цикл **do-while** для вывода текущего значения перемен- ной **num**, пока оно меньше нуля.

### do

### {



**JAVA**

*DoWhile.java*

### System.out.println( "Используем DoWhile: " + num ) ;

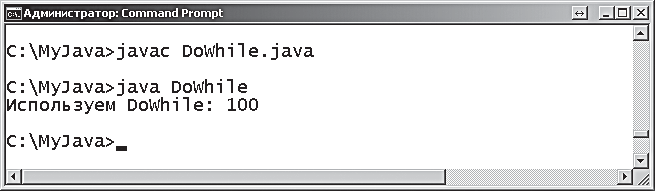
### }

**57**

### while ( num < 0 )

1. Добавьте в конец блока **do-while** модификатор для изменения зна- чения переменной **num** на каждой итерации, тем самым избежав бесконечного цикла.

### num += 10 ;

1. Сохраните программу под именем *DoWhile.java*, затем скомпили- руйте и запустите — вы увидите, что проверочное условие не вы- полняется, но оператор в цикле выполняется один раз.



**На заметку**

Присваивание в этом моди- фикаторе является сокра- щенной записью для **num = (num + 10)**.

**Выход из циклов**

**3. Создание операторов**

Ключевое слово **break** может применяться для преждевременного пре- рывания цикла при определенном условии. Оператор **break** располага- ется внутри блока операторов цикла и предваряется проверочным вы- ражением. Когда проверка возвращает значение **true**, цикл немедленно завершается и программа переходит к следующей задаче. Например, в случае вложенного цикла происходит переход к следующей итерации внешнего цикла.



**JAVA**

*Break.java*

1. Создайте новую программу с именем **Break**, содержащую стандарт- ный метод **main**.

### class Break

### {

### public static void main ( String[] args ) { }

### }

1. Внутри главного метода создайте два вложенных цикла **for**. Эти циклы будут отображать значение своих счетчиков на каждой из трех итераций.

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

### {

**58**

### for ( int j = 1 ; j < 4 ; j++ )

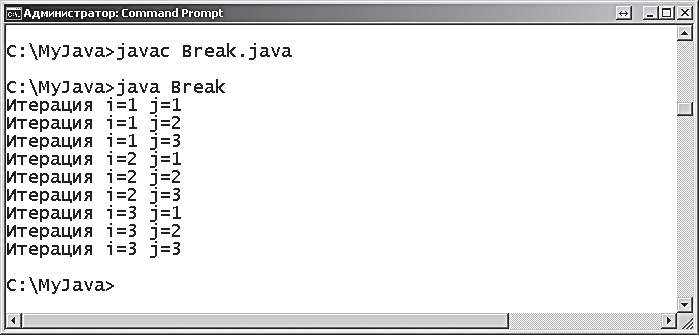
### {

### System.out.println( "Итерация i="+i+" j="+j ) ;

### }

### }

1. Сохраните программу под именем *Break.java*, затем скомпилируйте и запустите, чтобы увидеть вывод.



Здесь выполняются три итерации внешнего цикла, и на каждой ис- полняется внутренний цикл. Чтобы остановить второе исполнение внутреннего цикла, можно добавить оператор **break**.

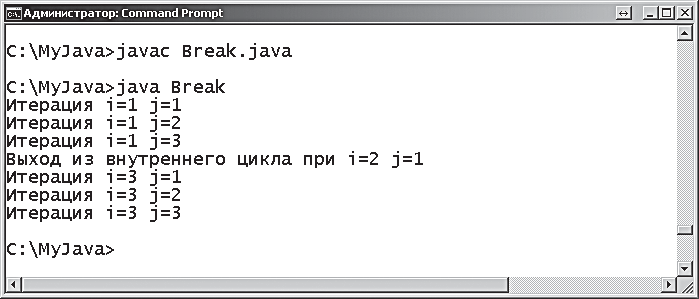
1. Добавьте оператор **break** в начало блока операторов внутреннего цикла, чтобы выполнить выход из этого цикла, затем перекомпи- лируйте и запустите заново программу.

### if ( i == 2 && j == 1 )

### {

### System.out.println( "Выход из внутреннего цикла при i=" +i+ " j=" +j ) ; break ;

### }



Существует возможность пропустить одну итерацию цикла при определенном условии — для этого используется ключевое слово **continue**. Оператор **continue** располагается внутри блока операто- ров цикла и также предваряется проверочным выражением. Если проверка выдает значение **true**, то данная итерация завершается.

1. Добавьте оператор **continue** в начало блока операторов внутренне- го цикла, чтобы пропустить первую итерацию внутреннего цикла, затем перекомпилируйте и запустите заново программу.

### if ( i == 1 && j == 1 )

### {

В данном случае оператор **break** пропускает все три итерации внутреннего цикла на второй итерации внешне- го цикла.



**На заметку**

### System.out.println( "Продолжение работы внутреннего цикла при i=" +i+ " j=" +j ) ; continue;

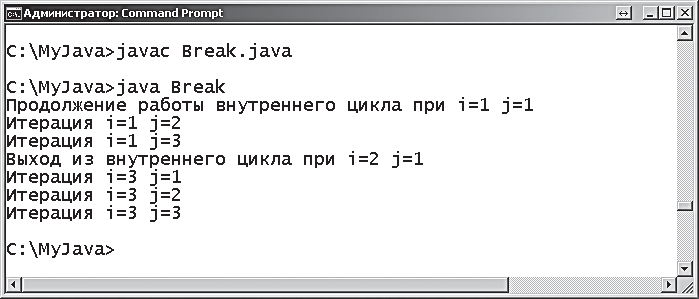
**59**

### }



**На заметку**

В данном случае оператор

**continue** позволяет пропу- стить первую итерацию вну- треннего цикла на первой итерации внешнего цикла.

**Возврат управления**

**3. Создание операторов**

Стандартное поведение операторов **break** и **continue** может быть изме- нено, если указать явно, что управление следует передать помеченному оператору цикла, указав имя его метки.

*Label.java*

1. Создайте новую программу с именем **Label**, содержащую стандарт- ный метод **main**.

### class Label



**JAVA**

### {

### public static void main ( String[] args ) { }

### }

1. Внутри главного метода создайте два вложенных цикла **for**, которые выводят значения своих счетчиков на каждой из трех итераций.

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

### {

### for ( int j = 1 ; j < 4 ; j++ )

### {

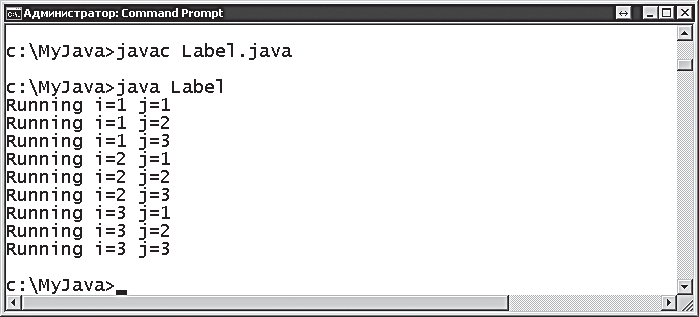
**60**

### System.out.println( "Итерация i="+i+ " j="+j ) ;

### }

### }

1. Сохраните программу под именем **Label.java**, затем скомпилируйте и запустите.



Для того чтобы пометить цикл, перед ним ставится имя метки и символ двоеточия.

1. Отредактируйте начало внешнего цикла, пометив его меткой

### outerLoop.

### outerLoop : for ( int i = 1 ; i < 4 ; i++ )

Чтобы явно указать программе, что нужно перейти к этому внеш- нему циклу, поставьте имя данной метки после ключевого слова **continue**.

1. Добавьте оператор **continue** в начало блока операторов внутренне- го цикла, чтобы перейти к следующей итерации внешнего цикла, затем перекомпилируйте и запустите заново программу.

### if ( i == 1 && j == 1 )

### {

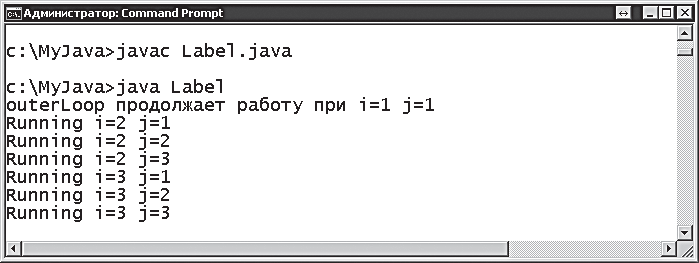
### System.out.println( "outerLoop продолжает работу при i=" +i+ " j=" +j ) ; continue outerLoop ;



**На заметку**

В данном случае оператор **continue** пропускает все три итерации первого прохода внутреннего цикла, возвра- щая управление внешнему циклу.

### }



Для явного указания того, что программа должна выйти из внеш- него цикла, введите после ключевого слова **break** имя этой метки.

1. Добавьте оператор **break** в начало блока операторов внутреннего цикла для выхода из внешнего цикла, затем снова скомпилируйте и запустите программу.

### if ( i == 2 && j == 3 )

### {

В данном случае опера- тор **break** пропускает все дальнейшие итерации всей структуры цикла, совершая выход из внешнего цикла.

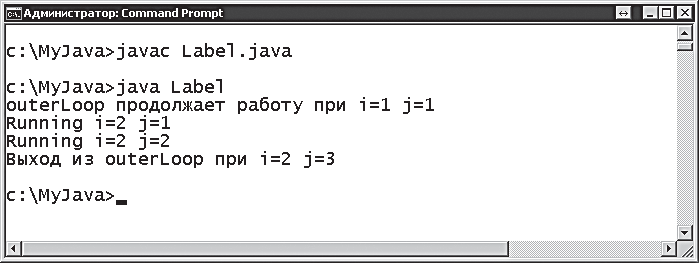


**На заметку**

**61**

### System.out.println( "Выход из outerLoop при i=" +i+ " j=" +j ) ; break outerLoop ;

### }



**Заключение**

**3. Создание операторов**

* + Ключевое слово **if** используется для оценки проверочного выра- жения на предмет логических значений **true** или **false**.
  + Блок операторов **if** может содержать один или более операторов, которые выполняются только тогда, когда проверочное выражение возвращает значение **true**.
  + Ключевое слово **else** определяет альтернативные операторы для выполнения программы в случае, когда проверка, выполняемая ключевым словом **if**, возвращает значение **false**.
  + Комбинация операторов **if else** позволяет программе осущест- влять условное ветвление.
  + Оператор **switch** в большинстве случаев предлагает хорошую аль- тернативу операторам **if else**, предоставляя различные опции для выбора.
  + Каждая опция **case** может быть завершена ключевым словом **break**, и, таким образом, выполняться будут только операторы, связанные с этой опцией.
  + Ключевое слово **default** может определять операторы, исполняе- мые в случае, когда все опции **case** возвращают значение **false**.

**62**

* + Цикл с периодичностью исполняет операторы, содержащиеся в нем, до тех пор, пока проверочное выражение не возвратит зна- чение **false**.
  + После ключевого слова **for** в скобках указываются инициализатор, проверочное выражение и модификатор счетчика.
  + В циклах **while** и **do-while** во избежание входа в бесконечный цикл нужно изменять значение переменной, используемой в провероч- ном выражении.
  + В циклах **for** и **while** проверочное выражение оценивается в самом начале — перед первой итерацией цикла.
  + В циклах **do-while** проверочное выражение оценивается в конце цикла — после первой итерации.
  + Итерацию цикла можно пропустить с помощью ключевого слова

### continue.

* + Используя ключевое слово **break**, можно завершать работу цикла.
  + Вложенные внутренние циклы могут использовать метки вместе с ключевыми словами **break** и **continue**, чтобы передавать управ- ление внешнему циклу.

1. **Работа с данными**

***В данной главе описывается, каким образом работать со значениями данных, используя различные программные конструкции***

***языка Java.***

# Преобразование типов

# Создание массивов переменных

# Передача аргументов

# Передача множественных аргументов

# Обход элементов в цикле

# Изменение значений элемента

# Добавление размеров массива

# Перехват исключений

# Заключение

**64**

**4. Работа с данными**

Все числовые классы имеют метод **parse...** и **toString**, позволяющие преобразовы- вать строковые и числовые значения.

**Преобразование типов**

В программировании на Java при работе со значениями следует вни- мательно учитывать типы данных во избежание ошибок компиляции. Например, если методу, который требует значение типа **int**, передать переменную типа **float**, то это приведет к ошибке компилятора. Это означает, что зачастую для того, чтобы обрабатывать данные, их нужно конвертировать из одного типа в другой.

Числовые значения можно легко преобразовывать (приводить) из одного числового типа данных в другой, используя синтаксис:

#### ( *тип-данных* ) *значение*

Необходимо понимать, что при приведении типов данных может про- исходить некоторая потеря точности. Так случается, например, при преобразовании числа с плавающей точкой (**float**) в целое число (**int**), поскольку значение после десятичной точки будет отбрасываться. На- пример, приведение значения 9.9 к целочисленному типу выдаст в ре- зультате значение 9.

Значения символьного типа данных (**char**) могут быть автоматически использованы в качестве значений целочисленного типа (**int**), потому что каждый из них имеет уникальное целочисленное представление, а именно числовой код ASCII, поддерживаемый компилятором Java. Например, латинская буква A имеет числовой код 65.

Числовые значения могут быть преобразованы к строковому типу (**String**) при помощи метода **toString()**. Он в качестве аргумента в сво- их скобках принимает числовое значение. Например, преобразование целочисленной переменной **num** к строковому типу имеет вид **Integer. toString(num)**. Аналогично преобразование переменной плавающего типа **num** в строковую будет иметь вид **Float.toString(num)**. На самом деле такие преобразования не всегда требуются, поскольку, например, объединение двух переменных в строку, если одна из них имеет стро- ковый тип, происходит автоматически.



**На заметку**

С большей вероятностью вам будет нужно преобразовывать величи- ны строкового типа в числа, чтобы потом использовать их в арифме- тических вычислениях. Значение строкового типа может быть преоб- разовано в целочисленное при помощи метода **Integer.parseInt()**. Данный метод принимает строковую переменную в качестве аргумента в своих скобках. Например, преобразование строковой переменной **msg** в целое число будет иметь вид **Integer.parseInt(msg)**. Аналогично для переменной с плавающей точкой — **Float.parseFloat(msg)**. При преобразовании строковой величины в числовой тип данных строка должна содержать действительное числовое значение, иначе компиля- тор сообщит об ошибке.

1. Создайте новую программу с именем **Convert**, содержащую стан- дартный метод **main**.

### class Convert

### {

### public static void main ( String[] args ) { }

### }

1. Внутри главного метода объявите и проинициализируйте одну строковую переменную и одну переменную с плавающей точкой.

### float daysFloat = 365.25f ; String weeksString = "52" ;

1. Преобразуйте переменную с плавающей точкой в целочисленную.

### int daysInt = (int) daysFloat ;

1. Преобразуйте строковую переменную в целочисленную.

### int weeksInt = Integer.parseInt( weeksString ) ;

1. Произведите арифметические действия над преобразованными значениями и выведите результат.

### int week = ( daysInt / weeksInt ) ; System.out.println( "Дней в неделе: " + week ) ;

1. Сохраните программу под именем **Convert.java**, затем скомпили-

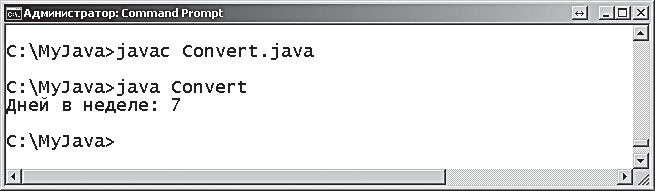
руйте и запустите.



**JAVA**

*Convert.java*

**65**



**66**

**4. Работа с данными**



**Внимание**

Не забывайте, что массив ин- дексируется, начиная с нуля. Это означает, что **index[2]** обращается к третьему элементу массива, а не ко второму.

**Создание массивов переменных**

Массив — это простая переменная, которая может содержать несколь- ко значений, в отличие от обычной переменной, содержащей един- ственное значение.

Объявление массива включает в себя тип данных с использованием клю- чевых слов, обозначающих типы данных, после которого следуют ква- дратные скобки, обозначающие, что это будет массив. Затем в объявле- нии стоит имя массива в соответствии с договоренностями именования.

Массив можно проинициализировать при объявлении, присвоив зна- чение соответствующего типа данных, записанное в фигурные скобки и разделенное запятыми. Например, объявление целочисленного мас- сива, проинициализированного тремя значениями, может выглядеть следующим образом:

### int[ ] numbersArray = { 1, 2, 3 } ;

При этом создается массив длиной, равной количеству элементов, ука- занных в списке. В данном случае массив состоит из трех элементов. Значения элементов сохраняются в массиве под порядковыми номера- ми (индексами), по которым можно обратиться к этим элементам. Для обращения к элементам следует написать имя массива и соответствую- щий индекс элемента в квадратных скобках. Элементы индексируются, начиная с нуля. Например, чтобы обратиться к первому элементу из массива, указанного выше, мы запишем **numbersArray[0]**.

Несмотря на то, что значения, хранящиеся в каждом элементе массива, могут быть изменены так же, как обычная переменная, размер массива определяется при объявлении и не может быть изменен впоследствии. Общее число элементов в массиве можно узнать, если обратиться к свой- ству массива **length**. Для этого нужно записать через точечную запись

«имя массива.length». Например, **numbersArray.length** возвратит раз-

мер массива, приведенного выше, в данном случае целое число 3.

Массивы могут быть объявлены также и без присвоения начальных значений элементам. Это можно сделать при помощи ключевого слова **new**, создавая пустой «объект» массива указанного размера. Число тре- буемых элементов указывается в присваивании внутри квадратных ско- бок после типа данных. Например, объявление пустого целочисленного массива с тремя элементами может выглядеть следующим образом:

### int[ ] numbersArray = new int[3] ;

Во время создания пустого массива элементам присваиваются значе- ния в зависимости от типов данных: для типов **int** и **float** — 0, для

типа **String** — значение **null**, для типа **char** — **\0**, а для типа **boolean** — значение **false**.

**67**

1. Создайте новую программу с именем **Array**, содержащую стандарт- ный метод **main**.

### class Array

### {

### public static void main ( String[] args ) { }

### }

1. Внутри главного метода объявите и проинициализируйте строко- вый массив из трех элементов.

### String[] str = { "Java ", "Хороший", " Язык" } ;

1. Объявите пустой целочисленный массив с тремя элементами.

### int[] num = new int[3] ;

1. Присвойте значения первым двум элементам целочисленного мас- сива.

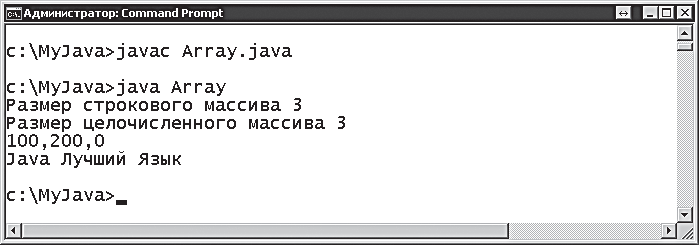
### num[0] = 100 ; num[1] = 200 ;

1. Присвойте новые значения второму элементу строкового массива.

### str[1] = "Лучший" ;

1. Выведите длину каждого массива и содержимое всех элементов. **System.out.println( "Размер строкового массива " + str.length ); System.out.println( "Размер целочисленного массива "+ num.length);**

### System.out.println( num[0] + "," +num[1]+ ","+num[2] ) ; System.out.println( str[0] + str[1] + str[2] ) ;

1. Сохраните программу под именем *Array.java*, затем скомпилируйте и запустите.



**JAVA**

*Array.java*



**На заметку**

Строковые переменные должны быть заключены в кавычки.

**Передача аргументов**

**4. Работа с данными**

Код на Java, объявляющий стандартный метод **main**, включает внутри своих скобок аргумент, который создает строковый массив с именем **args**:

### public static void main( String[] args ) { }

Основное назначение массива **args[]** — передавать значения (аргу- менты) в программу при ее вызове. Значения, которые нужно передать в программу, добавляются в командную строку при вызове программы после ее имени. Например, команда, которая передаст строку **Java** ко- манде с именем **Run**, будет выглядеть так: **Run Java**.

Одно значение, переданное в программу, автоматически помещается в первый элемент массива **args[]**. Тем самым, к нему можно обратить- ся в виде **args[0]**.

Важно знать, что массив **args[]** содержит строковый тип данных, поэтому любое числовое значение, переданное в программу, будет со- храняться как строковое представление этого числа. Это означает, что программа не сможет использовать данное значение в арифметической операции до тех пор, пока его не преобразовать к числовому типу дан- ных, например **int**. Например, **Run 4** передает число 4 в программу, которая сохраняется как строковое значение **"4"**, а не как целочис- ленное **4**. Следовательно, результат выполнения выражения **args[0]+3** выдаст объединенную строку "43", а не сумму 7. Но с помощью пре- образования, используя метод **Integer.parseInt()**, можно получить требуемый результат, если вам нужна сумма.

**68**

Строку, содержащую пробелы, можно передавать в программу как одно строковое значение, заключив всю эту строку в двойные кавычки. Например, **Run "Java In Easy Steps"**.

Передача аргументов в программу может, например, использоваться для того, чтобы указывать опции запуска конкретной программы. Оп- ция передается в программу в виде строкового значения **args[0]** и мо- жет быть впоследствии оценена с помощью метода **String.equals()**. Например, **args[0].equals("b")** сравнивает аргумент со строковым значением **"b"**.

1. Создайте новую программу с именем **Option**, содержащую стан- дартный метод **main**.

### class Option

### {

### public static void main ( String[] args ) { }

### }

1. Внутри главного метода напишите оператор **if**, в котором произ- ведите поиск аргумента с именем **"-en"**.

### if ( args[0].equals( "-en" ) )

### {



**JAVA**

*Option.java*

### System.out.println( "Опция для Английского языка" ) ;

### }

1. Добавьте альтернативную ветку **else** к оператору **if** для поиска ар- гумента **"-es"**.

### else if ( args[0].equals( "-es" ) )

### {

### System.out.println( "Опция для Испанского языка" ) ;

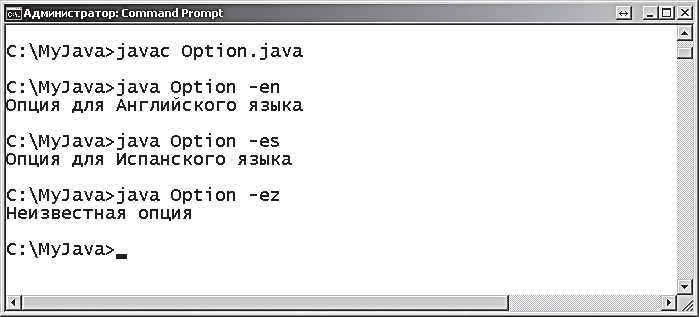
**69**

### }

1. Добавьте еще одну альтернативную ветку **else** для вывода ответа по умолчанию.

### else System.out.println( "Неизвестная опция" ) ;

1. Сохраните программу под именем *Option.java*, затем скомпилируй- те и запустите.



**Передача множественных аргументов**

**4. Работа с данными**

С помощью командной строки в программу можно передать несколько аргументов, которые следуют после имени программы и пробела. Ар- гументы должны отделяться, по крайней мере, одним пробелом, а их значения помещаются в элементы массива **args[]**. После этого к каж- дому из значений можно обращаться по индексу, равно как и к любому другому элементу массива, — **args[0]** для первого аргумента, **args[1]** для второго аргумента и так далее.

Чтобы убедиться, что пользователь ввел необходимое количество аргу- ментов, программе нужно проверить свойство **length** массива **args[]**. Если проверка не дала необходимого результата, можно использовать ключевое слово **return** для выхода из главного метода, тем самым про- изведя выход из программы.

*Args.java*

**70**

1. Создайте новую программу с именем **Args**, содержащую стандарт- ный метод **main**.

### class Args



**JAVA**

### {

### public static void main ( String[] args ) { }

### }

1. Внутри главного метода введите условный оператор **if** для провер- ки количества введенных аргументов и вывода соответствующего сообщения, а также выхода из программы в случае, если количе- ство аргументов не соответствует требуемому — в данном случае трем.

### if ( args.length != 3 )

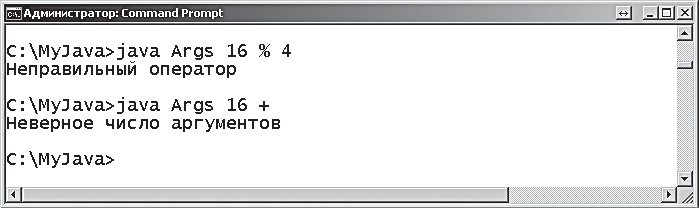
### {

### System.out.println( "Неверное число аргументов" ) ; return ;

### }

1. После оператора **if** создайте две целочисленные переменные и про- инициализируйте их значениями первого и третьего аргументов соответственно.

### int num1 = Integer.parseInt( args[0] ) ; int num2 = Integer.parseInt( args[2] ) ;

1. Добавьте строковую переменную, проинициализировав ее значе- нием объединения всех трех аргументов.

**71**

### String msg = args[0] + args[1] + args[2] + "=" ;

1. Добавьте оператор **if else**, выполняющий арифметические дей- ствия над аргументами и объединяющий результат в переменной строкового типа.

### if ( args[1].equals("+") ) msg += (num1 + num2);

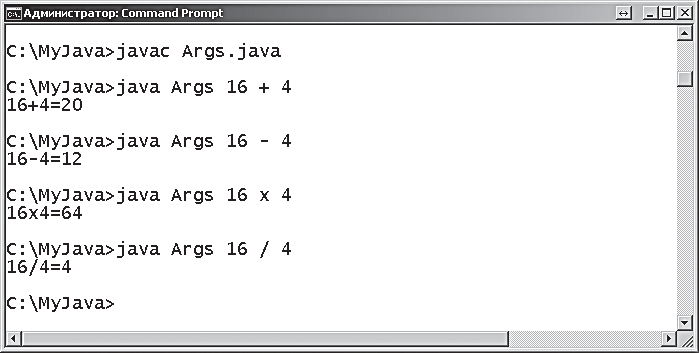
### else if ( args[1].equals("-") ) msg += (num1 - num2); else if ( args[1].equals("x") ) msg += (num1 \* num2); else if ( args[1].equals("/") ) msg += (num1 / num2); else msg = "Неправильный оператор" ;

1. Добавьте следующую строку в конец главного метода для вывода

получившейся строки.

### System.out.println( msg ) ;

1. Сохраните программу под именем *Args.java*, затем скомпилируйте и запустите с тремя аргументами — целым числом, любым сим- волом арифметической операции (**+**, **-**, **х** и **/**) и еще одним целым числом.



1. Перезапустите программу с некорректным вторым аргументом и с неверным числом аргументов.



**Совет**

Ключевое слово **return** ис- пользуется для выхода из текущего метода. Оно также может возвращать значение в точку, где метод был вы- зван (см. главу 6).



**Совет**

Программа сообщит об ошибке, если введены не числовые значения. Подроб- нее об обработке ошибок см. в конце этой главы.

**Обход элементов в цикле**

**4. Работа с данными**

Для того чтобы прочитать все значения, хранящиеся в элементах мас- сива, можно использовать любые типы циклов. Счетчик цикла должен начинаться с индекса первого элемента массива и возрастать до индекса последнего элемента. Заметьте, что индекс последнего элемента в мас- сиве всегда будет на единицу меньше, чем длина массива, поскольку индексы начинаются с нуля.

Весьма полезным считается использованием свойства массива **length** (длина) для условной проверки, определяющей, когда цикл должен за- вершить свою работу. Это значит, что цикл будет продолжаться до тех пор, пока счетчик не превысит значение индекса последнего элемента.



**JAVA**

*Loops.java*

1. Создайте новую программу с именем **Loops**, содержащую стандарт- ный метод **main**.

### class Loops

### {

### public static void main ( String[] args ) { }

### }

1. Внутри главного метода напишите условный оператор, выполняю- щий проверку того, введены ли какие-либо аргументы из команд- ной строки в массив **args[]**.

**72**

### if ( args.length > 0 ) { }

1. Добавьте внутри фигурных скобок условного оператора цикл **for**

для вывода значений, хранящихся в каждом элементе.

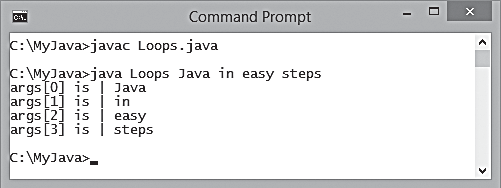
### for ( int i = 0 ; i < args.length ; i++ )

### {

### System.out.println( "args[" +i+ "] - | "+ args[i] ) ;

### }

1. Сохраните программу под именем **Loops.java**, затем скомпилируй- те и запустите с аргументами **Java In Easy Steps**.



1. Отредактируйте файл *Loops.java*, добавив строковый массив, а также цикл **while** для вывода значений, хранящихся в каждом элементе.

### String[] htm = { "HTML5", "in", "easy", "steps" } ; int j = 0 ;

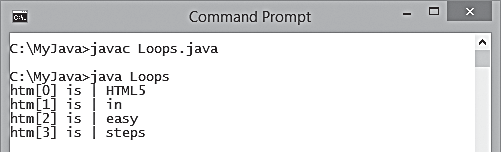
### while ( j < htm.length )

### {

### System.out.println( "htm[" +j+ "] - | " + htm[j] ) ; j++ ;

### }

1. Сохраните изменения, затем заново скомпилируйте и запустите программу.



1. Отредактируйте файл *Loops.java*, добавив еще один строковый массив, а также цикл **do while** для вывода значений, хранящихся в каждом элементе.

### String[] xml = { "XML", "in", "easy", "steps" } ; int k = 0 ;

**73**

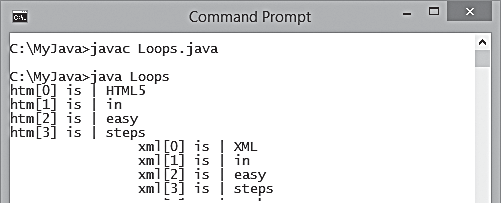
### if ( xml.length > 0 ) do

### {

### System.out.println( "\t\txml["+k+"] - | "+xml[k] ) ; k++ ;

### } while ( k < xml.length ) ;

1. Сохраните изменения, затем заново скомпилируйте и запустите программу.





**Совет**

Обратите внимание, что оператор **do** предваряется условной проверкой, чтобы перед тем, как попытаться вывести значение первого элемента, убедиться, что массив не пустой.

**Изменение значений элемента**

**4. Работа с данными**

Значение любого элемента массива можно изменить, присвоив новое значение конкретному элементу, используя его индекс. К тому же, для того чтобы эффективно разместить элементы, хранящиеся в другом массиве, можно использовать любой тип цикла. Особенно это полезно при комбинировании данных из множественных массивов в один мас- сив объединенных данных.



**JAVA**

*Elements.java*

1. Создайте новую программу с именем **Elements**, содержащую стан- дартный метод **main**.

### class Elements

### {

### public static void main ( String[] args ) { }

### }

1. Добавьте в главный метод проинициализированный целочислен- ный массив, представляющий собой ежемесячные объемы продаж киоска по кварталам года.

**74**

### int[] kiosk\_q1 = { 42000 , 48000 , 50000 } ; int[] kiosk\_q2 = { 52000 , 58000 , 60000 } ; int[] kiosk\_q3 = { 46000 , 49000 , 58000 } ; int[] kiosk\_q4 = { 50000 , 51000 , 61000 } ;

1. Добавьте проинициализированные целочисленные массивы, пред-

ставляющие собой ежемесячные продажи за четыре квартала года.

### int[] outlet\_q1 = { 57000 , 63000 , 60000 } ; int[] outlet\_q2 = { 70000 , 67000 , 73000 } ; int[] outlet\_q3 = { 67000 , 65000 , 62000 } ; int[] outlet\_q4 = { 72000 , 69000 , 75000 } ;

1. Теперь создайте пустой целочисленный массив из 12 элементов,

в котором будут объединены все ежемесячные продажи, а также создайте целочисленную переменную, в которой будет записана общая величина объема продаж.

### int[] sum = new int[ 12 ] ; int total = 0 ;

1. Добавьте цикл **for**, для того чтобы занести в каждый элемент пу- стого массива объединенное значение из других массивов.

### for ( int i = 0 ; i < kiosk\_q1.length ; i++ )

### {

### sum[ i ] = kiosk\_q1[i] + outlet\_q1[i] ; sum[i+3] = kiosk\_q2[i] + outlet\_q2[i] ; sum[i+6] = kiosk\_q3[i] + outlet\_q3[i] ; sum[i+9] = kiosk\_q4[i] + outlet\_q4[i] ;

### }

1. Теперь добавьте еще один цикл **for** для вывода каждого из объеди- ненных ежемесячных объемов продаж, а также вычисления общей суммы.

### for ( int i = 0 ; i < sum.length ; i++ )

### {

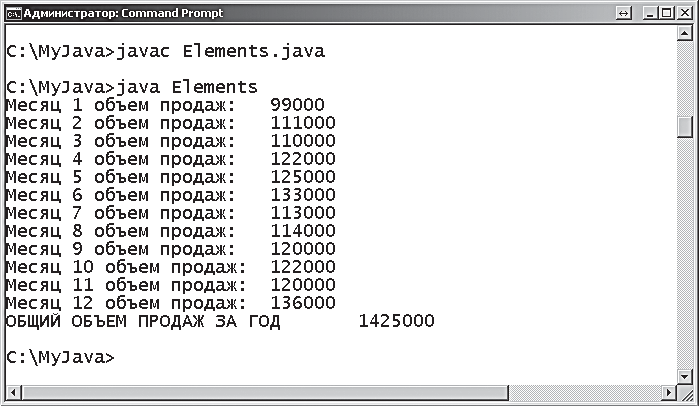
### System.out.println( "Месяц "+ ( i+1 ) + " объем продаж:\t" + sum[i] ) ; total += sum[i] ;

### }

**75**

1. Добавьте последний оператор в конец главного метода для вывода общей суммы (годового объема).

### System.out.println( "ОБЩИЙ ОБЪЕМ ПРОДАЖ ЗА ГОД\t" + total );

1. Сохраните программу под именем *Elements.java*, затем скомпили- руйте и запустите.



**На заметку**

Счетчик увеличивается на единицу для того, чтобы произвести числа месяца от 1 до 12.

Старайтесь избегать размер- ности больше, чем три, — это может вас запутать.



**Внимание**

**4. Работа с данными**

**Добавление размеров массива**

В массивах можно хранить множественные наборы элементов, при- чем каждый из таких наборов станет иметь собственную размерность. К отдельным значениям можно будет обращаться с помощью много- мерного массива, используя соответствующие индексы для каждой размерности, например **num [1] [3]**.

Например, чтобы организовать бизнес-ежедневник с записями для каж- дого дня, потребуется массив из 52 элементов (один на неделю), каждый из которых содержит еще один массив из семи элементов (для каждого дня). Объявление такого массива будет выглядеть следующим образом:

### int[][] dailyRecord = new int [52] [7] ;

Такой массив массивов обеспечивает элемент для каждого рабочего дня. Значения для каждого дня заносятся в многомерный массив с ука- занием соответствующего индекса для каждого размера. Например, чтобы задать значение для первого дня шестой недели, мы запишем:

### dailyRecord [5] [0] = 5000 ;

Каждый массив содержит собственное свойство длины (**length**), к кото- рому можно обратиться, указав требуемую размерность. Например, для вышеуказанного массива выражение **dailyRecord.length** возвратит значение 52 — размер первого измерения. Чтобы найти размер второго измерения, выражение **dailyRecord[0].length** возвратит значение 7.

Двухмерные массивы часто используются для хранения координат, где первая размерность представляет координату по оси X, а вторая коор- дината — по оси Y, например **point[3][5]**.

Трехмерные массивы могут использоваться для хранения коорди- нат XYZ аналогичным способом, но, конечно, запись **point[4][8][2]** сложнее для восприятия.

При организации многомерных массивов очень удобно пользоваться вложенными циклами — каждый уровень цикла может обращаться к элементам соответствующей размерности.

**76**



**JAVA**

*Dimensions.java*

1. Создайте новую программу с именем **Dimensions**, содержащую стандартный метод **main**.

### class Dimensions

### {

### public static void main ( String[] args ) { }

### }

1. В главном методе создайте двухмерный массив для хранения коор- динаты XY.

### boolean[][] points = new boolean[5][20] ;

1. Определите одну точку Y на каждой оси X.

### points[0][5] = true ; points[1][6] = true ; points[2][7] = true ; points[3][8] = true ; points[4][9] = true ;

1. Для прохода по первому индексу массива добавьте цикл **for**, встав-

ляя на каждой итерации символ новой строки.

### for ( int i = 0 ; i < points.length ; i++ )

### {

### System.out.print( "\n" ) ;



**На заметку**

Логические переменные имеют по умолчанию значе- ние **false**.

### }

1. Внутри фигурных скобок цикла **for** добавьте еще один цикл **for**

для итерации по второму индексу массива.

**77**

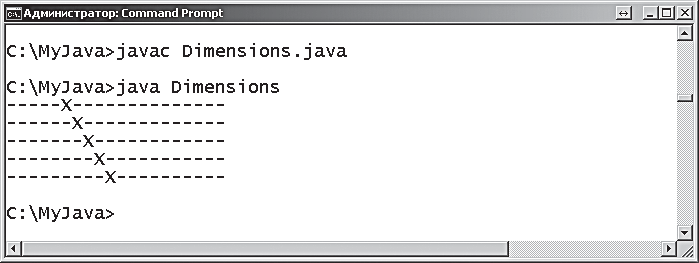
### for ( int j = 0 ; j < points[0].length ; j++ ) { }

1. Внутри фигурных скобок второго цикла **for** добавьте выраже- ние для вывода символов в соответствии с логическим значением элемента.

### char mark = ( points[i][j] ) ? 'X' : '-' ; System.out.print( mark ) ;

1. Сохраните программу под именем *Dimensions.java*, затем скомпи-

лируйте и запустите.



**Перехват исключений**

**4. Работа с данными**

В программах на этапе выполнения могут возникать проблемы, при- водящие к ошибкам (исключениям), которые, в свою очередь, вызы- вают остановку выполнения программы. Очень часто причиной таких исключений является некорректный пользовательский ввод. Следова- тельно, хорошо продуманная программа должна попытаться учесть все возможные варианты, при которых пользователь может вызвать исключение на этапе выполнения.

Части кода, в которых возможно возникновение исключений, заключа- ются в специальные блоки операторов **try catch**. Такие блоки позво- ляют программе обрабатывать исключения, не приводя к остановкам, а выглядят они следующим образом:

### try

### {

#### *операторы, где может возникнуть исключение*

### }

### catch( Exception e )

### {

**78**

#### *операторы, обрабатывающие исключение*

### }

вращает дополнительную информацию о перехвачен- ных исключениях.

В скобках, следующих после ключевого слова **catch**, определяется класс исключения, которое нужно перехватить. Класс **Exception** является исключением верхнего уровня, он перехватывает все исключения. Ис- пользуя многочисленные операторы **catch**, можно перехватывать раз- личные классы исключений более низкого уровня.

Наиболее распространенные исключения — это **NumberFormat- Exception**, которое возникает, когда программа находит значение не- корректного типа, а также **ArrayIndexOutOfBoundsException**, которое возникает при попытке обращения к элементу массива с индексом, выходящим за пределы разрешенного диапазона. Для каждого из этих исключений полезно создавать отдельный ответ для объяснения поль- зователю природы возникшей ошибки.



Метод **e.getMessage()** воз-

**Совет**

Блок операторов **try catch** может быть расширен дополнительным необязательным блоком **finally**, содержащий код, который будет вы- полняться в любом случае, независимо от того, возникло исключение в программе или нет.

1. Создайте новую программу с именем **Exceptions**, содержащую стандартный метод **main**.

### class Exceptions

### {

### public static void main ( String[] args ) { }

### }

1. Внутри главного метода напишите оператор **try**, в котором будет выводиться единственный целочисленный аргумент.

### try

### {



**JAVA**

*Exceptions.java*

### int num = Integer.parseInt( args[0] ) ; System.out.println( "Вы ввели: "+ num ) ;

### }

1. Добавьте оператор **catch**, чтобы обработать исключение, кото- рое возникнет в том случае, если программа запускается без ар- гумента.

### catch( ArrayIndexOutOfBoundsException e )

**79**

### { System.out.println( "Требуется целочисленный аргумент." ) ; }

1. Добавьте оператор **catch** для обработки исключения, которое воз- никнет в том случае, когда программа запускается с нечисловым аргументом.

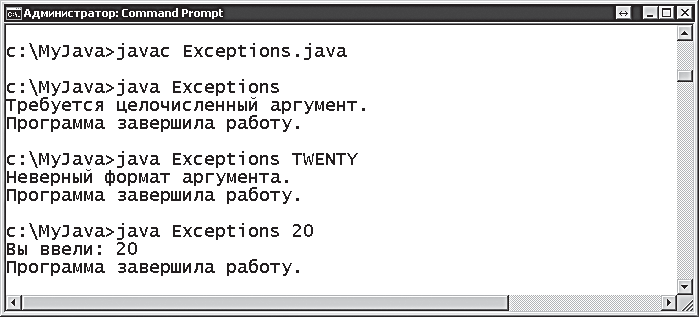
### catch( NumberFormatException e )

### { System.out.println( "Неверный формат аргумента." ) ; }

1. Добавьте оператор **finally** в конец программы.

### finally { System.out.println( "Программа завершила работу.");}

1. Сохраните программу под именем *Exceptions.java*, затем скомпили- руйте и запустите, пытаясь вызвать исключения.



**Заключение**

**4. Работа с данными**

* + Любые числовые значения можно конвертировать в другие типы числовых данных с помощью приведения типов, а также к строко- вому типу, используя метод **toString()**.
  + Значение строкового типа может быть сконвертировано в цело- численное (**int**) с помощью метода **Integer.parseInt()**, а так- же в значение плавающего типа (**float**), используя метод **Float. parseFloat()**.
  + Массив — это переменная, которая может содержать множествен- ные значения, которые инициализируются списком внутри фигур- ных скобок.
  + Пустой массив может быть создан при помощи ключевого сло- ва **new**.
  + Свойство массива **length** содержит целочисленную величину, рав- ную числу элементов этого массива.
  + К любому элементу массива можно обращаться по его индексу.
  + Главный метод программы создает строковый массив с именем

**args**, в котором сохраняются аргументы командной строки.

**80**

* + Первый аргумент командной строки автоматически сохраняется в элементе **args[0]**.
  + Множественные аргументы, передаваемые в программу из команд- ной строки, должны быть разделены пробелом.
  + Циклы являются идеальным способом для того, чтобы прочитать значение, хранящееся в элементах массива.
  + Данные из нескольких массивов можно комбинировать и форми- ровать из них новый массив объединенных данных.
  + Многомерные массивы могут содержать множественные наборы значений элементов, каждый из которых имеет свою размерность.
  + Блок операторов **try catch** используется для обработки исключе- ний, возникающих на этапе исполнения.
  + Класс **Exception** перехватывает все исключения, включая

### NumberFormatException и ArrayIndexOutOfBoundsException.

* + Блок **try catch** может быть расширен оператором **finally**, содер- жащим код, который будет выполняться в любом случае.

1. **Работа с данными**

***В этой главе демонстрируется, как управлять программными данными, используя различные методы из библиотеки языка Java.***

# Изучение классов Java

# Математические вычисления

# Округление чисел

# Генерация случайных чисел

# Управление строками

# Сравнение строк

# Поиск строк

# Обработка символов

# Заключение

**5. Работа с данными**

Вы можете щелкнуть мышью по ссылке **Frames**, чтобы просматривать докумен- тацию в мультиоконном режиме.

**82**

**Изучение классов Java**

Язык Java содержит обширную библиотеку протестированного кода, который размещен в так называемых пакетах («packages»). Основным пакетом языка Java является пакет **java.lang**, который по умолчанию имеет доступ к прикладному программному интерфейсу Java (Java API). Это означает, что при создании программ всегда доступны свой- ства и методы, предоставляемые пакетом **java.lang**. Например, функ- циональность стандартного вывода, предоставляемая методом **System. out.println()**, на самом деле вызывает метод класса **System**, который является частью пакета **java.lang**.

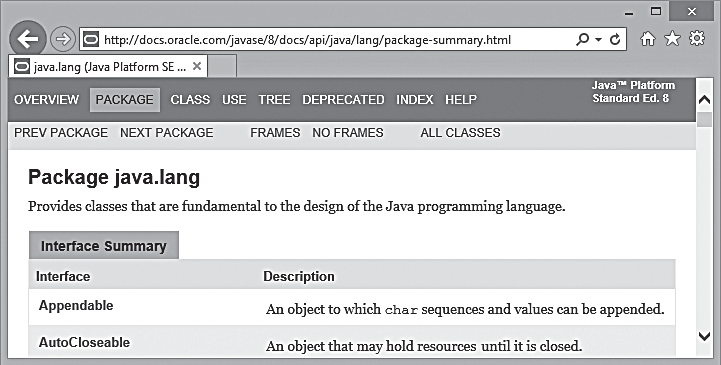
Содержимое пакета организовано в иерархическом порядке, который позволяет обращаться к любому его элементу, используя точечную за- пись. Например, класс **System** содержит свойство **out** (поле), которое, в свою очередь, включает метод **println()** — к нему можно обратиться как **System.out.println()**.

Для того чтобы изучить подробнее классы в Java, можно обратиться к документации, которая предоставляет информацию о любом до- ступном элементе. Документация доступна во Всемирной паутине по адресу **docs.oracle.com/javase/8/docs/api**. Также вы можете ее скачать для офлайн-изучения. Объем документации достаточно большой, но познакомиться с ней все равно стоит. Для начала полезно ознакомить- ся со страницей API Overview (Обзор API-функций), которая содержит список всех пакетов с их кратким описанием.

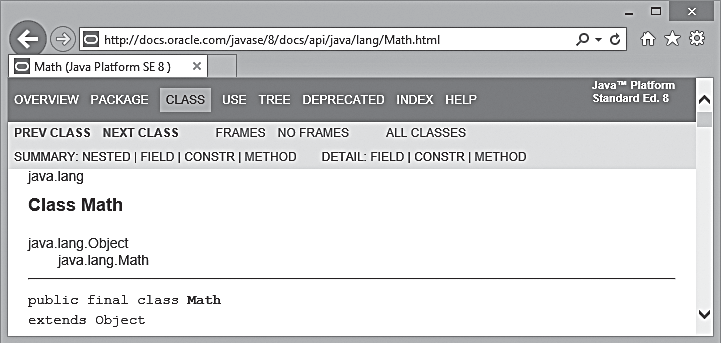


**На заметку**

1. Запустите веб-браузер и откройте страницу обзора API-функций по адресу **docs.oracle.com/javase/8/docs/api**.
2. Просмотрите перечисленные в алфавитном порядке пакеты на вкладке **Overview**, затем пролистайте вниз, найдите пакет **java. lang** и щелкните по его ссылке.



1. Просмотрите перечисленные в алфавитном порядке классы в раз- деле **Class Summary** на соответствующей странице, прокрутите страницу вниз и, найдя класс **Math**, щелкните по его гиперссылке.



1. Просмотрите все методы данного класса, перечисленные в алфа- витном порядке в разделе **Method Summary**, затем щелкните на любой из гиперссылок, чтобы ознакомиться с синтаксисом кон- кретного метода и его назначением.
2. Щелкните на элементе **Package** в меню страницы, чтобы возвра- титься к странице пакета **java.lang** и изучить другие классы.

Изучите информацию, до- ступную по другим ссылкам в меню, чтобы ознакомиться с документацией ближе.



**Совет**

**83**

**Математические вычисления**

**5. Работа с данными**

В пакете **java.lang** существует класс **Math**, который предоставляет две константы, используемые для математических вычислений. Первая из них — **Math.PI** — хранит значение числа Pi, а вторая — **Math.E** — хра- нит основание натурального логарифма. Обе константы имеют тип **double** (двойной точности, с 15 знаками после запятой).

*Pi.java*

**84**

### Совет

Вычисленное значение π обычно имеет достаточную точность.

1. Создайте новую программу с именем **Pi**, содержащую стандартный метод **main**.

### class Pi



**JAVA**

### {

### public static void main ( String[] args ) { }

### }

1. Внутри главного метода объявите и проинициализируйте пере- менную типа **float** из аргумента командной строки, а также вто- рую переменную типа **float**, которую нужно получить с помощью приведения типов из константы **Math.PI**.

### float radius = Float.parseFloat( args[0] ) ; float shortPi = (float) Math.PI ;

1. Проведите математические вычисления, используя приведенные зна- чения и присваивая результаты еще двум переменным типа **float**.

### float circ = shortPi \* ( radius + radius ) ; float area = shortPi \* ( radius \* radius ) ;

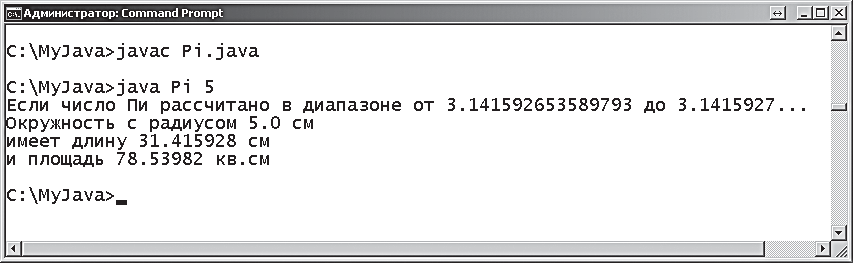


1. Выведите значение константы **Math.PI**, а также ее эквивалента, при- веденного к типу **float**, затем выведите результаты вычислений.

### System.out.print( "Если число Пи рассчитано в диапазоне от " + Math.PI ) ; System.out.println( " до " + shortPi + "..." ) ; System.out.println( "Окружность с радиусом " + radius + " см"); System.out.print( "имеет длину " + circ + " см" ) ; System.out.println( " и площадь " + area + " кв.см" ) ;

1. Сохраните программу под именем *Pi.java*, затем скомпилируйте и запустите.

Вышеуказанный класс **Math** предоставляет множество различных мето- дов, полезных для выполнения математических вычислений. Например, используя метод **Math.pow()**, можно любое заданное число возвести в ука- занную степень. Первым аргументом является число, заданное в скобках, а вторым — показатель степени, в которую нужно возвести.



Метод **Math.sqrt()** возвращает квадратный корень числа, указыва- емый в качестве единственного аргумента.

Оба этих метода возвращают значение типа **double**.

* 1. Создайте новую программу с именем **Power**, содержащую стандарт- ный метод **main**.

### class Power

### {

### public static void main ( String[] args ) { }

### }

* 1. Внутри главного метода объявите и проинициализируйте целочис- ленную переменную, используя аргумент командной строки.

### int num = Integer.parseInt( args[0] ) ;

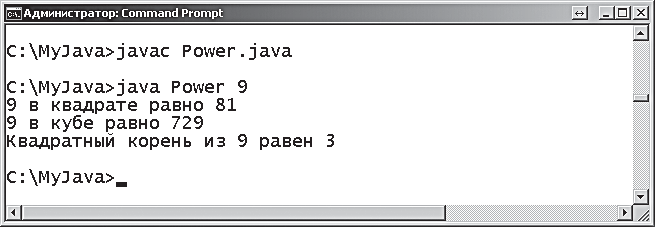
* 1. Произведите математические вычисления, приведя результаты к целочисленному типу.

### int square = (int) Math.pow( num , 2 ) ; int cube = (int) Math.pow( num , 3 ) ; int sqrt = (int) Math.sqrt( num ) ;

* 1. Выведите результаты вычислений.

### System.out.println( num + " в квадрате равно " + square ) ;

### System.out.println( num + " в кубе равно " + cube ) ; System. out.println( "Квадратный корень из " + num + " равен "+ sqrt ) ;

* 1. Сохраните программу под именем *Power.java*, затем скомпилируйте и запустите.



**JAVA**

*Power.java*



**На заметку**

Оба этих примера можно улучшить, если добавить блок операторов **try catch**, чтобы отловить пользо- вательские ошибки, —

см. главу 4.

**85**

**Округление чисел**

**5. Работа с данными**

Класс **Math** внутри пакета **java.lang** предлагает три метода для окру- гления чисел с плавающей точкой до ближайшего целого. Простейший из методов **Math.round()** округляет число, являющееся его аргументом, вверх или вниз до ближайшего целого.

Метод **Math.floor()** производит округление до ближайшего целого вниз, а метод **Math.ceil()** округляет до ближайшего целого вверх.

В то время как метод **Math.round()** возвращает значение целочислен- ного типа (**int**), методы **Math.floor()** и **Math.ceil()** возвращают зна- чение типа **double**.



**JAVA**

*Round.java*

**86**



**Совет**

По умолчанию метод **Math. round()** будет округлять

в сторону увеличения — та- ким образом, 7,5 округлит- ся до 8.

1. Создайте новую программу с именем **Round**, содержащую стандарт- ный метод **main**.

### class Round

### {

### public static void main ( String[] args ) { }

### }

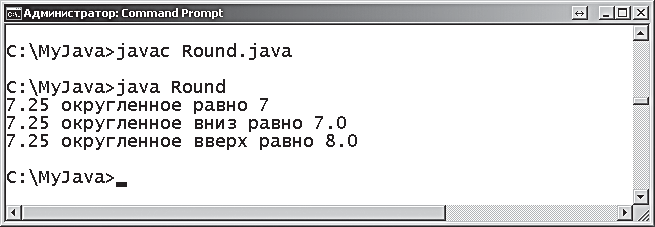
1. Внутри главного метода объявите и проинициализируйте пере- менную типа **float**.

### float num = 7.25f ;

1. Выведите округленное значение, которое приобрело тип **int** в ре- зультате округления.

### System.out.println( num+" округленное равно "+Math.round( num));

1. Выведите округленное значение в виде значений типа **double**. **System.out.println( num+" округленное вниз равно " +Math.floor( num ) ); System.out.println( num+" округленное вверх равно " + Math.ceil( num ) ) ;**
2. Сохраните программу под именем *Round.java*, затем скомпилируй- те и запустите.



Класс **Math** пакета **java.lang** предлагает два метода для сравнения двух числовых значений — это методы **Math.max()** и **Math.min()**, принимаю- щие в качестве двух своих аргументов числа. **Math.max()** возвращает большее из двух чисел, а **Math.min()** — меньшее из них.

Сравниваемые числа, используемые этими двумя методами, могут иметь любой из числовых типов данных, но возвращаемый результат будет всегда иметь тип **double**.

* 1. Создайте новую программу с именем **Compare**, содержащую стан- дартный метод **main**.

### class Compare

### {

### public static void main ( String[] args ) { }

### }

* 1. Внутри главного метода объявите и проинициализируйте две пере- менных типа **float** и типа **int**.

### float num1 = 24.75f ; int num2 = 25 ;

* 1. Выведите большее из двух значений.

### System.out.println( "Наибольшее: " + Math.max( num1, num2 ) ) ;

* 1. Выведите меньшее из двух значений.

### System.out.println( "Наименьшее: " + Math.min( num1, num2 ) ) ;

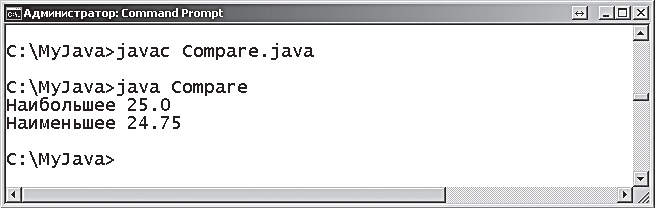
* 1. Сохраните программу под именем *Compare.java*, затем скомпили- руйте и запустите.



**JAVA**

*Compare.java*

**87**



**Генерация случайных чисел**

**5. Работа с данными**

Класс **Math** пакета **java.lang** предоставляет возможность для генера- ции случайных чисел при помощи метода **Math.random()**, который воз- вращает случайное число двойной точности (типа **double**) в диапазоне от 0,0 до 0,999. Желаемый диапазон можно расширить при помощи умножения на случайное число. Например, умножив на 10, можно соз- дать случайное число от 0,0 до 9,999. После этого, если округлить по- лученное число при помощи метода **Math.ceil()**, то полученное целое число попадает в диапазон от 1 до 10.



**JAVA**

*Random.java*

**88**



**Совет**

Программа, моделирующая лотерею и описанная далее, комбинирует все эти три шага в одном операторе.

1. Создайте новую программу с именем **Random**, содержащую стан- дартный метод **main**.

### class Random

### {

### public static void main ( String[] args ) { }

### }

1. Внутри главного метода присвойте переменной типа **float** значе- ние случайного числа и выведите это значение.

### float random = (float) Math.random() ; System.out.println( "Случайное число: " + random ) ;

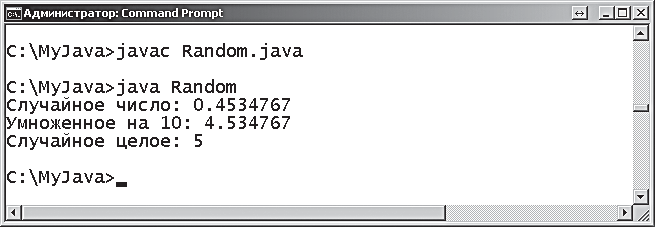
1. Второй переменной типа **float** присвойте значение случайного числа, умноженное на 10, и также выведите ее значение.

### float multiplied = random \* 10 ;

### System.out.println( "Умноженное на 10: " + multiplied ) ;

1. Теперь занесите округленное целое значение, полученное в преды- дущем шаге случайного числа, в переменную типа **int**, затем вы- ведите ее значение.

### int randomInt = (int) Math.ceil( multiplied ) ; System.out.println( "Случайное целое: " + randomInt ) ;

1. Сохраните программу под именем *Random.java*, затем скомпили- руйте и запустите.

При помощи метода **Math.random()** можно сгенерировать последова- тельность из шести неповторяющихся целых чисел в диапазоне от 1 до 49 включительно, чтобы смоделировать выпадение номеров в ло- терее.

* 1. Создайте новую программу с именем **Lottery**, содержащую стан- дартный метод **main**.



**JAVA**

### class Lottery

### {

### public static void main ( String[] args ) { }

### }

*Lottery.java*

* 1. Внутри главного метода создайте целочисленный массив из 50 элементов, а затем заполните элементы с 1 по 49 целыми числа- ми от 1 до 49.

### int[] nums = new int[50] ;

### for( int i = 1 ; i < 50 ; i++ ) { nums[i] = i ; }

* 1. Перемешайте значения в элементах массива от 1 до 49.

### for( int i = 1 ; i < 50 ; i++ )

### {

### int r = (int) Math.ceil( Math.random() \* 49 ) ; int temp = nums[i] ;

**89**

### nums[i] = nums[r] ; nums[r] = temp ;

### }

* 1. Выведите только те значения, которые содержатся в элементах от 1 до 6.

### for ( int i = 1 ; i < 7 ; i++ )

### {

### System.out.print( Integer.toString( nums[i]) + " " );

### }



**Совет**

* 1. Сохраните программу под именем *Lottery.java*, затем скомпилируй- те и запустите три раза, чтобы произвести три различных последо- вательности чисел.

К данной программе мы вер- немся при изучении графи- ческого пользовательского интерфейса в главе 10.

**5. Работа с данными**

**Array.length** — это свой- ство, но **String.length()** — это метод, поэтому он содержит скобки.

**90**

**Управление строками**

В языке Java строка (элемент типа **String**) представляет собой 0 или более символов, заключенных в кавычки. Корректными строками яв- ляются, например: **String txt1 = "Моя первая строка" ;**

### String txt2 = "" ; String txt3 = "2" ; String txt4 = "null" ;

Пустые кавычки в примере с переменной **txt2** инициализируют пере-

менную, содержащую пустую строку. Числовое значение, присвоенное переменной **txt3**, — это строковое представление числа. Ключевое сло- во языка Java **null** обычно представляет собой отсутствие какого-либо значения, но оно является простым строковым литералом, если заклю- чено в кавычки.



**Внимание**

Строка в языке Java, по существу, является набором символов, каждый из которых содержит собственные данные, подобно элементам масси- ва. Поэтому логично представлять строку как массив символов и, рабо- тая с ней, применять характеристики массива.

Класс **String** является частью базового пакета **java.lang** и предостав- ляет метод **length()**, который возвращает размер строки подобно свойству массива **length**. Каждая строковая переменная создается как экземпляр класса **String**, поэтому методы данного класса можно ис- пользовать, применив точечную запись к имени переменной. Напри- мер, чтобы получить размер строки, сохраненной в переменной **txt**, мы запишем **txt.length()**.

В классе **String** пакета **java.lang** существует альтернатива оператору конкатенации **+** для объединения строковых значений друг с другом — это метод **concat()**, который требует единственного аргумента, опре- деляющего вторую строку, добавляемую к основной. Например, чтобы добавить строку **txt1**, хранящуюся в переменной **txt2**, к строке, храня- щейся в **txt1**, мы напишем **txt1.concat( txt2 )**.

1. Создайте новую программу с именем **StringLength**, содержащую стандартный метод **main**.

### class StringLength

### {

### public static void main ( String[] args ) { }

### }

1. Внутри главного метода создайте и проинициализируйте две пере- менные строкового типа.

### String lang = "Java" ;

### String series = " in easy steps" ;

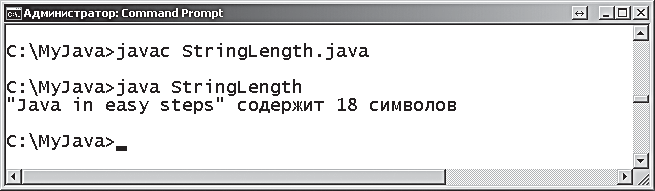
1. Добавьте еще одну строковую переменную и присвойте ей значе- ние двух объединенных выше созданных строк.

### String title = lang.concat( series ) ;

1. Выведите объединенную строку, используя знаки кавычек, и ее размер.

### System.out.print( "\"" + title + "\" содержит " ) ; System.out.println( title.length() + " символов" ) ;

1. Сохраните программу под именем *StringLength.java*, затем скомпи-

лируйте и запустите.



**JAVA**

*StringLength.java*

**91**



**На заметку**

Знаки кавычек не являют- ся частью строки, поэтому не включаются в подсчет символов, в отличие от про- белов.

Будьте внимательны при написании заглавных букв **C** в названии методов

### toUpperCase и toLowerCase.

**Сравнение строк**

Класс **String** пакета **java.lang** предоставляет полезный метод **equals()**, он был показан в главе 4, где оценивался аргумент командной строки, представленный аргументом **args[0]**. Данный метод также можно ис- пользовать для сравнения любых двух строковых переменных, исполь- зуя точечную запись — имя метода ставится после имени переменной, а вторая переменная является аргументом. Например, для сравнения переменных **txt2** и **txt1** запишем **txt1.equals( txt2 )**. В случае если обе переменные содержат одни и те же символы в одинаковом порядке, то метод возвращает значение **true**, в противном же случае возвращает значение **false**.

Строки, которые содержат одинаковые буквы, но в разных регистрах, не являются равными (например, **Java** и **JAVA**), поскольку у символов различаются коды ASCII. Например, значение символа верхнего реги- стра **A** равно 65, в то время как символ нижнего регистра **a** имеет зна- чение 97.

В программах очень часто используется сравнение введенной пользова- телем строки с нужной строкой, чтобы введенная строка соответство- вала определенному регистру. Для этих целей класс **String** предлагает два метода: **toUpperCase()** и **toLowerCase()**. Входная строка указыва- ется этим методам в качестве аргумента, а они, в свою очередь, воз- вращают трансформированную строку. Типичным примером может являться ситуация, когда введенный пользователем строковый пароль преобразуется к нижнему регистру, перед тем как будет сравниваться с верным паролем, хранящимся в программе в нижнем регистре. Это позволит пользователю вводить свой пароль не глядя на регистр, в том случае, если регистры, независимые от паролей, разрешены.



**Внимание**

Точечная запись позволяет использовать методы друг за другом, выполняя операции последовательно. Это означает, что запись **toLowerCase().equals()** может быть использована для преобразова- ния строкового значения к нижнему регистру, а затем сравнения по- лученного результата с указанным аргументом.

1. Создайте новую программу с именем **StringComparison**, содержа- щую стандартный метод **main**.

### class StringComparison

### {

### public static void main ( String[] args ) { }

**92**

**5. Работа с данными**

### }

1. Внутри главного метода создайте и проинициализируйте строко- вую переменную, содержащую корректный пароль из символов нижнего регистра.

### String password = "bingo" ;

1. Добавьте блок операторов **try catch**, для того чтобы перехватить исключение, которое возникнет в случае, если аргумент, представ- ляющий пароль, не будет введен.

### try { }

### catch( Exception e )

### {



**JAVA**

*StringComparison.java*

### System.out.println( "Требуется ввод пароля." ) ;

### }

1. Добавьте блок **if else** в фигурные скобки оператора **try** для оцен- ки введенного пользователем аргумента.

### if ( args[0].toLowerCase().equals( password ) )

### {

### System.out.println( "Пароль подтвержден." ) ;

### }

**93**

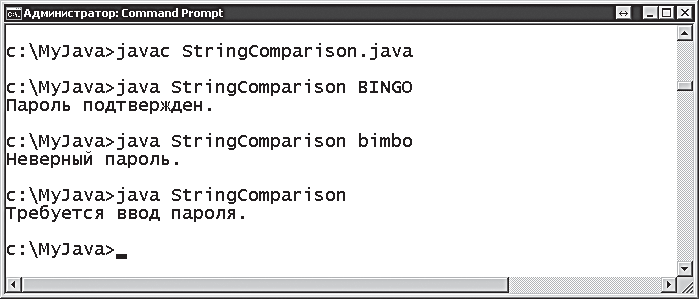
### else

### {

### System.out.println( "Неверный пароль." ) ;

### }

1. Сохраните программу под именем *StringComparison.java*, затем скомпилируйте и запустите.



**Поиск строк**

**5. Работа с данными**

Класс **String** пакета **java.lang** предлагает методы **startsWith()** и **endsWith()** для сравнения частей строк. Данные метода весьма по- лезны, например, при нахождении строк, имеющих одинаковое начало либо одинаковое окончание. Когда часть строки соответствует указан- ному аргументу, метод возвращает значение **true**, в противном случае возвращает значение **false**.

Существует возможность скопировать часть какой-то строки, указав позицию первого символа, с которого нужно начинать копировать, в качестве аргумента для метода **substring()**. Данный метод возвратит так называемую подстроку к оригинальной строке, начиная с указан- ной стартовой позиции и заканчивая последним символом первона- чальной строки.

Метод **substring()** может принимать необязательный второй аргу- мент, который будет указывать позицию последнего копируемого сим- вола. В этом случае метод возвратит подстроку первоначальной стро- ки, начиная с указанной стартовой позиции и заканчивая указанной конечной позицией.

Для поиска символа или подстроки в строке используется метод **indexOf()**. Данный метод возвращает числовую позицию первого вхождения указанного символа или подстроки внутри обрабатывае- мой строки. Если соответствие не найдено, то метод возвращает зна- чение -1.

**94**



**JAVA**

*StringSearch.java*

1. Создайте новую программу с именем **StringSearch**, содержащую стандартный метод **main**.

### class StringSearch

### {

### public static void main ( String[] args ) { }

### }

1. Внутри главного метода создайте и проинициализируйте строко- вый массив, содержащий названия книг.

### String[] books =

### { "Java in easy steps", "XML in easy steps" , "HTML in easy steps" , "CSS in easy steps" , "Gone With the Wind" , "Drop the Defense" } ;

1. Создайте и проинициализируйте три целочисленных переменных- счетчика.

### int counter1 = 0 , counter2 = 0 , counter3 = 0 ;

1. Добавьте цикл **for** для обхода строкового массива, выводя первые четыре символа каждого названия.

### for ( int i = 0 ; i < books.length ; i++ )

### {

### System.out.print( books[i].substring( 0,4 ) + " | " ) ;

### }

1. Добавьте в блок цикла **for** оператор для подсчета названий, содер- жащих указанное окончание.

**95**

### if ( books[i].endsWith( "in easy steps" ) ) counter1++ ;

1. Добавьте еще один оператор в блок цикла **for** для подсчета назва- ний, содержащих указанное начало.

### if ( books[i].startsWith( "Java" ) ) counter2++ ;

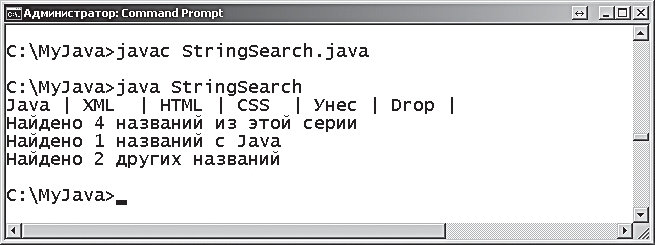
1. Добавьте еще один оператор для подсчета названий, не содержа- щих указанную подстроку.

### if ( books[i].indexOf( "easy" ) == -1 ) counter3++ ;

1. В конец главного метода добавьте следующие операторы для выво- да результатов каждого поиска.

### System.out.println( "\nНайдено " + counter1 + " названий из этой серии" ) ; System.out.println( "Найдено " + counter2 + " названий с Java" ) ; System.out.println( "Найдено " + counter3 + " других названий" ) ;

1. Сохраните программу под именем *StringSearch.java*, затем скомпи-

лируйте и запустите.



**На заметку**

Оператор **! NOT** не может использоваться для провер- ки того, что метод **indexOf()** выдал ошибку, поскольку он возвращает не логическое значение, а целое.

**Обработка символов**

**5. Работа с данными**

Класс **String** пакета **java.lang** предоставляет метод **trim()**, который используется для удаления любых пустых символов в начале и в конце строки, указанной в качестве аргумента этому методу. При этом удаля- ются все лишние пробелы, знаки новой строки и табуляция, и возвра- щается укороченная версия новой строки.

К отдельному символу строки можно обратиться, указав номер, со- ответствующий его позиции в строке, в качестве аргумента методу **charAt()**. Данный метод обращается со строкой как с массивом симво- лов, в котором первый символ находится в нулевой позиции – подобно массивам, чьи элементы индексируются, начиная с нуля. Таким обра- зом, к первому символу строки можно обратиться при помощи записи **charAt(0)**, ко второму символу — **charAt(1)** и так далее.

Поскольку индексы начинаются с нуля, то последний символ в стро- ке будет всегда на единицу меньше, чем общее число символов в этой строке. Это означает, что последний символ будет иметь индекс, эк- вивалентный числу **length() — 1**. Следовательно, чтобы обратить- ся к последнему символу строки с именем **str**, нужно записать **str. charAt( str.length() -1 )**.

Класс **String** предлагает специальный метод **replace()** для замены всех вхождений определенного символа другим символом. Данный ме- тод принимает два аргумента, которые определяют заменяемый символ и символ, который встает на его место. Например, для замены в строке всех символов **a** на символы **z**, мы запишем **replace('a' , 'z' )**.

**96**

Чтобы проверить, содержит ли строка какие-либо символы, существу- ет специальный метод **isEmpty()**. Данный метод возвратит значение **true**, если строка совершенно пустая, в противном случае он будет воз- вращать значение **false**.



**JAVA**

*CharacterSwap.java*

1. Создайте новую программу с именем **CharacterSwap**, содержащую стандартный метод **main**.

### class CharacterSwap

### {

### public static void main ( String[] args ) { }

### }

1. Внутри главного метода объявите и проинициализируйте пустую строковую переменную.

### String txt = "" ;

1. Присвойте переменной определенное значение из каких-либо сим- волов, добавив в начало и конец несколько пробелов.

### if ( txt.isEmpty() ) txt = " Боррокудо " ;

1. Выведите значение строковой переменной и количество символов, которые она содержит.

### System.out.println( "Строка: " + txt ) ;

### System.out.println( "Длина первоначальной строки: " + txt.length() ) ;

1. Удалите пробелы в начале и конце, а затем выведите значение пере- менной и ее размер заново.

### txt = txt.trim() ;

### System.out.println( "Строка: " + txt ) ; System.out.println( "Длина строки: " + txt.length() ) ;

1. Выведите первый символ строки.

### char initial = txt.charAt(0) ;

### System.out.println( "Первая буква: " + initial ) ;

1. Теперь выведите последний символ строки.

### initial = txt.charAt( ( txt.length() -1 ) ); System.out.println( "Последняя буква: " + initial ) ;

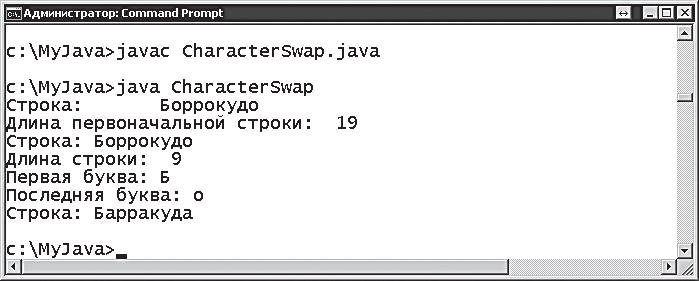
**97**

1. Замените все символы "o", встречающиеся в строке, символами "a".

### txt = txt.replace( 'о' , 'а' ) ; System.out.println( "Строка: " + txt ) ;

1. Сохраните программу под именем *CharacterSwap.java*, затем ском-

пилируйте и запустите.



**Заключение**

**5. Работа с данными**

* + В документации по языку Java можно найти информацию о мето- дах и свойствах для каждого класса.
  + Базовые классы языка Java содержатся в пакете **java.lang**.
  + Класс **Math** предоставляет константы **Math.PI** и **Math.E**.
  + Метод **Math.pow()** осуществляет возведение в указанную степень, а метод **Math.sqrt()** возвращает квадратный корень указанного числа.
  + Числа можно округлять до целых значений, используя методы

### Math.round(), Math.floor() и Math.ceil().

* + При помощи методов **Math.max()** и **Math.min()** можно сравнивать числа.
  + Метод **Math.random()** возвращает случайное число двойной точно- сти в диапазоне от 0,0 до 0,999999999999999.
  + Строка представляет собой 0 или более символов, заключенных в знаки кавычек.
  + Подобно свойству массива **length** метод **length()** возвращает раз- мер строки.

**98**

* + Метод **concat()** используется для добавления одной строки к другой.
  + Метод **equals()** возвращает значение **true** только в том случае, когда две строки содержат идентичные символы в одинаковом по- рядке.
  + Регистр символа строки можно изменять, используя методы

### toUpperCase() и toLowerCase().

* + Используя методы **startsWith()** и **endsWith()**, можно сравнивать строки.
  + При помощи методов **indexOf()** и **substring()** можно производить поиск подстроки в указанной строке.
  + Метод **isEmpty()** возвращает значение **true** только в том случае, когда строка не содержит ни одного символа.
  + Для работы с символами строки используются методы **trim()**, **charAt()** и **replace()**.

1. **Создание классов**

***В этой главе демонстрируется, как создавать Java-программы,***

***применяющие множественные методы***

***и классы.***

# Программа как набор методов

# Область видимости

# Использование множественных классов

# Расширение существующего класса

# Создание объектного класса

# Создание экземпляра объекта

# Инкапсуляция свойств

# Создание объектных данных

# Заключение

**Программа как набор методов**

**6. Создание классов**

Программы обычно разбиваются на отдельные методы — таким обра- зом создаются модули кода, каждый из которых выполняет определен- ную задачу и может быть вызван какое угодно количество раз по мере необходимости. Такая разбивка программы на множественные методы позволяет также легче отслеживать ошибки, поскольку каждый из ме- тодов может быть проверен отдельно. Методы можно объявлять вну- три фигурных скобок, которые следуют за объявлением класса, при этом используются те же самые ключевые слова, что и при объявле- нии главного метода. Любому новому методу нужно давать имя, следуя стандартным договоренностям именования, а также указывать по не- обходимости аргументы в скобках после имени метода.



**JAVA**

*Methods.java*

1. Создайте новую программу с именем **Methods**, содержащую стан- дартный метод **main**.

### class Methods

### {

### public static void main ( String[] args ) { }

### }

1. Внутри фигурных скобок главного метода добавьте операторы для вывода сообщения и вызова второго метода с именем **sub**.

**100**

### System.out.println( "Сообщение из главного метода." ) ; sub() ;

1. После главного метода перед последней фигурной скобкой класса добавьте второй метод для вывода сообщения.

### public static void sub()

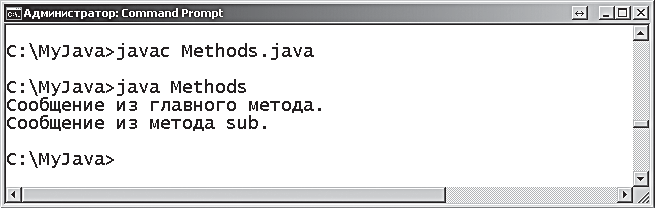
### {

### System.out.println( "Сообщение из метода sub." ) ;

### }



**На заметку**

Если метод вызывается без аргументов, то достаточно написать только его имя

и пустые скобки.

1. Сохраните программу под именем *Methods.java*, затем скомпили- руйте и запустите.

В классе могут содержаться несколько методов с одним и тем же име- нем, только если они имеют либо различное число аргументов, либо аргументы различного типа. Это называется «перегрузкой» методов.

**101**

* 1. Создайте новую программу с именем **Overload**, содержащую стан- дартный метод **main**.

### class Overload

### {

### public static void main ( String[] args ) { }

### }

* 1. Внутри фигурных скобок главного метода добавьте три оператора, вызывающих различные перегруженные методы и передающие им значения аргументов.

### System.out.println( write( 12 ) ) ; System.out.println( write( "Двенадцать" ) ) ; System.out.println( write( 4 , 16 ) ) ;

* 1. После главного метода перед последней фигурной скобкой класса

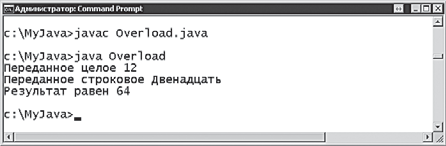
добавьте три перегруженных метода, каждый из которых возвра- щает строку вызвавшему его оператору.

### public static String write( int num )

### { return ( "Переданное целое " + num ) ; } public static String write( String num )

### { return ( "Переданное строковое " + num ) ; } public static String write( int num1 , int num2 )

### { return ( "Результат равен " + ( num1 \* num2 ) ) ; }

* 1. Сохраните программу под именем *Overload.java*, затем скомпили- руйте и запустите.



**JAVA**

*Overload.java*



**Внимание**

В объявлении каждого из перегруженных методов должно быть указано, что метод возвращает строко- вую величину (используется **String**, а не **void**).

**Область видимости**

**6. Создание классов**

Переменная, которая объявляется внутри метода, доступна только в пределах этого метода — ее так называемая область видимости огра- ничена данным методом, в котором она объявлена. Это означает, что в другом методе можно объявить переменную с точно таким же име- нем без какого-либо конфликта.



**JAVA**

*Scope.java*

1. Создайте новую программу с именем **Scope**, содержащую стандарт- ный метод **main**.

### class Scope

### {

### public static void main ( String[] args ) { }

### }

**102**

Переменная-счетчик, объ- явленная в цикле **for**, недо- ступна снаружи цикла — ее область видимости ограни- чена блоком операторов цикла.

1. Внутри фигурных скобок главного метода объявите и проинициа- лизируйте локальную строковую переменную, а затем выведите ее значение.

### String txt = "Это локальная переменная метода main" ; System.out.println( txt ) ;

1. После главного метода перед последней фигурной скобкой класса

добавьте еще один метод с именем **sub**. **public static void sub( ) { }**

1. Внутри фигурных скобок метода **sub** объявите и проинициали- зируйте локальную строковую переменную с тем же именем, что и переменная в главном методе.

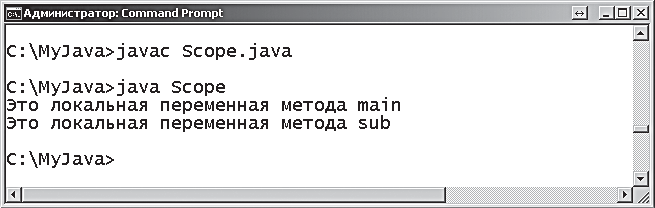
### String txt = "Это локальная переменная метода sub" ; System.out.println( txt ) ;

1. Добавьте вызов метода **sub** в конец метода **main**.

### sub() ;



**На заметку**

1. Сохраните программу под именем **Scope.java**, затем скомпилируй- те и запустите.

Ключевое слово **static**, которое используется в объявлениях метода, указывает, что данный метод является «методом класса» — он досту- пен из любого другого метода данного класса.

**103**

Аналогично, при помощи ключевого слова **static** может быть объяв- лена «переменная класса», которая доступна глобально для всего клас- са. Объявление этой переменной должно быть выполнено перед объ- явлением главного метода сразу после фигурных скобок, следующих за объявлением класса.

В программе могут встретиться одноименные глобальная переменная класса и локальная переменная метода. Локальная переменная мето- да имеет приоритет до тех пор, пока глобальная переменная класса не будет вызвана при помощи точечной записи с использованием имени класса, либо не объявлена локальная переменная с тем же именем.

1. Отредактируйте код в файле *Scope.java*, добавив глобальную пере- менную класса с тем же именем, что и локальная переменная ме- тода.

### final static String txt =

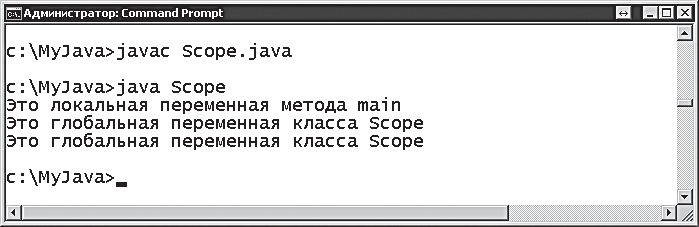
### "Это глобальная переменная класса Scope" ;

1. Добавьте оператор в конец главного метода для вывода значений глобальной переменной класса.

### System.out.println( Scope.txt ) ;

1. Закомментируйте строку, которая объявляет локальную перемен- ную в методе **sub** — таким образом, оператор вывода будет теперь обращаться к глобальной переменной с тем же именем.

### //String txt = "Это локальная переменная метода sub";

1. Сохраните изменения, затем снова скомпилируйте и запустите программу на выполнение, чтобы увидеть изменения.



**Совет**

По возможности используй- те локальные переменные метода, чтобы избегать конфликтов, — глобаль- ные переменные класса используются в основном для констант.

**Использование множественных классов**

**6. Создание классов**

Подобно тому, как программа может содержать несколько методов, большие программы могут состоять из нескольких классов, каждый из которых представляет свою функциональность. Такая модульность предпочтительнее варианта, когда вся программа пишется при помо- щи одного класса, и это делает процесс отладки более простым.

Ключевое слово **public**, которое добавляется при объявлении, является так называемым модификатором доступа, который определяет види- мость элемента для других классов. Его можно использовать в объяв- лении класса, чтобы явно указать, что класс будет доступен (виден) для других классов. Если ключевое слово **public** опущено, по умолчанию разрешается доступ из других локальных классов. Однако для главно- го метода всегда нужно использовать ключевое слово **public**, для того чтобы этот метод был виден компилятору.



**JAVA**

*Multi.java*

**104**

1. Создайте новую программу под именем **Multi**, содержащую стандартный метод **main** (включая ключевое слово **public**, как обычно).

### class Multi

### {

### public static void main ( String[] args ) { }

### }

Компилятор автоматически находит классы в файлах

с расширением *.java* и соз- дает скомпилированные файлы с расширением *.class* для каждого из них.

1. Внутри фигурных скобок главного метода объявите и проинициа- лизируйте переменную строкового типа, а затем выведите ее содер- жимое.

### String msg = "Это локальная переменная класса Multi" ; System.out.println( msg ) ;



**Совет**

1. Выведите содержимое константы с именем **txt** из класса **Data**.

### System.out.println( Data.txt ) ;

1. Вызовите метод с именем **greeting** из класса **Data**. **Data.greeting() ;**
2. Вызовите метод **line** из класса **Draw**. **Draw.line() ;**
3. Сохраните программу под именем *Multi.java*.
4. Начните новый файл с создания класса **Data**. **class Data**

### {

### }

1. Объявите и проинициализируйте константу класса.

### public final static String txt =

### "Это глобальная переменная класса Data" ;

1. Добавьте метод класса.

### public static void greeting

### {

### System.out.print( "Это глобальный метод " ) ; System.out.println( "класса Data" ) ;

### }

1. Сохраните файл под именем *Data.java* в том же каталоге, что и про- грамма *Multi.java*.
2. Создайте новый файл, создав класс **Draw** и метод **line** для доступа по умолчанию — без ключевого слова **public**.

### class Draw

### {

### static void line()

### {



**JAVA**

*Data.java*

**105**



**JAVA**

*Draw.java*

### System.out.println( " ");

### }

### }



**На заметку**

1. Сохраните файл под именем *Draw.java* в том же каталоге, что и про- грамма *Multi.java*, затем скомпилируйте и запустите.

Ключевое слово **public** раз- решает доступ из любого другого класса, а доступ по умолчанию разрешает от- крывать доступ только для классов одного пакета.

**Расширение существующего класса**

**6. Создание классов**

Класс может наследовать свойства любого другого класса, если его объявить при помощи ключевого слова **extends** с указанием имени класса, из которого следует наследовать свойства. Например, следую- щее объявление **class Extra extends Base** наследует из класса **Base**.

Класс, который наследует свойство, называется подклассом, а класс, от которого наследуется свойство, называется суперклассом. В объяв- лении, представленном выше, класс **Base** является суперклассом, а класс **Extra** — подклассом.

Методы и переменные, созданные в суперклассе, обычно могут быть использованы таким образом, как будто они были созданы в подклассе, при условии, если они не объявлены с ключевым словом **private**, кото- рое запрещает доступ извне оригинального класса.

Метод в подклассе будет перегружать метод с тем же именем из супер- класса, только если их аргументы не различаются. К методу суперклас- са можно явно обратиться, используя имя класса и точечную запись. Например, **SuperClass.run()**.

Следует отметить, что оператор **try catch** в любом методе суперкласса не перехватывает исключения, которые появляются в подклассе — вы- зываемый оператор нужно заключать в свой собственный блок опера- торов **try catch** для перехвата таких исключений.

**106**

* 1. Начните новый класс с именем **SuperClass**. **class SuperClass { }**



**JAVA**

*SuperClass.java*

* 1. Внутри фигурных скобок класса добавьте метод, который выводит строку для идентификации.

### public static void hello( )

### {

### System.out.println( "Привет из Суперкласса" ) ;

### }

* 1. Добавьте второй метод, который пытается вывести переданный ар- гумент, затем сохраните файл под именем *SuperClass.java*.

### public static void echo( String arg )

### {

### try

### { System.out.println( "Вы ввели: " + arg ) ; } catch( Exception e )

### { System.out.println( "Требуется аргумент" ) ; }

### }

* 1. Создайте новую программу с именем **SubClass**, которая расширяет класс **SuperClass**.

### class SubClass extends SuperClass

### {

### public static void main ( String[] args ) { }

### }

* 1. После главного метода добавьте еще один метод, который выводит строку для идентификации, перегружая наследуемый метод с тем же именем.

### public static void hello()

### {



**JAVA**

*SubClass.java*

### System.out.println( "Привет из Подкласса" ) ;

### }

**107**

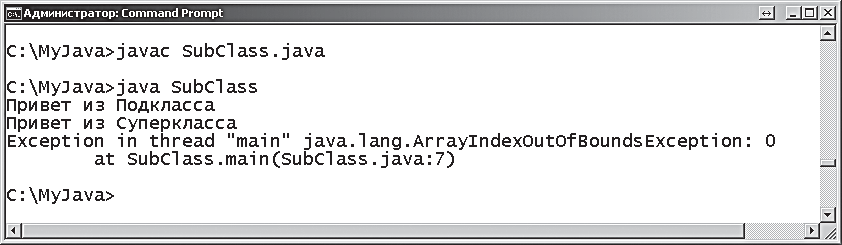
* 1. Внутри фигурных скобок главного метода добавьте вызов пере- гружаемого метода, а затем явный вызов метода с тем же именем в суперклассе.

### hello() ; SuperClass.hello() ;

* 1. Добавьте вызов другого наследуемого метода.

### echo( args[0] ) ;

* 1. Сохраните программу под именем *SubClass.java*, затем скомпили- руйте и запустите, не вводя аргумент в командной строке.



* 1. Отредактируйте файл *SubClass.java*, добавив для вызова метода свой собственный блок операторов **try catch**, перехватывающий исключения, а затем скомпилируйте и заново запустите програм- му, чтобы увидеть, что проблема решена.



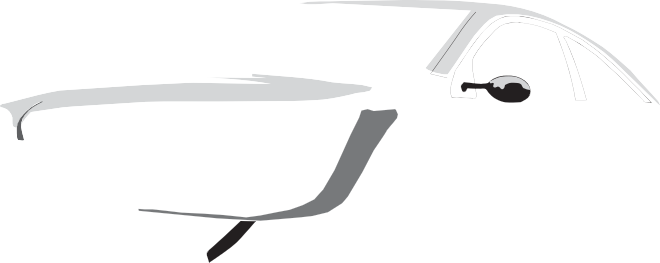
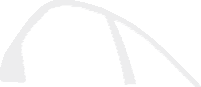
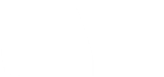
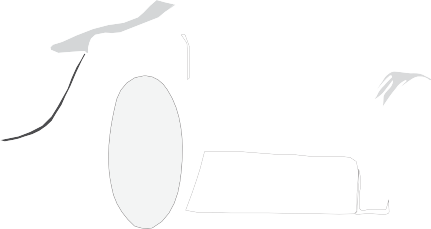
**На заметку**

Более подробную информа- цию по перехвату исключе- ний см. в главе 4.

**Создание объектного класса**

**6. Создание классов**

Объекты реального мира, окружающие нас, можно описать при помо- щи атрибутов (характеристик или свойств), а также действий, которые они выполняют. Например, автомобиль можно описать при помощи атрибутов «красный», «купе», а также при помощи действия «ускоряет- ся». Применительно к Java-программированию все это можно предста- вить при помощи класса **Car**, содержащего свойство **color** и **bodyType**, а также метод **accelerate()**.



Поскольку в Java-программировании широко применяются атри- буты и методы объекта, говорят, что Java является объектно- ориентированным языком программирования.

Объекты в Java создаются при помощи определения класса в качестве шаблона, из которого могут быть произведены копии (экземпляры). Каждому экземпляру класса можно присвоить атрибуты и методы, описывающие объект. Далее мы создаем класс **Car** в качестве шабло- на класса с атрибутами и методами по умолчанию, описанными выше. После этого создается экземпляр класса **Car**, который наследует те же самые атрибуты и методы по умолчанию.

**108**



**JAVA**

*FirstObject.java*

1. Начните новый шаблон класса с именем **Car**. **class Car**

### {

### }

1. Внутри фигурных скобок класса **Car** объявите и проинициализи- руйте две глобальные строковые константы, описывающие атри- буты.

### public final static String color = "Красный" ; public final static String bodyType = "Купе" ;

1. Добавьте глобальный метод, описывающий действие.

### public static String accelerate()

### {

### String motion = "Ускоряется..." ; return motion ;

### }



**На заметку**

1. После класса **Car** начните новый класс с именем **FirstObject**, со- держащий стандартный метод **main**.

### class FirstObject

### {

### public static void main ( String[] args ) { }

**109**

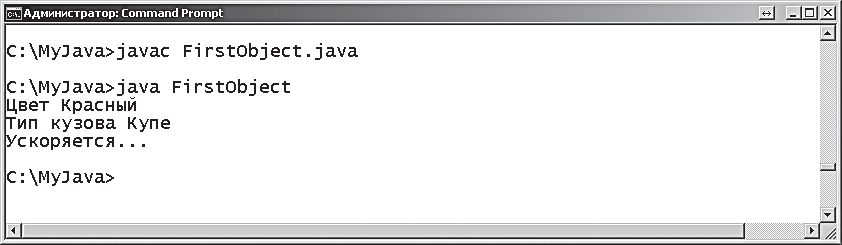
Ключевое слово **static** объявляет переменные и ме- тоды класса в данном случае как члены класса **Car**.

### }

1. Внутри фигурных скобок главного метода добавьте операторы для вывода каждого из значений атрибутов класса **Car**, а также вызова его метода.

### System.out.println( "Цвет " + Car.color ) ; System.out.println( "Тип кузова " + Car.bodyType ) ; System.out.println( Car.accelerate() ) ;

1. Сохраните программу под именем *FirstObject.java*, затем скомпили-

руйте и запустите.



**Совет**

Классы объектов обычно создаются перед классом программы, содержащим метод **main**.

**Создание экземпляра объекта**

**6. Создание классов**

Каждый класс содержит встроенный метод, который можно использо- вать для создания нового экземпляра этого класса. Этот метод называ- ется «конструктор». Он имеет то же самое имя, что и класс, и запуска- ется с ключевым словом **new**.

Каждый экземпляр класса наследует атрибуты и действия объекта. Принцип наследования является одним из основных в Java-програм- мировании. Благодаря ему программы могут использовать «готовые» свойства.

Для большей гибкости шаблоны класса можно определять в отдельном файле, который затем можно использовать в разных программах.



**JAVA**

*Car.java*

**110**

1. Начните новый файл, скопировав шаблон класса **Car** из предыду- щего примера.

### class Car

### {

### public final static String color = "Красный" ; public final static String bodyType = "Купе" ; public static String accelerate()

### {

### String motion = "Ускоряется..." ; return motion ;

### }

### }

1. Сохраните файл под именем *Car.java*.



**JAVA**

*FirstInstance.java*

1. Создайте новую программу с именем **FirstInstance**, содержащую стандартный метод **main**.

### class FirstInstance

### {

### public static void main ( String[] args ) { }

### }

1. Внутри фигурных скобок главного метода добавьте операторы для вывода каждого из значений атрибутов класса **Car**, а затем вызови- те его метод.

### System.out.println( "Цвет автомобиля " + Car.color ) ; System.out.println( "Тип кузова "+ Car.bodyType ) ; System.out.println( Car.accelerate() ) ;

1. Теперь добавьте операторы для создания экземпляра **Porsche** клас-

са **Car**. **Car class**

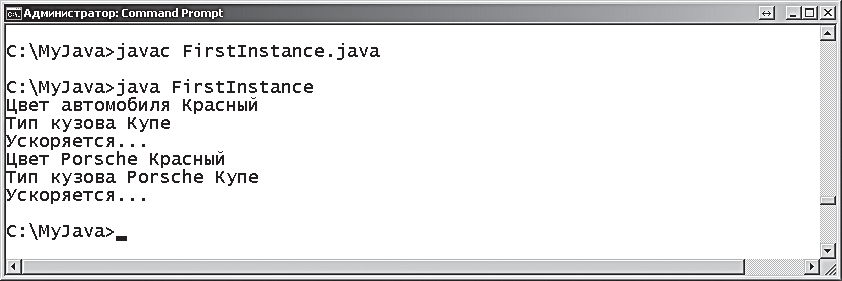
### Car Porsche = new Car() ;

1. Добавьте операторы для вывода унаследованных значений каждого атрибута **Porsche**, а также вызова его метода.

### System.out.println( "Цвет Porsche " + Porsche.color ) ; System.out.println( "Тип кузова Porsche " + Porsche.bodyType ); System.out.println( Porsche.accelerate() ) ;

1. Сохраните программу под именем *FirstInstance.java* рядом с файлом

шаблона *Car.java*, затем скомпилируйте и запустите.



Для нового объекта **Porsche** создается виртуальный класс, который копирует исходный класс **Car**. Оба этих объекта содержат статические переменные класса и метод класса, к которым можно обратиться, ис- пользуя точечную запись и имя класса, но поскольку они доступны глобально, то это является не очень хорошей практикой программи- рования.

В данном примере демонстрируется, как экземпляр объекта наследует свойства исходного класса. В следующем примере используются неста- тические члены — переменная экземпляра и метод экземпляра, к кото- рым нельзя обратиться извне класса, поскольку они глобально не до- ступны, и это является хорошей практикой программирования.

### На заметку

Вы не можете обратиться к переменной **motion** на- прямую — она находится за пределами видимости



внутри объявления метода.

**111**



**Совет**

Компилятор автоматически находит класс **Car** во внеш- нем файле *.java* и создает скомпилированный файл

*.class* для него.

**Инкапсуляция свойств**

**6. Создание классов**

Когда нужно описать переменные и методы объекта и защитить их от обработки внешним программным кодом, используется ключевое сло- во **private** (закрытый). Для вызова значений переменных, а также вы- зова методов, в объект можно включать общедоступные методы. Дан- ный прием инкапсулирует переменные и методы внутри структуры объекта. Это демонстрируется в следующих шагах, которые воспроиз- водят предыдущий пример с инкапсуляцией метода и атрибутов.



**JAVA**

*SafeInstance.java*

1. Начните новый класс с именем **Car**. **class Car**

### {

### }

1. Внутри фигурных скобок класса объявите три закрытые строковые переменные для хранения атрибутов объекта.

### private String maker ; private String color ; private String bodyType ;

**112**

1. Добавьте закрытый метод, описывающий действие.

### private String accelerate()

### {

### String motion = "Ускоряется..." ; return motion ;

### }

1. Добавьте общедоступный метод для присваивания значения пере- данного аргумента каждой из закрытых переменных.

### public void setCar( String brand , String paint , String style )

### {

### maker = brand ; color = paint ; bodyType = style ;

### }

1. Добавьте еще один общедоступный метод, обращающийся к значе- ниям закрытых переменных и вызывающий закрытый метод.

### {

### System.out.println( maker + " цвет " + color ) ; System.out.println( maker + " тип кузова " + bodyType ) ; System.out.println( maker + " " + accelerate() + "\n" ) ;

### }

1. После класса **Car** начните еще один класс с именем **SafeInstance**, содержащий стандартный метод **main**.

### class SafeInstance

### {

### public static void main ( String[] args) { }

### }



**На заметку**

**113**

1. Внутри фигурных скобок главного метода добавьте оператор, соз- дающий экземпляр класса **Car**.

### Car Porsche = new Car() ;

1. Добавьте оператор, который вызывает общедоступный метод клас- са **Car** для присваивания значений его закрытым переменным.

### Porsche.setCar( "Porsche" , "Красный" , "Купе" ) ;

1. Теперь добавьте оператор для вызова еще одного общедоступного метода класса **Car**, вызывающего сохраненные значения атрибутов, а также закрытый метод, определяющий действие.

### Porsche.getCar() ;

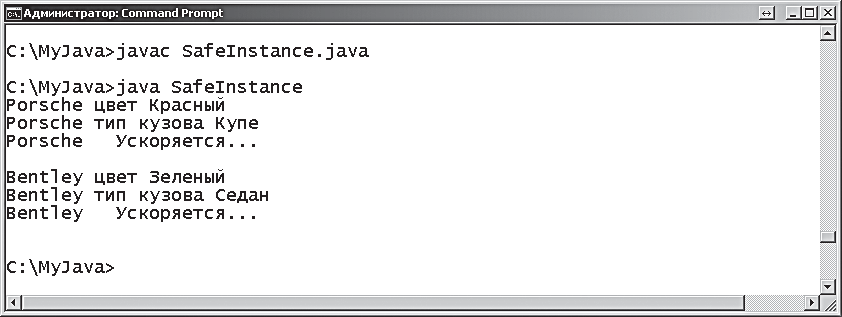
1. Создайте еще один экземпляр, сначала присвоив, а потом вызвав значение.

### Car Bentley = new Car() ;

### Bentley.setCar( "Bentley" , "Зеленый" , "Седан" ) ; Bentley.getCar() ;

1. Сохраните программу под именем *SafeInstance.java*, затем компили- руйте и запустите.

Проинициализированная строковая переменная хранит значение **null** — та- ким образом, вызов метода **getCar()** перед методом **setCar()** возвратит значение **null** каждой из переменных.



**Создание объектных данных**

**6. Создание классов**

Метод конструктора объекта можно вызывать напрямую в классе объекта, для того чтобы проинициализировать его переменные. Это позволит хранить отдельно объявление и присваивание, что является хорошим стилем программирования. Этот пример демонстрируется в следующих шагах, которые воспроизводят предыдущий пример с ин- капсулируемыми методом и атрибутами вместе с инициализацией при помощи конструктора.



**JAVA**

*Constructor.java*

1. Начните новый класс с именем **Car**. **class Car**

### {

### }

1. Внутри фигурных скобок класса объявите три закрытые строковые переменные для хранения атрибутов объекта.

### private String maker ; private String color ; private String bodyType ;

**114**

1. Добавьте метод constructor, в котором инициализируются все три

переменные со значениями атрибутов.

### public Car()

### {

### maker = "Porsche" ; color = "Серебристый" ; bodyType = "Купе" ;

### }



**Внимание**

В объявлениях конструк- тора не указываются типы данных.

1. Добавьте закрытый метод, описывающий действие.

### private String accelerate()

### {

### String motion = "Ускоряется..." ; return motion ;

### }

1. Добавьте общедоступный метод для присваивания значений пере- данного аргумента каждой из закрытых переменных.

### public void setCar( String brand , String paint , String style )

### {

### maker = brand ; color = paint ; bodyType = style ;

### }

1. Добавьте еще один общедоступный метод, который выводит значе- ния закрытых переменных и вызывает закрытый метод.

### public void getCar()

### {

### System.out.println( maker + " цвет " + color ) ; System.out.println( maker + " тип кузова " + bodyType ) ; System.out.println( maker + " " + accelerate() + "\n" ) ;

### }

1. После класса **Car** добавьте еще один класс с именем **Constructor**, содержащий стандартный метод **main**.

### class Constructor

### {

### public static void main ( String[] args ) { }

**115**

### }

1. Внутри фигурных скобок главного метода добавьте операторы для создания экземпляра класса **Car** и вызова первоначальных значе- ний по умолчанию.

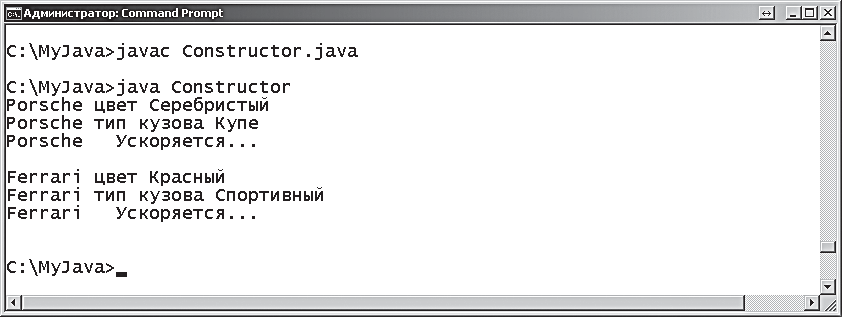
### Car Porsche = new Car() ; Porsche.getCar() ;

1. Создайте еще один экземпляр, присвоив и вызвав значения.

### Car Ferrari = new Car() ;

### Ferrari.setCar( "Ferrari" , "Красный" , "Спортивный" ) ; Ferrari.getCar() ;

1. Сохраните программу под именем *Constructor.java*, затем скомпи- лируйте и запустите.



**Заключение**

**6. Создание классов**

* + Разбиение программ на многочисленные методы, которые можно вызывать при необходимости, увеличивает гибкость и упрощает отслеживание ошибок.
  + Перегруженные методы имеют то же имя, но принимают другие ар- гументы.
  + Переменные, объявленные внутри метода, имеют локальную об- ласть видимости, а переменные класса — глобальную область ви- димости в пределах этого класса.
  + Ключевое слово **static** используется для объявления методов и пе- ременных класса, которые имеют глобальную область видимости в пределах этого класса.
  + Ключевое слово **public** явно разрешает доступ из любого класса.
  + Объявление класса может содержать ключевое слово **extends**, в ко- тором указывается суперкласс, откуда объявляемый класс будет наследовать свойства.
  + Для того чтобы явно обращаться к определенному методу класса или к его переменной, можно использовать имя класса и точечную запись.

**116**

* + Объекты реального мира имеют атрибуты и действия, которые можно представить в программах с помощью переменных и ме- тодов.
  + Объекты в Java создаются как шаблон класса, из которого можно сделать копии (экземпляры).
  + У каждого класса существует метод **constructor**, который можно запустить с помощью ключевого слова **new** для создания экземпля- ра этого класса.
  + Экземпляры наследуют атрибуты и методы класса, из которого они произведены.
  + Инкапсуляция защищает переменные экземпляра и методы экзем- пляра от доступа из внешних классов.
  + Ключевое слово **private** запрещает доступ снаружи класса.
  + Метод конструктора можно вызывать для инициализации атрибу- тов этого объекта.

**Импортирование функций**

**7**

***В данной главе демонстрируется, как из специализированных***

***Java-классов импортировать в программы дополнительную***

***функциональность.***

# Работа с файлами

# Чтение консольного ввода

# Чтение файлов

# Запись файлов

# Сортировка элементов массива

# Создание списочных массивов

# Работа с датой

* **Форматирование чисел**
* **Заключение**

**Работа с файлами**

**7. Импортирование функций**

Java содержит пакет с именем **java.io**, предназначенный для работы с процедурами ввода и вывода в файл. Такой пакет можно сделать до- ступным, если в самое начало файла *.java* включить оператор **import**. Для этого используется шаблон \*, означающий включение всех классов пакета. Например, **import java.io.\* ;**.

В пакете **java.io** содержится класс с именем **File**, который может быть использован для доступа к файлам или целым каталогам. Вначале нуж- но создать объект **File**, используя ключевое слово **new** и указывая имя файла или имя каталога в качестве аргумента для конструктора. На- пример, синтаксис, создающий объект **File** с именем **info**, который будет представлять файл в локальной системе под именем *info.dat*, вы- глядит следующим образом:

### File info = new File( "info.dat" ) ;

Данный файл будет расположен в том же каталоге, что и программа, но аргумент может содержать полный путь к файлу, расположенному в любом месте. Обратите внимание, что при создании объекта **File** в действительности файл не создается, а происходит лишь представле- ние конкретного элемента файловой системы.

**118**

Как только объект **File**, представляющий конкретный файл системы, создан, можно вызывать его методы для работы с файлом. Наиболее используемые и полезные методы объекта **File** с кратким их описани- ем перечислены в таблице ниже.



**На заметку**

Имя файла, указанного

в качестве аргумента для

конструктора, должно быть заключено в кавычки.

|  |  |
| --- | --- |
| **Метод Возвращает** | |
| **exists()** | **true**, если файл существует, **false** — в про- тивном случае |
| **getName()** | имя файла в виде строки |
| **length()** | размер файла в байтах (тип **long** ) |
| **createNewFile()** | **true**, если удалось создать новый файл |
| **delete()** | **true** в случае успешного удаления файла |
| **renameTo(File)** | **true** в случае успешного переименования файла |
| **list()** | массив имен файлов или каталогов (стро- кового типа) |

1. Создайте новую программу, которая импортирует функции всех классов пакета **java.io**.

### import java.io.\* ;

1. Добавьте класс с именем **ListFiles**, содержащий стандартный ме- тод **main**.

### class ListFiles

### {



**JAVA**

*ListFiles.java*

### public static void main( String[] args ) { }

### }

1. Внутри фигурных скобок главного метода добавьте оператор, соз- дающий объект **File** для каталога с именем **data**.

### File dir = new File( "data" ) ;

1. Добавьте оператор **if** для вывода имен всех файлов в данном ката- логе либо вывода сообщения в случае, если каталог пустой.

### if ( dir.exists() )

### {

### String[] files = dir.list() ;

**119**

### System.out.println( files.length + " файлов найдено..." ) ; for ( int i = 0 ; i < files.length ; i++ )

### {

### System.out.println( files[i] ) ;

### }

### }

### else

### { System.out.println( "Каталог не найден." ) ;

1. Сохраните программу под именем *ListFiles.java* внутри каталога *data*, содержащего несколько файлов, затем скомпилируйте и за- пустите; вы увидите перечисленные имена файлов в качестве вы- вода.



**Чтение консольного ввода**

**7. Импортирование функций**

Пакет **java.io** позволяет в программах на языке Java читать дан- ные, вводимые пользователем в командной строке. Подобно объек- ту **System.out**, отправляющему вывод в командную строку, объект **System.in** можно использовать для чтения из командной строки при помощи объекта **InputStreamReader**. Вводимые данные читаются по- байтно, затем конвертируются в целочисленные значения, представ- ляющие значения символов в формате Unicode.

Для того чтобы прочитать целую строку вводимого текста, метод **readLine()** объекта **BufferedReader** читает символы, декодированные при помощи объекта **InputStreamReader**. Данный метод следует вызы- вать внутри блока операторов **try catch**, чтобы перехватывать исклю- чения **IOException** (исключения, вызванные вводом-выводом).

Обычно метод **readLine()** заносит введенные данные в строковую пе- ременную для следующей обработки в программе.

* 1. Создайте новую программу, которая импортирует функциональ- ность всех классов. **java.io**



**JAVA**

### import java.io.\* ;

*ReadString.java*

**120**

* 1. Добавьте класс с именем **ReadString**, содержащий стандартный ме- тод **main**.

### class ReadString

### {

### public static void main( String[] args ) { }

### }

* 1. Внутри фигурных скобок главного метода добавьте оператор для вывода сообщения, предлагающего пользователю ввести данные.

### System.out.print( "Введите название данной книги: " ) ;

* 1. Добавьте оператор, создающий объект **InputStreamReader**, позво- ляющий читать вводимые с командной строки данные.

### InputStreamReader isr =

### new InputStreamReader( System.in ) ;

* 1. Создайте объект **BufferedReader** для чтения декодированного ввода.

### BufferedReader buffer = new BufferedReader( isr ) ;

* 1. Объявите и проинициализируйте пустую строковую переменную, в которой будет сохраняться ввод.

### String input = "" ;

* 1. Добавьте блок операторов **try catch**, в котором будет происходить чтение данных, вводимых из командной строки, и сохранение их в переменной.

### try

### {

### input = buffer.readLine() ; buffer.close() ;

**Совет**

Хорошей практикой явля- ется вызов метода **close()** объекта **BufferedReader**, если он больше не нужен.

### }



### catch ( IOException e )

### {

### System.out.println( "Произошла ошибка ввода" ) ;

### }

* 1. Выведите сообщение, включающее сохраненное значение ввода.

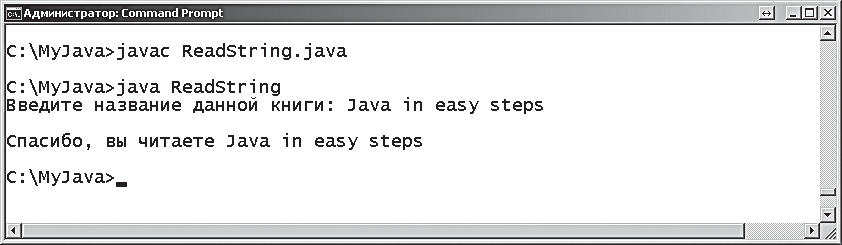
### System.out.println( "\nСпасибо, вы читаете " + input ) ;

**121**

* 1. Сохраните программу под именем *ReadString.java*, затем скомпили- руйте и запустите.



* 1. Введите запрашиваемый текст, нажмите клавишу **Enter**, и вы уви- дите выводимое сообщение, содержащее этот текст.



**Чтение файлов**

**7. Импортирование функций**

Пакет **java.io** содержит класс с именем **FileReader**, который специ- ально предназначен для чтения текстовых файлов. Он является под- классом класса **InputStreamReader**, который используется для чтения консольного ввода и преобразует поток байтов в целые числа, пред- ставляющие символы в формате ЮНИКОД.

Объект **FileReader** создается при помощи ключевого слова **new**, а так- же имени файла, из которого нужно читать, в качестве аргумента. Ар- гумент может включать полный путь к файлу за пределами каталога, где расположена программа.

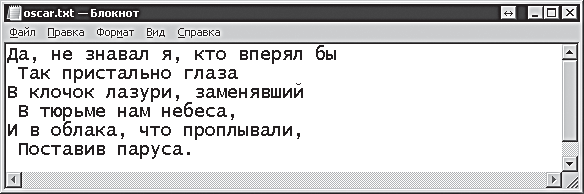
Для эффективного чтения текста строка за строкой используется метод **readLine()** объекта **BufferedReader**, который читает символы, декоди- рованные объектом **FileReader**.

Данный метод следует вызывать внутри блока **try catch** для перехвата возможных исключений ввода-вывода.

Для того чтобы прочитать все строки текстового файла, в цикле вы- зывается метод **readLine()**. В конце файла он возвратит значение **null**, которое можно использовать для выхода из цикла.

* + 1. Откройте простой текстовый редактор, скажем, Блокнот (Notepad), и введите несколько строк текста, например знаменитую фразу Оскара Уайльда из произведения «Баллада Редингской тюрьмы».

**122**



* + 1. Сохраните текстовый файл под именем *oscar.txt*, затем создайте но- вую программу, которая импортирует функции всех классов паке- та **java.io**.

### import java.io.\* ;

* + 1. Добавьте класс с именем **ReadFile**, содержащий стандартный ме- тод **main**.

### class ReadFile

### { public static void main( String[] args ) { } }

* + 1. Внутри фигурных скобок главного метода добавьте блок операто- ров **try catch**.

### try { }

### catch ( IOException e )

### {

### System.out.println( "Произошла ошибка чтения" ) ;

### }



**На заметку**

* + 1. Внутри фигурных скобок блока **try** добавьте оператор для созда- ния объекта **FileReader**.

### FileReader file = new FileReader( "oscar.txt" ) ;

* + 1. Создайте объект **BufferedReader** для чтения файла.

### BufferedReader buffer = new BufferedReader( file ) ;

* + 1. Объявите и проинициализируйте пустую строковую переменную для хранения строки текста.

### String line = "" ;

* + 1. Добавьте цикл для чтения в переменную содержимого текстового файла, а также для вывода всех строк текста.

### while ( ( line = buffer.readLine() ) != null )

### { System.out.println( line ) ; }

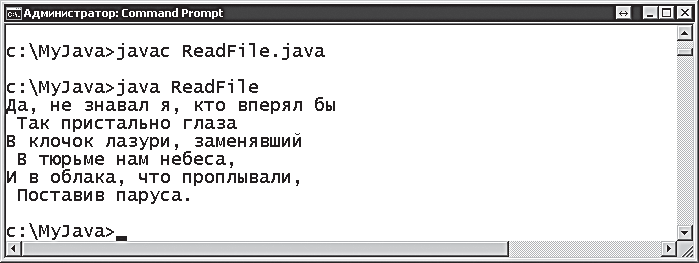
* + 1. Не забудьте закрыть объект **BufferedReader**, поскольку он больше не нужен.

### buffer.close() ;

* + 1. Сохраните программу под именем *ReadFile.java* рядом с текстовым файлом, затем скомпилируйте и запустите.

Имя текстового файла, ука- занного в качестве аргумен- та для **FileReader**, должно быть заключено в кавычки.

**123**





**JAVA**

*WriteFile.java*

**124**

**Запись файлов**

В пакете **java.io** классы **FileReader** и **BufferedReader**, использующие- ся для чтения текстовых файлов, имеют аналоги с именами **FileWriter** и **BufferedWriter**, которые можно использовать для записи текстовых файлов.

Объект **FileWriter** создается с использованием ключевого слова **new** и принимает имя файла, в который нужно производить запись, в ка- честве аргумента. Аргумент может включать полный путь к файлу за пределами каталога, в котором расположена программа.

Объект **BufferedWriter** создается с использованием ключевого слова **new** и принимает в качестве аргумента имя объекта **FileWriter**. После этого при помощи метода **write** объекта **BufferedWriter** можно запи- сывать текст, в котором строки разделяются вызовом метода **newLine()**. Эти методы следует располагать внутри блока операторов **try catch** для перехвата возможных исключений ввода-вывода.

Если файл с указанным именем уже существует, то метод **write()** ста- нет перезаписывать содержимое файла, в противном случае будет соз- даваться новый файл с этим именем, в который станет записываться содержимое.

1. Создайте новую программу, которая импортирует функции всех классов пакета **java.io**.

### import java.io.\* ;

1. Добавьте класс с именем **WriteFile**, содержащий стандартный ме- тод **main**.

### class WriteFile

### {

### public static void main ( String[] args ) { }

**7. Импортирование функций**

### }

1. Внутри фигурных скобок главного метода добавьте блок операто- ров **try catch**.

### try { }

### catch ( IOException e )

### {

### System.out.println( "Произошла ошибка записи" ) ;

### }

1. Внутри фигурных скобок блока **try** добавьте оператор для созда- ния объекта **FileWriter** для текстового файла с именем *tam.txt*.

### FileWriter file = new FileWriter( "tam.txt" ) ;

1. Создайте объект **BufferedWriter** для записи файла.

### BufferedWriter buffer = new BufferedWriter( file ) ;

1. Добавьте операторы для записи строк текста, а также символов новой строки в текстовый файл. Например, перевод произведения

«Тэм О’Шентер» Роберта Бернса.

### buffer.write( "Дул ветер из последних сил," ) ; buffer.newLine() ;

### buffer.write( "И град хлестал, и ливень лил," ); buffer.newLine() ;

### buffer.write( "И вспышки молний тьма глотала," ) ; buffer.newLine() ;

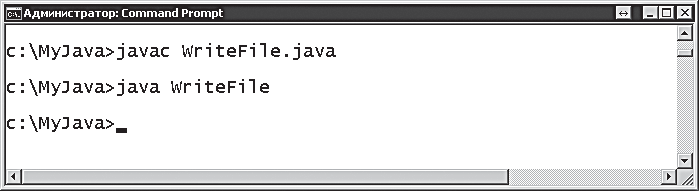
### buffer.write( "И небо долго грохотало..." ) ; buffer.newLine() ;

### buffer.write( "В такую ночь, как эта ночь," ) ; buffer.newLine() ;

### buffer.write( "Сам дьявол погулять не прочь." ) ;

1. Не забудьте закрыть объект **BufferedWriter**, поскольку он больше не нужен.

### buffer.close() ;

1. Сохраните программу под именем *WriteFile.java*, затем скомпили- руйте и запустите; она запишет текстовый файл в тот же каталог, где находится программа.

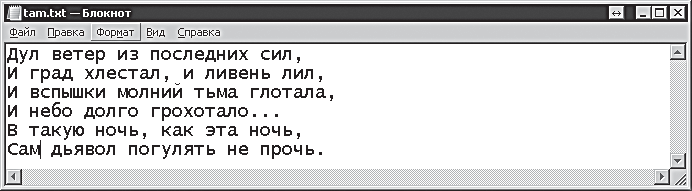


**Совет**

Вместо того чтобы переза- писывать текст в файле при помощи метода **write()**, вы можете использовать метод **append()** объекта

**BufferedWriter** для добавле- ния текста в файл.

**125**





**JAVA**

*Sort.java*

**126**

Более подробно о перегруз- ке методов см. в главе 6.

**Сортировка элементов массива**

В Java содержится пакет с именем **java.util**, который предлагает по- лезные утилиты для работы с набором данных. Данный пакет можно сделать доступным любой программе, включив в самое начало файла

*.java* оператор **import**. Например, используя шаблон \*, можно импорти- ровать все классы данного пакета: **import java.util.\* ;**.

В пакете **java.util** содержится класс с именем **Arrays**, предлагающий методы, которые можно использовать для работы с массивами. Данная функциональность может быть доступна в программах, если импор- тировать все классы пакета **java.util**, либо, если программа требует только одного класса, указать в операторе **import** определенное имя класса. Например, для импорта класса **Arrays** запишем **import java. util.Arrays ;**.

В классе **Arrays** содержится метод **sort()**, который может упорядочи- вать элементы массива либо в алфавитном порядке, либо численно.

1. Создайте новую программу, импортирующую функциональность класса **java.util.Arrays**.

### import java.util.Arrays ;

1. Добавьте класс с именем **Sort**, содержащий стандартный метод

### main.

### class Sort

### { public static void main( String[] args ) { } }

1. После метода **main** добавьте метод для отображения содержимого всех элементов переданного строкового массива.

### public static void display( String[] elems )

### {



**Совет**

### System.out.println( "\nСтроковый Массив:" ) ; for ( int i = 0 ; i < elems.length ; i++ )

### System.out.println( "Элемент "+i+" — "+elems[i] ) ;

### }

1. Добавьте перегруженную версию метода **display()** для отображе- ния содержимого всех элементов переданного целочисленного мас- сива.

**7. Импортирование функций**

### public static void display( int[] elems )

### {

### System.out.println( "\nЦелочисленный Массив:" ) ; for ( int i = 0 ; i < elems.length ; i++ ) System.out.println( "Элемент "+i+" — "+elems[i] ) ;

### }

1. Внутри фигурных скобок главного метода объявите и проинициа- лизируйте строковый и целочисленный массивы.

### String[] names = { "Майк" , "Дэйв" , "Энди" } ; int[] nums = { 200 , 300 , 100 } ;

1. Выведите содержимое всех элементов каждого массива.

### display( names ) ; display( nums ) ;

1. Отсортируйте содержимое элементов обоих массивов.

### Arrays.sort( names ) ; Arrays.sort( nums ) ;

1. Выведите содержимое всех элементов каждого массива еще раз.

### display( names ) ; display( nums ) ;

1. Сохраните программу под именем *Sort.java*, затем скомпилируйте

и запустите.



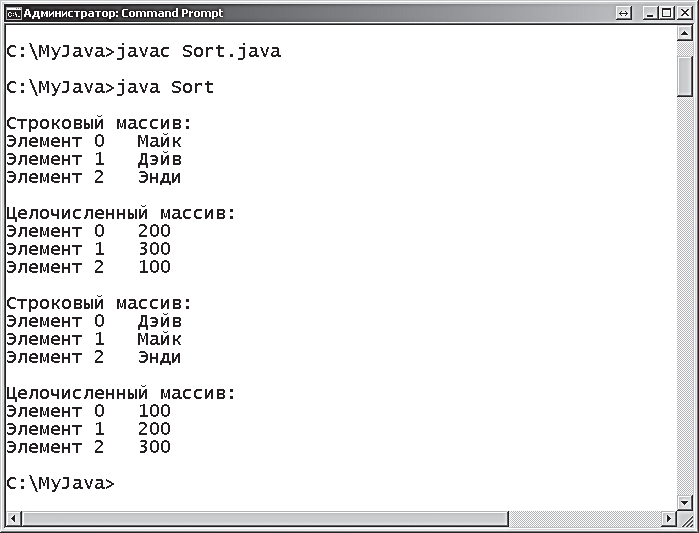
**На заметку**

В данном примере в каждом из циклов **for** выполняет-

ся только один оператор,

поэтому фигурных скобок не требуется, но для ясности иногда можно их добавлять.

**127**



**7. Импортирование функций**

Найти количество элемен- тов, содержащихся в списке, вы можете, вызвав метод **size()**.

**128**

Лямбда-выражения появи- лись в версии Java 8 для краткой записи анонимных методов.

**Создание списочных массивов**

В пакете **java.util** содержится класс с именем **ArrayList**, который хра- нит данные в упорядоченном «наборе», являющемся последовательно- стью с изменяемым размером. Этот класс делается доступным в про- грамме при помощи команды импорта: **import java.util.ArrayList;**. Список может содержать повторяющиеся элементы, и объект **ArrayList** имеет полезные методы, позволяющие работать с элементами массива, используя их порядковые номера (индексы). Например, вызов **get(0)** позволяет получить значение, хранящееся в первом элементе, в то вре- мя как **remove(1)** удаляет второй элемент списка.



**Совет**

При помощи метода списка **set()** можно изменять значения элемен- тов, указав в качестве аргументов этому методу порядковый номер и новое значение. Метод **add()** позволяет в определенное место списка добавлять элементы, указав в качестве аргументов порядковый номер и значение. При этом список расширяется, в него включаются допол- нительные элементы, а порядковые номера сдвигаются.

Объект **ArrayList** создается при помощи ключевого слова **new**, но, как и другие наборы Java, при создании нужно указывать тип элемента, в котором будет содержаться список. Обычно списки содержат стро- ковые элементы, поэтому объект **ArrayList** должен иметь суффикс

### <String>.

Такие наборы, как **ArrayList** содержат метод **forEach()**, с помощью ко- торого можно обходить элементы списка.

Элементы списка можно также обойти при помощи «лямбда- выражения», которое можно указать методу **forEach()** в качестве аргу- мента. Лямбда-выражения — это простые и короткие анонимные ме- тоды, которые определяются в том же месте, где и выполняются. Они начинаются со скобок, содержащих аргументы, затем ставится после- довательность символов **->**, после которой идет блок операторов:

**( *аргумент/-ы* ) -> { *оператор/-ы* }**

Тип данных аргументов может быть объявлен либо явно, либо может подразумеваться из контекста — **(String x)** может быть заменено на **(x)**. Кроме того, фигурные скобки могут быть опущены, если блок опе- раторов лямбда-выражений содержит только один оператор.

**Новинка**

При помощи метода **forEach()** значение текущего элемента на каждой итерации может быть передано лямбда-выражению в качестве его ар- гумента, а затем отображено в выводе при помощи его оператора.

1. Создайте новую программу, которая импортирует все методы клас- са **java.util.ArrayList**.

### import java.util.ArrayList ;

1. Добавьте класс с именем **Lists**, содержащий стандартный метод

### main.

### class Lists

### { public static void main( String[] args ) { } }

1. Внутри фигурных скобок главного метода добавьте оператор для создания строкового списочного массива с именем **list**.

### ArrayList<String> list = new ArrayList<String>() ;

1. Далее добавьте операторы, заполняющие элементы списка строко- выми значениями, а затем выведите весь список.

### list.add( "Альфа" ) ;

### list.add( "Дельта" ) ; list.add( "Чарли" ) ; System.out. println( "Список: " + list ) ;

1. Теперь идентифицируйте текущее значение, хранящееся во втором элементе, а затем замените его новым строковым значением.

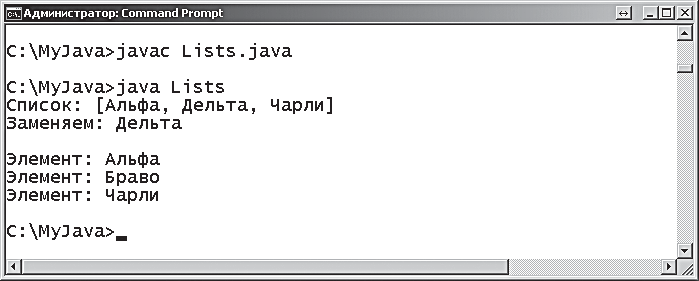
### System.out.println( "Заменяем: " + list.get(1) + "\n" ) ; list.set( 1, "Браво" ) ;

1. Наконец, сделайте обход списка и выведите строковое значение,

хранящееся в каждом элементе.

### list.forEach( ( x ) -> System.out.println( "Элемент: " + x )

### ) ;

1. Сохраните программу под именем **Lists.java**, затем скомпили- руйте и запустите.



**JAVA**

*Lists.java*



**На заметку**

Так же, как и обычные мас- сивы, элементы списочного массива нумеруются, начи- ная с нуля.



**Совет**

Графический компонент **JComboBox**, представленный в следующей главе, содер-

жит раскрывающийся список вариантов и при его созда- нии нужно также указывать тип данных.

**129**

**Новинка**

Пакет **java.time** был пред- ставлен в версии Java 8, для того чтобы облегчить работу с датой и временем.

**7. Импортирование функций**



**JAVA**

*DateTime.java*

**130**

**Работа с датой**

Пакет **java.time** содержит класс с именем **LocalDateTime**, в котором есть полезные методы для извлечения определенных полей, описы- вающих конкретные метки времени. Для работы с этим классом нуж- но либо импортировать конкретно его командой **import java.time. LocalDateTime;**, либо импортировать все классы данного пакета, ис- пользуя шаблон **import java.time.\* ;**.

При помощи метода **now()** можно создать новый объект **LocalDateTime**, содержащий поля, описывающие текущее значение даты и времени. Данные поля инициализируются текущим значением локального си- стемного времени.

Значения отдельных полей объекта **LocalDateTime** можно получить, используя соответствующие методы. Например, метод **getYear()** вы- зывает значение поля «Год». Точно так же значения всех полей можно и изменять, применяя соответствующие методы объекта **LocalDateTime** и указывая значение для установки. Например, новое значение года можно задать, указав его в качестве аргумента методу **withYear()**.

1. Создайте новую программу, импортирующую все методы класса

### java.time.LocalDateTime.

### import java.time.LocalDateTime ;

1. Добавьте класс с именем **DateTime**, содержащий стандартный ме- тод **main**.

### class DateTime

### {

### public static void main ( String [] args ) { }

### }

1. Внутри фигурных скобок главного метода добавьте оператор для создания объекта **LocalDateTime**, который будет содержать теку- щее значение времени.

### LocalDateTime date = LocalDateTime.now() ;

1. Выведите текущее значение даты и времени.

**131**

### System.out.println( "\nСейчас " + date ) ;

1. Увеличьте значение года и выведите измененное значение текущего времени.

### date = date.withYear( 2018 ) ;

### System.out.println( "\nТеперь текущее время " + date ) ;

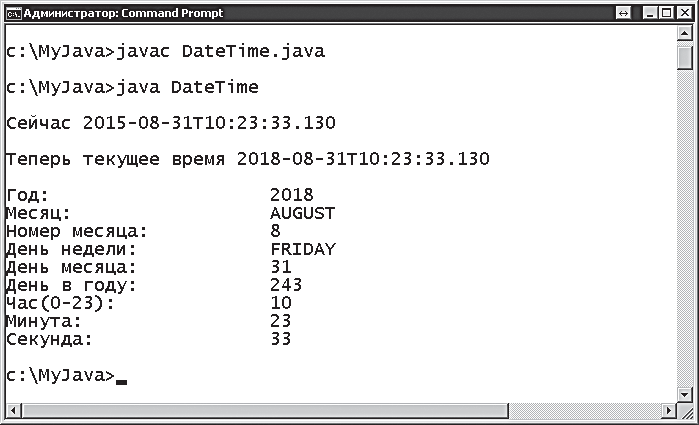


1. Выведите отдельные поля объекта **LocalDateTime**. **String fields = "\nГод:\t\t\t" + date.getYear() ; fields += "\nМесяц:\t\t\t" + date.getMonth() ;**

### fields += "\nНомер месяца:\t\t" + date.getMonthValue() ; fields += "\nДень недели:\t\t" + date.getDayOfWeek() ; fields += "\nДень месяца:\t\t" + date.getDayOfMonth() ; fields += "\nДень в году:\t\t" + date.getDayOfYear(); fields += "\nЧас (0-23):\t\t" + date.getHour() ;

### fields += "\nМинута:\t\t\t" + date.getMinute() ; fields += "\nСекунда:\t\t" + date.getSecond() ; System.out.println( fields ) ;

1. Сохраните программу под именем *DateTime.java*, затем скомпили-

руйте и запустите.

**Совет**

Использование конкатена- ции строк в данном приме- ре является эффективным способом для вывода, по- скольку функция **println()** вызывается только один раз, вместо того чтобы вызывать ее много раз для вывода каждого отдельного поля.



**На заметку**

Вместо **LocalDateTime** вы можете использовать класс **ZonedDateTime**, если вам требуется поле временной зоны.

**Новинка**

Пакет **java.time.format** был введен в версии Java 8 для упрощения представления значения даты.

**132**

**7. Импортирование функций**



**JAVA**

*Formats.java*

**Форматирование чисел**

В Java содержится пакет с именем **java.text**, который предлагает по- лезные классы для форматирования чисел, в том числе значений валют. Пакет можно сделать доступным в любой программе, включив в начало файла *.java* оператор **import**. Это можно сделать, используя шаблон \*, включающий все классы пакета: **import java.text.\* ;**, либо указывая определенный класс по имени.

В пакете **java.text** содержится класс с именем **NumberFormat**, который содержит методы, используемые для форматирования числовых зна- чений при выводе: добавление разделителей, символов валют, а также знаков процента.

При создании объекта **NumberFormat** определяется тип его формати- рования — метод **getNumberInstance()** для разделителей, **getCurren- cyInstance()** — для символов валюты и **getPercentInstance()** — для знаков процента. Форматирование применяется указанием числового значения, которое нужно отформатировать, в качестве аргумента мето- ду **format()** объекта **NumberFormat**.

Для форматирования объектов, содержащих дату и время, использует- ся пакет **java.time.format**, в котором имеется класс **DateTimeFormat**. Объект **DateTimeFormatter** содержит шаблон форматирования, кото- рый указывается в качестве строкового аргумента методу **ofPattern()**. Шаблон форматирования включает буквы, определенные в документа- ции, а также символы для разделителей, которые выбираете вы сами. Например, шаблон **M/d/y** определяет месяц, день и год, разделенные слешем. Формат применяется указанием шаблона форматирования в качестве аргумента методу **format()** объекта **java.time**.

1. Создайте новую программу, импортирующую функциональность всех методов класса **NumberFormat** пакета **java.text**, а также всех методов класса **DateTimeFormatter** пакета **java.time.format**.

### import java.text.NumberFormat ;

### import java.time.format.DateTimeFormatter ;

1. Добавьте класс с именем **Formats**, содержащий стандартный метод

### main.

### class Formats

### {

### public static void main ( String [] args )

### {

### }

### }

1. Внутри фигурных скобок главного метода добавьте операторы для вывода числа с разделителями групп.

### NumberFormat nf = NumberFormat.getNumberInstance() ; System.out.println( "\nЧисло: " + nf.format(123456789) ) ;

1. Добавьте выражения для вывода числа с символом валюты.

### NumberFormat cf = NumberFormat.getCurrencyInstance() ; System.out.println( "\nВалюта: " + cf.format(1234.50f) ) ;

1. Добавьте выражения для вывода числа со знаком процента.

### NumberFormat pf = NumberFormat.getPercentInstance() System.out.println( "\nПроцент: " + pf.format( 0.75f ) ) ;

1. Добавьте оператор, создающий объект **LocalDateTime**, содержащий

значение текущего времени.

### java.time.LocalDateTime now = java.time.LocalDateTime.now() ;

1. Добавьте операторы для вывода форматированного значения

даты.

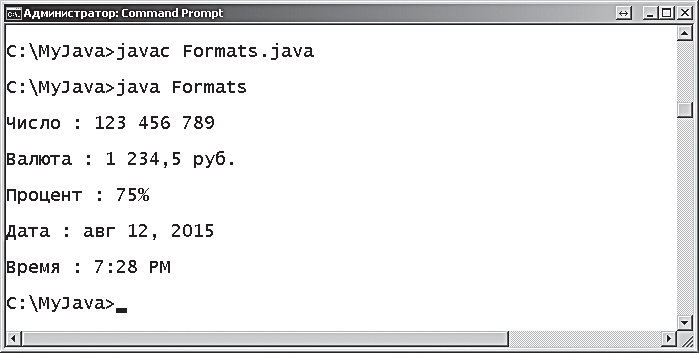
### DateTimeFormatter df = DateTimeFormatter.ofPattern( "MMM d, yyy" ) ;

### System.out.println( "\nДата: " + now.format( df ) ) ;

1. Добавьте операторы для вывода форматированного значения вре- мени.

### DateTimeFormatter tf = DateTimeFormatter.ofPattern( "h:m a" ) ;

### System.out.println( "\nВремя: " + now.format( tf ) ) ;

1. Сохраните программу под именем *Formats.java*, затем скомпили- руйте и запустите.



**Совет**

Операторы могут обращать- ся к классу, который не был импортирован, если указать полный путь к пакету, как показано здесь, в опера- торе, создающем объект **LocalDateTime**.



**Внимание**

Буквы, используемые

в шаблонах, чувствительны к регистру — для подробно- го описания используемых

**133**

шаблонов обратитесь к доку- ментации.

**Заключение**

**7. Импортирование функций**

* + Для того чтобы в программе использовать функциональность дру- гих классов, применяется оператор **import**, который ставится в на- чале программы.
  + Оператор **import** может импортировать как отдельные классы, определенные по имени, так и классы пакета при помощи символа шаблона **\***.
  + Для обработки процедур ввода и вывода существует пакет **java.io**.
  + Объект **File** можно использовать для доступа к файлам и каталогам.
  + Объект **InputStreamReader** декодирует введенные байты в симво- лы, а объект **BufferedReader** читает эти декодированные символы.
  + Для декодирования байтов текстового файла в символы использу- ется объект **FileReader**.
  + Объекты **FileWriter** и **BufferedWriter** можно использовать для создания и изменения текстового файлов.
  + Пакет **java.util** содержит утилиты для обработки наборов дан- ных, подобных массивам класса **Arrays**.

**134**

* + В пакете **java.util** содержится также класс **ArrayList**, у которого есть методы для обработки элементов последовательностей.
  + Объект **ArrayList** — это набор, который должен указывать тип элемента, содержащегося в списке, например **<String>**.
  + Лямбда-выражение — это анонимный метод, который определяет- ся сразу же на месте выполнения.
  + Пакет **java.time** содержит класс **LocalDateTime**, представляющий поля для компонентов даты и времени.
  + Пакет **java.text** содержит класс **NumberFormat**, который использу- ется для форматирования чисел и значений валют.
  + Пакет **java.time.format** содержит класс **DateTimeFormatter**, ко- торый определяет шаблоны для форматирования значений даты и времени.

**Построение интерфейсов**

**8**

***В этой главе демонстрируется, как использовать компоненты Java Swing для создания графических программных***

***интерфейсов.***

* **Создание окна**
* **Добавление кнопок**
* **Добавление меток**
* **Добавление текстовых полей**
* **Добавление элементов выбора**
* **Добавление переключателей**
* **Изменение внешнего вида интерфейса**
* **Размещение компонентов**
* **Заключение**

Буква **x** в названии **javax. swing** подразумевает **JAVA** e**X**tra (дополнение к Java).

**136**

**8. Построение интерфейсов**

Менеджеры шаблонов описаны более подробно в конце этой главы.

**Создание окна**

В программах на Java можно создавать графический пользовательский интерфейс (GUI), используя графический компонент библиотеки Java под названием Swing. Пакет **javax.swing** содержит классы, при помощи которых можно создать разнообразные компоненты, используя стиль операционной системы. Чтобы включить эту возможность, нужно до- бавить оператор импорта в программу: **import javax.swing.\*;**.



**На заметку**

Для создания графического пользовательского интерфейса нужно ор- ганизовать класс, к которому можно добавлять компоненты, исполь- зуемые при построении интерфейса. Проще всего это сделать, объявив подкласс класса **JFrame** при помощи ключевого слова **extends** — тем самым наследуя атрибуты и методы, которые позволят пользователю работать с окном: перемещать, изменять размер и закрывать.

Конструктор класса должен включать операторы, удовлетворяющие следующим минимальным требованиям:

* заголовок окна — указывается в качестве строкового аргумента на- следуемому методу **super()** класса **JFrame**;
* размер окна — указывается значение ширины и высоты в пикселях как аргументы для метода **setSize()**;
* действие при закрытии пользователем окна — определяется кон- стантой, являющейся аргументом для метода **setDefaultClose- Operation()**;
* параметр отображения окна — указывается в виде аргумента гра- фического типа для метода **setVisible()**.

Дополнительно в конструкторе можно добавлять компонент окна — так называемый контейнер **JPanel**, к которому будут добавляться бо- лее мелкие компоненты при использовании наследуемого метода **add()** класса **JFrame**.



**Совет**

Контейнер **JPanel** (или панель) по умолчанию использует менеджер шаблона **FlowLayout**, который располагает компоненты в колонках сле- ва направо, привязываясь к правой границе окна.

В шагах, представленных ниже, описывается создание базового окна, содержащего контейнер **JPanel** с менеджером шаблона **FlowLayout**. Это окно будет использоваться в дальнейших примерах книги для демон- страции добавления различных компонентов к контейнеру **JPanel**.

1. Создайте новую программу, в которую импортируются все компо- ненты Swing.

### import javax.swing.\* ;

1. Создайте в классе **JFrame** подкласс с именем **Window**, содержащий стандартный метод **main**.

### class Window extends JFrame

### {



**JAVA**

*Window.java*

### public static void main ( String[] args ) { }

### }

1. Перед методом **main** создайте контейнер **JPanel**. **JPanel pnl = new JPanel( ) ;**
2. Затем добавьте конструктор с указанием параметров окна и добав- лением объекта **JPanel** к классу **JFrame**.

### public Window()

### {

### super( "Окно Swing" ) ; setSize( 500 , 200 ) ;

### setDefaultCloseOperation( EXIT\_ON\_CLOSE ) ; add( pnl ) ;

### setVisible( true ) ;



**На заметку**

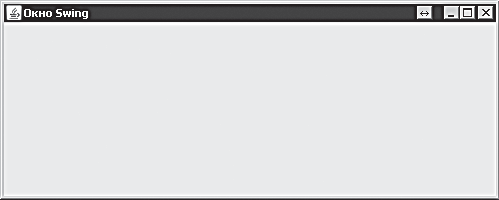
В данном примере значение **EXIT\_ON\_CLOSE** — это кон- станта, являющаяся членом класса **JFrame**. Она произво- дит выход из программы при закрытии окна.

### }

**137**

1. Создайте экземпляр класса **Window** с добавлением следующей стро- ки в главный метод.

### Window gui = new Window() ;

1. Сохраните программу под именем **Window.java**, затем скомпили- руйте и запустите — вы увидите, что появится базовое окно.



**Совет**

Обратите внимание, как ис- пользуется метод **add()** для добавления объекта **JPanel** на окно **JFrame**.

Подробнее о том, как созда- вать методы-обработчики событий, реагирующие на действия пользователя, например нажатие кнопки мыши, будут рассматривать- ся в следующей главе.

**8. Построение интерфейсов**

**Добавление кнопок**

На графический интерфейс можно добавлять кнопки, которые созда- ются при помощи класса **JButton**. Этими кнопками пользователь бу- дет взаимодействовать с программой, нажимая на них для выполнения определенного действия.

Объект **JButton** создается при помощи ключевого слова **new**, и его кон- структор принимает строковый аргумент, в котором указывается текст, отображаемый на кнопке.



**Совет**

На кнопках можно также располагать изображения. Для этого нужно сначала создать объект изображения — **ImageIcon**. Конструктору это- го объекта в качестве аргумента указывается имя файла изображения. Обычно файл изображения размещается рядом с программой, но это необязательно, поэтому аргумент может включать путь к изображе- нию за пределами каталога с программой.

Чтобы отобразить изображение на кнопке, нужно указать имя объекта **ImageIcon** в качестве аргумента конструктору **JButton** или указать два аргумента — строку и имя объекта **ImageIcon**. В этом случае на кнопке будет отображаться и текст, и изображение.

* 1. Отредактируйте копию файла *Window.java*, описанного ранее, за- менив имя класса **Window** в объявлении, конструкторе и экземпля- ре на **Buttons**.



**JAVA**

**138**

*Buttons.java*

### На заметку



* 1. Перед конструктором **Buttons()** создайте два объекта изображе- ний **ImageIcon**.

### ImageIcon tick = new ImageIcon( "tick.png" ) ; ImageIcon cross = new ImageIcon( "cross.png" ) ;

* 1. Затем создайте три объекта **JButton**: в первом из них будет ото-

бражаться текст, во втором — изображение, а в третьем — и текст, и изображение.

### JButton btn = new JButton( "Нажми меня" ) ; JButton tickBtn = new JButton( tick ) ;

### JButton crossBtn = new JButton( "СТОП" , cross ) ;

* 1. Внутри конструктора **Buttons()** добавьте три оператора для добав-

Для добавления компонен- тов на панель объект **JPanel** содержит метод **add()**.

ления компонентов **JButton** на контейнер **JPanel**. **pnl.add( btn ) ;**

### pnl.add( tickBtn ) ; pnl.add( crossBtn ) ;

* 1. Сохраните программу под именем *Buttons.java*, затем скомпили- руйте и запустите — вы увидите, как в окне появятся три кнопки.



Кнопки реагируют графически при нажатии на них, но не выполняют никаких действий до тех пор, пока в программе не будет задан метод- обработчик для реагирования на каждое событие нажатия.

В случае если вы намерены распространять программу в виде одиноч- ного Java-архива (JAR), то перед созданием объектов **ImageIcon** соот- ветствующие графические ресурсы должны быть загружены при по- мощи объекта **ClassLoader**.

Имя файла или его путь указывается методу **getResource()** объекта **ClassLoader**, и в результате будет возвращаться адрес URL, который можно будет использовать в качестве аргумента для конструктора **ImageIcon**. Для этого в пакете **java.net** существует полезный класс **URL**, которому можно все это присвоить.

* 1. Перед конструктором **Buttons()** создайте объект **ClassLoader**. **ClassLoader ldr = this.getClass().getClassLoader() ;**
  2. Загрузите адреса ресурсов (файлов изображений).

### java.net.URL tickURL = ldr.getResource( "tick.png" ) ; java.net.URL crossURL = ldr.getResource( "cross.png" ) ;

* 1. Отредактируйте конструкторы **ImageIcon()**, описанные на шаге 2,

чтобы использовать адреса URL.

### ImageIcon tick = new ImageIcon( tickURL ) ; ImageIcon cross = new ImageIcon( crossURL ) ;

* 1. Сохраните изменения, затем скомпилируйте и перезапустите про-

грамму на выполнение. Результат будет такой же, но теперь про- грамму можно будет распространять в файле JAR.



**Совет**

Подробнее о том, как соз- давать Java-архивы (JAR) см. в главе 10.

**139**



**Совет**

Обратите внимание, что для ссылки на объект класса ис- пользуется метод **getClass()** и ключевое слово **this**.

**Добавление меток**

**8. Построение интерфейсов**

На графический интерфейс можно добавлять элемент «метка» при по- мощи класса **JLabel**. Данный элемент используется для отображения неизменяемого пользователем текста или изображения или и того и другого.

Объект **JLabel** создается при помощи ключевого слова **new**, и его кон- структор принимает строковый аргумент, указывающий текст для ото- бражения на метке, или имя объекта **ImageIcon**, представляющего изо- бражение для отображения. Он может также принимать три аргумента, определяющие текст, изображение и величину горизонтальной при- вязки в виде константы объекта **JLabel**. Например, **JLabel( "text", img, JLabel.CENTER )** осуществляет привязку по центру.

В случае если объект **JLabel** содержит текст и изображение, то отно- сительное положение текста можно определять при помощи указания константы **JLabel** вкачестве аргумента методам **setVerticalPosition()** и **setHorizontalPosition()** объекта **JLabel**.

Существует дополнительный элемент ToolTip — всплывающая под- сказка, которая появляется при наведении курсора мыши на объект. Ее можно создавать при помощи метода **setToolTipText()**, указав ему в качестве аргумента строку текста с подсказкой.

**140**

* + 1. Отредактируйте копию файла *Window.java*, заменив имя класса



**JAVA**

**Window** в объявлении, конструкторе и экземпляре на **Labels**.

*Labels.java*

* + 1. Перед конструктором **Labels()** создайте объект **ImageIcon** для изо- бражения.

### ImageIcon duke = new ImageIcon( "duke.png" ) ;

* + 1. Затем создайте три объекта **JLabel**: первый для вывода изображе- ния, второй для вывода текста, а третий — и для того, и для дру- гого.

### JLabel lbl1 = new JLabel( duke ) ;

### JLabel lbl2 = new JLabel( "Дюк — талисман технологии Java." ) ; JLabel lbl3 = new JLabel( "Дюк" , duke , JLabel.CENTER ) ;

* + 1. Внутри конструктора **Labels()** добавьте следующий оператор, соз-

дающий подсказку для первой метки.

### lbl1.setToolTipText( "Дюк — талисман Java" ) ;

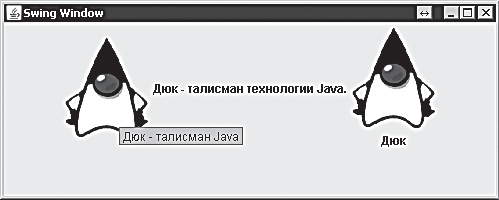
* + 1. После третьей метки напишите два оператора для привязки текста по центру и внизу.

### lbl3.setHorizontalTextPosition( JLabel.CENTER ) ; lbl3.setVerticalTextPosition( JLabel.BOTTOM ) ;

* + 1. Теперь допишите три оператора для добавления компонентов

### JLabel на контейнер JPanel. pnl.add( lbl1 ) ; pnl.add( lbl2 ) ; pnl.add( lbl3 ) ;

* + 1. Сохраните программу под именем *Labels.java*, затем скомпилируй- те и запустите, поместив указатель мыши над первой меткой.



В случае если вы намерены распространять программу в одиночном Java-архиве, перед созданием объекта **ImageIcon** графические ресурсы должны быть загружены при помощи объекта **ClassLoader**.

Методу **getResource()** объекта **ClassLoader** нужно указать имя файла или путь к нему, а метод вернет адрес URL, который можно использо- вать в качестве аргумента для конструктора **ImageIcon**.

* + 1. Перед конструктором **Labels()** создайте объект **ClassLoader**. **ClassLoader ldr = this.getClass().getClassLoader() ;**
    2. Отредактируйте конструктор **ImageIcon()**, представленный на шаге 2, для того чтобы загрузить адрес URL файла ресурса, исполь- зуя объект **ClassLoader**.

### ImageIcon duke =

### new ImageIcon( ldr.getResource( "duke.png" ) ) ;

* + 1. Сохраните изменения, затем заново скомпилируйте и запустите программу. Теперь ее можно распространять в файле JAR.



**На заметку**

Константы для привязки

к компонентам **JLabel** вклю- чают **LEFT**, **CENTER**, **RIGHT**, **TOP** и **BOTTOM**.



**141**

**Совет**

Подробнее о создании Java- архива (JAR) см. в главе 10.

В тех случаях, когда вам нуж- но скрыть вводимые поль- зователем символы, вместо класса **JTextField** исполь- зуйте класс **JPasswordField**.



**Совет**

**142**

**8. Построение интерфейсов**

**Добавление текстовых полей**

В библиотеке Swing существует класс **JTextField**, который создает компонент графического интерфейса, представляющий собой одно- строчное текстовое поле. Данный компонент позволяет отобразить на экране редактируемый текст, с помощью которого пользователь может взаимодействовать с программой.

Объект **JTextField** создается при помощи ключевого слова **new**, и его конструктор может принимать строковый аргумент, задающий текст по умолчанию, который будет отображаться в этом поле. В этом случае размер компонента будет подогнан в соответствии с длиной указанной строки. В качестве альтернативного варианта аргументом может быть числовое значение, определяющее размер текстового поля. Конструк- тор может также принимать и оба аргумента одновременно — для тек- ста по умолчанию и для размера текстового поля.

При помощи класса **JTextArea** можно создавать многострочные тек- стовые поля. Конструктор данного класса принимает два числовых аргумента, определяющих число строк и ширину поля. Альтернатив- ным вариантом является указание трех аргументов, задающих текст по умолчанию, а также число строк и ширину. При помощи методов **setLineWrap()** и **setWrapStyleWord()** объекта **JTextArea** можно управ- лять переносом слов, подгоняя тем самым ширину вводимого текста под размер поля.

Если текст, введенный в компонент **JTextArea**, превышает его первона- чальный размер, компонент будет растягиваться. Можно задать ком- поненту фиксированный размер с возможностью скроллинга. Для это- го его нужно поместить в контейнер **JScrollPane**. Это можно сделать при помощи ключевого слова **new** с использованием имени объекта **JTextArea** как аргумента.

По умолчанию полоса прокрутки (**ScrollBar**) будет появляться, когда поле содержит текст, превышающий его размер. Но можно сделать так, чтобы полоса прокрутки включалась постоянно. Для этого нужно ука- зать константу контейнера **JScrollPane** в качестве аргумента методам **setVerticalScrollBarPolicy()** или **setHorizontalScrollBarPolicy()**. Например, для того, чтобы всегда отображать вертикальную полосу прокрутки, используйте константу **JScrollPane.VERTICAL\_SCROLLBAR\_ ALWAYS** в качестве аргумента.

1. Отредактируйте копию файла *Window.java*, заменив имя класса

**Window** в объявлении, конструкторе и экземпляре на **TextFields**.

1. Перед конструктором **TextFields()** создайте два объекта

### JTextField.

### JTextField txt1 = new JTextField( 38 ) ;

### JTextField txt2 = new JTextField( "Текст по умолчанию" , 38 ) ;

1. Создайте объект **JTextArea** высотой в пять строк.

### JTextArea txtArea = new JTextArea( 5 , 37 ) ;

1. Добавьте объект **JScrollPane**, который будет содержать объект

**JTextArea**, созданный на предыдущем шаге.

### JScrollPane pane = new JScrollPane( txtArea ) ;

1. В методе конструктора **TextFields()** добавьте операторы для вклю- чения опции переноса слов.

### txtArea.setLineWrap( true ) ; txtArea.setWrapStyleWord( true ) ;

1. Добавьте оператор для постоянного отображения вертикальной

полосы прокрутки для объекта **JTextArea**. **pane.setVerticalScrollBarPolicy**

### ( JScrollPane.VERTICAL\_SCROLLBAR\_ALWAYS ) ;

1. Допишите два оператора для добавления компонентов **JTextField**

на контейнер **JPanel**. **pnl.add( txt1 ) ; pnl.add( txt2 ) ;**

1. Вставьте еще один оператор для добавления контейнера **JScroll- Pane** (содержащего поле **JTextArea**) в контейнер **JPanel**.

### pnl.add( pane ) ;

1. Сохраните программу под именем *TextFields.java*, затем скомпили- руйте и запустите, вводя текст в текстовые поля.



**JAVA**

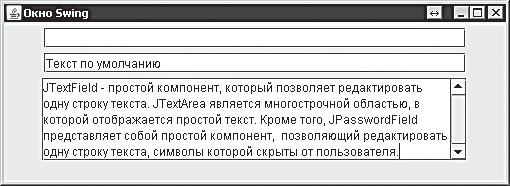
*TextFields.java*

### Внимание



Компонент **JTextArea** не имеет возможности про- крутки до тех пор, пока его не поместить внутрь компо- нента **JScrollPane**.

**143**



**Добавление элементов выбора**

**8. Построение интерфейсов**

В библиотеке Swing существует класс **JCheckBox**, при помощи которого создается компонент «флажок» (**CheckBox**). Его можно добавлять в гра- фический интерфейс, и этот компонент дает возможность выбирать либо отменять выбор отдельного элемента. Объект **JCheckBox** созда- ется при помощи ключевого слова **new**, и его конструктор принимает строковый аргумент, определяющий отображаемый рядом с флажком текст. Можно также указывать и второй аргумент, имеющий логиче- ское значение **true** — он делает флажок выбранным по умолчанию.

Класс **JComboBox** предлагает возможность создания выпадающего спи- ска элементов, из которого пользователь может выбрать единственный. Данный объект также создается при помощи ключевого слова **new**, а его конструктор принимает имя строкового массива в качестве аргумента. Каждый элемент этого массива соответствует элементу выбора выпа- дающего списка. Аналогичный список предлагается создать при помо- щи класса **JList**, который создает список фиксированного размера, и из него пользователь может выбирать один или более элементов. Объект **JList** создается при помощи ключевого слова **new**, а его конструктор принимает в качестве аргумента массив. Объекты **JList** и **JComboBox** являются наборами, поэтому они в своем объявлении должны содер- жать суффикс **<String>**, определяющий общий тип данных, который может содержать этот набор.

**144**

* 1. Отредактируйте копию файла *Window.java*, заменив имя класса



**JAVA**

**Window** в объявлении, конструкторе и экземпляре на **Items**.

*Items.java*

* 1. Перед конструктором **Items()** создайте строковый массив с эле- ментами, которые будет выбирать пользователь.

### String[] toppings =

### { "Пеперони" , "Грибная" , "С ветчиной" , "Томатная" } ;

* 1. Теперь создайте четыре объекта **JCheckBox**, каждый из которых бу- дет представлять элемент массива для выбора. Для одного из них установите значение выбора по умолчанию.

### JCheckBox chk1 = new JCheckBox( toppings[0] ) ; JCheckBox chk2 = new JCheckBox( toppings[1] , true ) ; JCheckBox chk3 = new JCheckBox( toppings[2] ) ; JCheckBox chk4 = new JCheckBox( toppings[3] ) ;

* 1. Добавьте строковый массив с элементами для выбора.

**145**

### String[] styles =

### { "В глубокой форме" , "Для гурманов" , "Тонкая" } ;

* 1. Создайте объект **JComboBox** для отображения элементов второго массива, предлагающихся к выбору.

### JComboBox<String> box1 =

### new JComboBox<String>( styles ) ;

* 1. Добавьте объект **JList** для отображения элементов первого масси- ва, предлагающихся к выбору.

### JList<String> lst1 = new JList<String>( toppings ) ;

* 1. В методе конструктора **Items()** добавьте операторы, помещающие каждый из компонентов **JCheckBox** в контейнер **JPanel**.

### pnl.add( chk1 ) ; pnl.add( chk2 ) ; pnl.add( chk3 ) ; pnl.add( chk4 ) ;

* 1. Добавьте операторы, которые устанавливают выбор элемента по

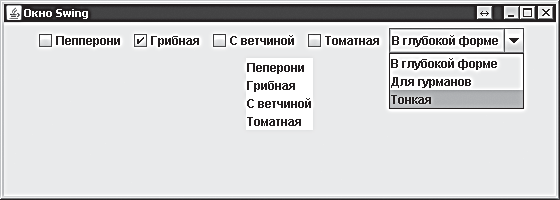
умолчанию, а затем помещают компонент **JComboBox** в контейнер

### JPanel. box1.setSelectedIndex( 0 ) ; pnl.add( box1 ) ;

* 1. Теперь добавьте оператор для помещения компонента **JList** на контейнер **JPanel**.

### pnl.add( lst1 ) ;

* 1. Сохраните программу под именем *Items.java*, затем скомпилируйте и запустите, выбирая элементы из списков.





**На заметку**

Если в компоненте **JComboBox** может быть выбран только один элемент, то в **JList** — несколько.



**Совет**

Подробнее о том, как созда- вать методы для обработки событий, реагирующие на действия пользователя, такие как выбор элементов списка, будет описано в сле- дующей главе.

**Добавление переключателей**

**8. Построение интерфейсов**

В библиотеке Swing существует класс **JRadioButton**, который исполь- зуется для создания такого компонента графического интерфейса, как

«переключатель». Он может быть использован для выбора пользовате- лем одного элемента из группы таких переключателей.

Объект **JRadioButton** создается при помощи ключевого слова **new**, а его конструктор принимает строковый аргумент, определяющий текст, отображаемый рядом с переключателем. Конструктор может также принимать и второй аргумент логического типа, имеющий значение **true**, который определяет, что соответствующий переключатель вы- бран по умолчанию.

Переключатели логически объединяются в группу при помощи объек- та **ButtonGroup**, и из этой группы может быть выбран одновременно только один из них. Переключатель добавляется в объект **ButtonGroup** при помощи его метода **add()** указанием имени данного переключателя в качестве аргумента этому методу.

* + 1. Отредактируйте копию файла **Window.java**, заменив имя класса



**JAVA**

**146**

**Window** в объявлении, конструкторе и экземпляре на **Radios**.

*Radios.java*



**На заметку**

Кнопки группируются

в объект **ButtonGroup** не физически, а логически.

* + 1. Перед конструктором **Radios()** создайте три объекта **JRadioButton**, один из которых сделайте выбранным по умолчанию.

### JRadioButton rad1 = new JRadioButton( "Красное" , true ) ; JRadioButton rad2 = new JRadioButton( "Розовое" ) ; JRadioButton rad3 = new JRadioButton( "Белое" ) ;

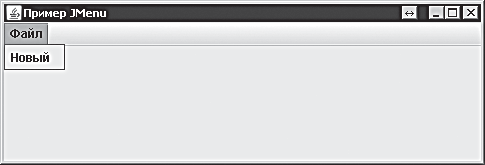
* + 1. Затем создайте объект **ButtonGroup**, при помощи которого сгруп-

пируйте переключатели.

### ButtonGroup wines = new ButtonGroup() ;

* + 1. В методе конструктора **Radios()** вставьте операторы для добавле- ния компонентов **JRadioButton** в группу **JButtonGroup**.

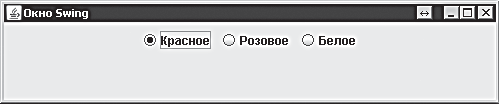
### wines.add( rad1 ) ; wines.add( rad2 ) ; wines.add( rad3 ) ;

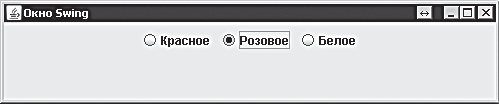
* + 1. Вставьте операторы для добавления компонентов **JRadioButton**

**147**

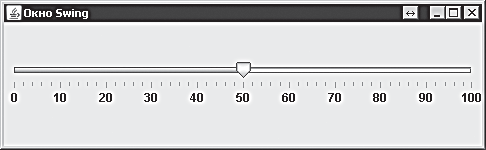
### в контейнер JPanel. pnl.add( rad1 ) ; pnl.add( rad2 ) ; pnl.add( rad3 ) ;

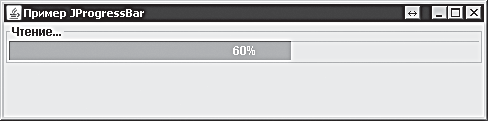
* + 1. Сохраните программу под именем *Radios.java*, затем скомпилируй- те и запустите, выбрав любую из кнопок.





Примеры выше демонстрировали наиболее общие компонен- ты библиотеки Swing — **JButton**, **JLabel**, **JTextField**, **JCheckBox**, **JComboBox**, **JList** и **JRadioButton**. В пакете **javax.swing** содержатся и другие, более специфичные компоненты, чье подробное описа- ние можно найти в документации по Java. Например, компоненты **JSlider**, **JProgressBar** и **JMenuBar**, представленные на рисунках ниже.







**Совет**

Подробнее о создании методов обработки событий, реагирующих на действия пользователя — в следую- щей главе.



**Совет**

Попробуйте, используя доку- ментацию, добавить компо- нент **JSlider** в программу Radios. Как использовать документацию — см. главу 5.

**Изменение внешнего вида интерфейса**

**8. Построение интерфейсов**

Пакет **java.awt** (Abstract Window Toolkit) содержит классы для «рисо- вания», которые вы можете использовать для «украшения» внешнего вида компонентов интерфейса. Этот пакет можно сделать доступным в программе, включив начальный оператор **import java.awt.\* ;**.

Внутри пакета **java.awt** содержится класс **Color**, в котором имеются константы, представляющие несколько базовых цветов, такие, как, на- пример, **Color.RED**. При помощи ключевого слова **new** можно создавать экземпляры класса **Color**, которые будут представлять пользователь- ские цвета. Конструктор может принимать три целочисленных аргу- мента, значения которых лежат в диапазоне от 0 до 255. Эти аргументы будут представлять компоненты RGB (красный, зеленый, синий), фор- мирующие пользовательский цвет.

Любой компонент интерфейса содержит методы **setBackground()** и **setForeground()**, которые в качестве своего аргумента принимают объект **Color**, чтобы раскрасить компонент указанным цветом.

Обратите внимание, что фон заднего плана (background) компонентов **JLabel** по умолчанию всегда прозрачен, поэтому, перед тем как изме- нять цвета компонентов, нужно установить значение непрозрачности в **true** при помощи метода **setOpaque()**.

**148**

В пакете **java.awt** существует также класс **Font**, который можно ис- пользовать для изменения шрифта отображаемого текста. Конструк- тор объекта **Font** может принимать три аргумента, которые определя- ют имя, стиль и размер шрифта:

* имя должно быть одним из трех платформонезависимых имен:

**Serif**, **SansSerif** или **Monospaced**;

* в качестве стиля должна выступать одна из этих трех констант:

**Font.PLAIN**, **Font.BOLD** или **Font.ITALIC**;

* размер должен быть целым числом, указывающим количество точек.

Для того чтобы изменить цвет шрифта любого из компонентов интер- фейса, существует метод **setFont()**, который принимает в качестве ар- гумента объект **Font**.

1. Отредактируйте копию файла *Window.java*, заменив имя класса

**Window** в объявлении, в конструкторе и в экземпляре на **Custom**.

1. Добавьте оператор в самое начало программы для импортирова- ния функциональности всех классов пакета **java.awt**.

### import java.awt.\* ;

### Перед конструктором Custom() создайте три объекта JLabel. JLabel lbl1 = new JLabel( "Пользовательский задний фон" ) ; JLabel lbl2 = new JLabel( "Пользовательский передний фон" ) ; JLabel lbl3 = new JLabel( "Пользовательский шрифт" ) ;

1. Создайте объект **Color**.

### Color customColor = new Color( 255 , 0 , 0 ) ;

1. Создайте объект **Font**.

### Font customFont = new Font( "Serif" , Font.PLAIN , 32 ) ;

1. В конструкторе **Custom()** добавьте операторы для изменения цвета заднего фона объекта **JLabel**, используя константу **Color**.

### lbl1.setOpaque( true ) ; lbl1.setBackground( Color.YELLOW ) ;

1. Добавьте оператор для изменения цвета переднего плана объекта

**JLabel**, используя пользовательский объект **Color**. **lbl2.setForeground( customColor ) ;**

1. Добавьте оператор для изменения текста на компоненте **JLabel**

с использованием пользовательского шрифта.

### lbl3.setFont( customFont ) ;

1. Добавьте все метки на контейнер **JPanel**.

### pnl.add( lbl1 ) ; pnl.add( lbl2 ) ; pnl.add( lbl3 ) ;

1. Сохраните программу под именем *Custom.java*, затем скомпилируй- те и запустите для просмотра результата изменения компонентов.



**JAVA**

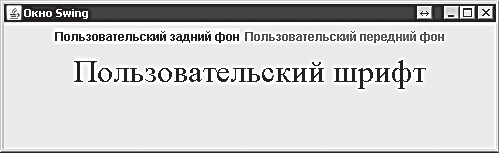
*Custom.java*



**149**

**Совет**

В данном случае пользова- тельский цвет эквивалентен значению **Color.RED**, по- скольку здесь указывается максимальное значение для красного, а зеленый и синий цвета полностью отсутст- вуют.



**Размещение компонентов**

**8. Построение интерфейсов**

В пакете **java.awt** содержится набор классов для менеджера компонов- ки, которые можно использовать для размещения компонентов в кон- тейнере различными способами.

Менеджер компоновки создается с использованием ключевого слова **new** и впоследствии может быть использован в качестве аргумента для конструктора **JPanel**, который будет указывать, чтобы панель исполь- зовала именно этот шаблон. При добавлении компонентов на панель они будут размещаться в соответствии с правилами указанного менед- жера шаблонов.

|  |  |
| --- | --- |
| **Менеджер шаблонов Правила размещения компонентов** | |
| **BorderLayout** | Вверху, внизу, справа, слева и в цен- тре (используется по умолчанию) |
| **BoxLayout** | В одну строку или столбец |
| **CardLayout** | Поочередно в указанной области |
| **FlowLayout** | В строке слева направо с возмож- ностью переноса (используется по умолчанию для **JPanel**) |
| **GridBagLayout** | В сетке ячеек (могут их растягивать) |
| **GridLayout** | В таблице по строкам и столбцам |
| **GroupLayout** | Горизонтально и вертикально |
| **SpringLayout** | С привязкой к границам компонентов |

Каждый из менеджеров компоновки подробнее опи- сывается в разделе докумен- тации **java.awt**.



**На заметку**

**150**

Объект верхнего уровня **JFrame** содержит контейнер, который раз- мещает компоненты, используя по умолчанию менеджер компоновки **BorderLayout**. На него можно помещать до пяти контейнеров **JPanel**, которые могут использовать свои менеджеры компоновки (по умолча- нию **FlowLayout**), представленные в таблице выше. Используя разно- образие менеджеров компоновки, можно добиться различных вариан- тов размещения компонентов.

Контейнер content pane может быть представлен объектом **java.awt. Container**, чей метод **add()** может указывать позицию и имя помеща- емого в данный контейнер компонента.

1. Отредактируйте копию файла *Window.java*, заменив объявление класса, конструктор и экземпляр **Window** на **Layout**, затем в начале программы добавьте оператор для импортирования функциональ- ности пакета **java.awt**.

### import java.awt.\* ;

1. Перед конструктором **Layout()** создайте объект **Container**, пред- ставляющий собой контейнер **JFrame**.

### Container contentPane = getContentPane() ;

1. Создайте еще один объект **JPanel**, используя менеджер компонов- ки **GridLayout** в виде сетки 2 х 2.

### JPanel grid = new JPanel( new GridLayout( 2 , 2 ) ) ;

1. В конструкторе **Layout()** вставьте операторы, добавляющие ком- поненты **JButton** на оба объекта **JPanel**.

### pnl.add( new JButton( "Да" ) ) ; pnl.add( new JButton( "Нет" ) ) ; pnl.add( new JButton( "Отмена" ) ) ; grid.add( new JButton( "1" ) ) ;

### grid.add( new JButton( "2" ) ) ;

### grid.add( new JButton( "3" ) ) ;

### grid.add( new JButton( "4" ) ) ;

1. Теперь вставьте операторы, добавляющие обе панели и кнопку в контейнер **contentPane**.

### contentPane.add( "North" , pnl ) ; contentPane.add( "Center" , grid ) ;

### contentPane.add( "West" , new JButton( "Запад" ) ) ;

1. Сохраните программу под именем *Layout.java*, затем скомпилируй- те и запустите, чтобы увидеть размещение компонентов.



**JAVA**

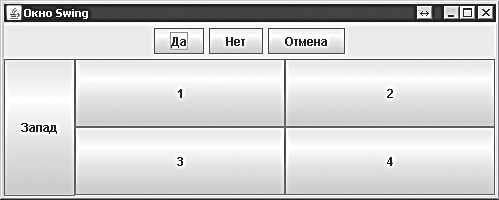
*Layout.java*

**151**



**Внимание**

Если **FlowLayout** использует размер объекта **JButton**, то другие менеджеры компо- новки растягивают ком- поненты, чтобы заполнить пространство компоновки.



**Заключение**

**8. Построение интерфейсов**

* + Для создания компонентов графического пользовательского интер- фейса используются классы библиотеки Java Swing, содержащиеся в пакете **javax.swing**.
  + Окно создается в виде контейнера верхнего уровня **JFrame**.
  + В конструкторе **JFrame** указывается заголовок окна, размер, опера- ция по умолчанию для закрытия и значение видимости.
  + Контейнер **JPanel** отображает более мелкие компоненты, исполь- зуя менеджер компоновки **FlowLayout**.
  + Конструктор **JButton** может определять текст и изображения, ко- торые должны быть отображены на компоненте «кнопка».
  + Изображения, используемые в программе, представляются объек- том **ImageIcon**.
  + Если программы должны быть распространены в виде отдельного Java-архива (JAR), то они должны использовать объект **ClassLoader** для указания исходного файла изображения.
  + Объект **JLabel** отображает неизменяемый текст и изображение.

**152**

* + Текст, который можно редактировать, отображается в полях

### JTextField и JTextArea.

* + Объект **JScrollPane** организует полосу прокрутки для поля

### JTextArea.

* + При помощи компонентов **JCheckBox**, **JComboBox** и **JList** можно отображать элементы для выбора.
  + Компоненты **JRadioButton** логически группируются при помощи компонента **JButtonGroup**.
  + Класс **Color** пакета **java.awt** представляет компоненты RGB зеле- ного цвета.
  + Для создания объектов, представляющих определенное имя, стиль и размер шрифта, используется класс **Font** пакета **java.awt**.
  + При помощи класса **Container** пакета **java.awt** несколько контей- неров можно добавлять в контейнер **JFrame**.
  + При создании контейнера **JPanel** можно указывать менеджер ком- поновки.

**Распознавание событий**

**9**

***В данной главе демонстрируется, как в Java-программах***

***создавать обработчики событий, которые реагируют на действия***

***пользователя.***

* + **«Прослушивание» событий**
  + **Генерация событий**
  + **Обработка событий кнопок**
  + **Обработка событий элементов**
  + **Реагирование на события клавиатуры**
  + **Ответ на события мыши**
  + **Вывод сообщений**
  + **Запрос пользовательского ввода**
  + **Проигрывание звука**
  + **Заключение**

**«Прослушивание» событий**

**9. Распознавание событий**

С программами, предлагающими графический интерфейс (GUI) поль- зователь может взаимодействовать, выполняя действия с мышью, кла- виатурой либо другими устройствами ввода. Действия пользователя вызывают «события», которые заставляют программу отвечать на них. Этот процесс называется «обработкой событий».

Для того чтобы программа могла распознавать события, вызванные действиями пользователя, в нее нужно добавить интерфейс «слушате- ля событий» — **EventListener** из пакета **java.awt.event**. Чтобы сде- лать его доступным в программе, нужно добавить в первоначальный оператор строку **import java.awt.event.\* ;**.

Для того чтобы включить интерфейс **EventListener** в объявление клас- са, нужно использовать ключевое слово **implements**. Например, объяв- ление класса для прослушивания событий кнопки мыши может выгля- деть следующим образом:

### class Click extends JFrame implements ActionListener { }

Документация по Java описывает множество интерфейсов **EventListener**, которые могут «слушать» различные события, но наиболее распростра- ненные представлены в таблице ниже вместе с кратким их описанием.

**154**

После ключевого слова **implements** можно добавлять несколько слушателей, раз- деляя их запятой.



**Совет**

|  |  |
| --- | --- |
| **Слушатель события Описание** | |
| **ActionListener** | Распознает события действия, которые происходят при нажатии или освобожде- нии кнопки |
| **ItemListener** | Распознает события элемента, которые происходят при выборе элемента или отме- не выбора |
| **KeyListener** | Распознает события клавиатуры, которые происходят, когда пользователь нажимает или отпускает клавишу |
| **MouseListener** | Распознает события кнопок мыши, которые происходят, когда пользователь нажимает или отпускает кнопку мыши либо когда ука- затель мыши входит в область компонента или покидает ее |
| **MouseMotionListener** | Распознает события движения указателя мыши, которые происходят, когда пользо- ватель двигает мышью |

**Генерация событий**

Для того чтобы интерфейс **EventListener** мог распознавать события, компоненты интерфейса должны их каким-либо образом генериро- вать. Если в программу добавлен какой-либо **EventListener**, как опи- сано выше, то в соответствующий компонент необходимо добавить генератор события.

Например, для того, чтобы программа реагировала на нажатие кнопки, в нее добавляется интерфейс **ActionListener** и должен быть вызван метод кнопки **addActionListener()**, в качестве аргумента которому указывается ключевое слово **this**. Таким образом генерируется собы- тие при нажатии на данную кнопку, и это событие будет распознавать- ся интерфейсом **ActionListener**.

Операторы, создающие кнопку, которая генерирует события, могут вы- глядеть следующим образом:

### JButton btn = new JButton( "Жми меня" ) ; btn.addActionListener( this ) ;

Когда пользователь нажимает кнопку, которая генерирует событие, ин- терфейс **ActionListener** распознает данное событие и ищет метод об- работчика событий в программе для выполнения ответных действий.

**155**

Каждый интерфейс **EventListener** имеет связанный с ним метод об- работчика событий, который вызывается, когда событие распознается. Например, когда нажимается кнопка, интерфейс **ActionListener** вы- зывает связанный метод с именем **actionPerformed()** и передает в ка- честве его аргумента объект **ActionEvent**.

Объект **ActionEvent** содержит информацию о событии и компоненте- источнике, вызвавшем его. Существует полезный метод **getSource()**, который идентифицирует объект, вызвавший событие. Его можно ис- пользовать для создания соответствующего ответного действия для компонента.

Метод обработчика события, создающий ответ на нажатие кнопки, мо- жет выглядеть приблизительно так:

### public void actionPerformed( ActionEvent event )

### {

### if ( event.getSource() == btn )

### {

### Операторы, которые будут выполняться по нажатию кнопки

### }

### }

**Обработка событий кнопок**

**9. Распознавание событий**

Компонент **JButton** в библиотеке Swing генерирует событие **ActionEvent**, которое распознается интерфейсом **ActionListener** и передается методу-обработчику **actionPerformed()**. У объекта **ActionEvent** существует метод **getSource()**, который идентифициру- ет исходный компонент, являющийся инициатором события, а также метод **getActionCommand()**, возвращающий строковое значение. Оно может быть либо текстовой меткой для кнопки, либо содержимым тек- стового поля компонента.

Одним из возможных ответов на событие нажатия кнопки может быть отключение компонента вызовом его метода **setEnabled()** с аргумен- том **false**, либо повторное его включение при помощи этого же метода **setEnabled()** с аргументом **true**.

1. Отредактируйте копию файла *Window.java*, заменив имя класса **Window** в объявлении, конструкторе и операторе экземпляра на **Actions**.



**JAVA**

*Actions.java*

**156**

1. В самое начало программы добавьте оператор для импортирова- ния функциональности всех классов пакета **java.awt.event**.

### import java.awt.event.\* ;

1. Отредактируйте объявление класса для добавления интерфейса

**ActionListener** в программу.

### class Actions extends JFrame implements ActionListener

1. Перед конструктором **Actions()** создайте две кнопки **JButton**, а также текстовое поле **JTextArea**.

### JButton btn1 = new JButton( "Кнопка 1" ) ; JButton btn2 = new JButton( "Кнопка 2" ) ; JTextArea txtArea = new JTextArea( 5 , 38 ) ;

1. Добавьте кнопки и текстовую область на контейнер **JPanel**. **pnl.add( btn1 ) ;**

### pnl.add( btn2 ) ; pnl.add( txtArea ) ;

1. Вставьте операторы для установки начального состояния двух ком- понентов.

### btn2.setEnabled( false ) ;

### txtArea.setText( "Кнопка 2 деактивирована" ) ;

1. В конструкторе **Actions()** вставьте операторы, которые генериру- ют событие **ActionEvent** при нажатии кнопок.

### btn1.addActionListener( this ) ; btn2.addActionListener( this ) ;

1. После метода конструктора добавьте метод-обработчик события

для интерфейса **ActionListener**, который будет отображать текст, идентифицирующий нажатую кнопку.

### public void actionPerformed( ActionEvent event )

### {

### txtArea.setText( event.getActionCommand()

### + " Нажата и деактивирована" ) ;



**На заметку**

Компоненты объявлены перед конструктором, поэто- му они доступны методу- обработчику событий.

### }

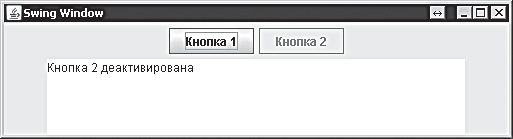
1. Вставьте операторы **if**, которые позволяют выполнить определен- ное ответное действие на нажатие каждой кнопки, в метод обра- ботчика событий.

### if ( event.getSource() == btn1 )

### { btn2.setEnabled( true ) ; btn1.setEnabled( false ) ; }

### if ( event.getSource() == btn2 )

### { btn1.setEnabled( true ) ; btn2.setEnabled( false ) ; }

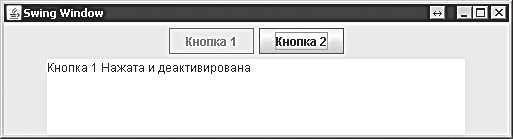
1. Сохраните программу под именем *Actions.java*, затем скомпили- руйте и запустите, пробуя нажимать кнопки.

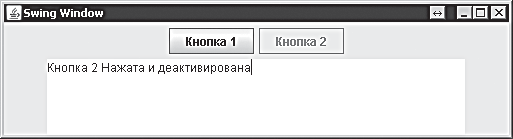


**Совет**

Иногда полезно деактиви- ровать компонент до того, как пользователь выполнит требуемое действие.

**157**





**Обработка событий элементов**

**9. Распознавание событий**

Изменения состояния компонентов **JRadioButton**, **JCheckBox** и **JCom- boBox** библиотеки Swing могут распознаваться интерфейсом **ItemLis- tener**, который, в свою очередь, передает событие **ItemEvent** методу-об- работчику **itemStateChanged()**. У объекта **ItemEvent** существует метод **getItemSelectable()**, который идентифицирует компонент, вызвавший событие, а также метод **getStateChange()**, возвращающий состояние компонента. Данный метод позволяет определить, что явилось причи- ной изменения статуса — выбор или отмена выбора элемента. Возвра- щенный статус может быть сравнен с константой **ItemEvent.SELECTED**.



**JAVA**

*States.java*

**158**

### Совет

Обратите внимание, как до- бавляется текст в текстовую область при помощи метода **append()**.

1. Отредактируйте копию файла *Window.java*, заменив имя класса **Window** в объявлении, конструкторе и операторе экземпляра на **States**. Затем добавьте в самое начало программы оператор для импортирования функциональности пакета **java.awt.event**.

### import java.awt.event.\* ;

1. Отредактируйте объявление класса, чтобы добавить интерфейс

**ItemListener** в программу.

### class States extends JFrame implements ItemListener

1. Перед конструктором **States()** создайте следующие компоненты.

### String[] styles =

### { "В глубокой форме" , "Для гурманов" , "Тонкая" } ; JComboBox<String> box = new JComboBox<String> ( styles ) ; JRadioButton rad1 = new JRadioButton( "Белое" ) ; JRadioButton rad2 = new JRadioButton( "Красное" ) ; ButtonGroup wines = new ButtonGroup() ;

### JCheckBox chk = new JCheckBox( "Пеперони" ) ; JTextArea txtArea = new JTextArea( 5 , 38 ) ;



1. В конструкторе **States()** вставьте операторы для группировки двух компонентов **JRadioButton**.

### wines.add( rad1 ) ; wines.add( rad2 ) ;

1. Вставьте операторы для добавления компонентов на контейнер

### JPanel.

### pnl.add( rad1 ) ; pnl.add( rad2 ) ; pnl.add( chk ) ;

### pnl.add( box ) ; pnl.add( txtArea ) ;

1. Добавьте операторы, чтобы выбираемые компоненты генерирова- ли событие **ItemEvent** при выборе или отмене выбора элемента.

### rad1.addItemListener( this ) ; rad2.addItemListener( this ) ; chk.addItemListener( this ) ; box.addItemListener( this ) ;

1. После метода конструктора добавьте метод-обработчик события для интерфейса **ItemListener**, который идентифицирует элементы, выбранные компонентами **JRadioButton**.

### public void itemStateChanged( ItemEvent event )

### {

### if ( event.getItemSelectable() == rad1 ) txtArea.setText( "Выбрано белое вино" ) ; if ( event.getItemSelectable() == rad2 ) txtArea.setText( "Выбрано красное вино" ) ;

### }

1. Добавьте оператор **if** в обработчик события для определения ста- туса компонента **JCheckBox**.

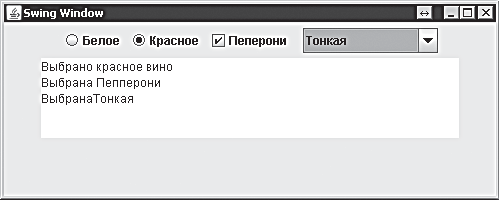
### if ( ( event.getItemSelectable() == chk ) &&

### ( event.getStateChange() == ItemEvent.SELECTED ) ) txtArea.append( "\nВыбрана Пеперони\n" ) ;

1. Добавьте оператор **if** в обработчик события для определения ста- туса компонента **JComboBox**.

### if ( ( event.getItemSelectable() == box ) &&

### ( event.getStateChange() == ItemEvent.SELECTED ) ) txtArea.append("Выбрана" + event.getItem().toString()) ;

1. Сохраните программу под именем *States.java*, затем скомпилируйте и запустите, пробуя выбрать различные элементы слева направо.



**Внимание**

Компонент **JComboBox** порож- дает два события **ItemEvents** при выборе элемента: пер- вое — это выбор элемента,

а второе — отмена выбора предыдущего выбранного элемента. Вот почему в ша- гах 8 и 9 нужно идентифици- ровать оба порождающих компонента, а также тип события **ItemEvent**.

**159**



**Совет**

Обратите внимание, как метод **getItem()** возвращает измененный элемент.

**Реагирование на события клавиатуры**

**9. Распознавание событий**

Компоненты библиотеки Swing, которые позволяют пользователю вво- дить текст, могут распознавать нажатие клавиш при помощи интер- фейса **KeyListener**, который передает событие **KeyEvent** следующим трем методам-обработчикам:

|  |  |
| --- | --- |
| **Обработчик события Описание** | |
| **keyPressed( KeyEvent )** | Вызывается при нажатии клавиши на клавиатуре |
| **keyTyped( KeyEvent )** | Вызывается после нажатия клавиши на клавиатуре |
| **keyReleased( KeyEvent )** | Вызывается при освобождении кла- виши на клавиатуре |

**160**



**JAVA**

*Keystrokes.java*

Если в программе реализуется интерфейс **KeyListener**, эти три мето- да должны быть обязательно объявлены (даже если используются не все три).

У объекта **KeyEvent** существует метод **getKeyChar()**, который воз- вращает символ, соответствующий нажатой клавише, а также метод **getKeyCode()**, возвращающий целочисленное значение, представляю- щее данную клавишу в формате ЮНИКОД. Дополнительный метод **getKeyText()** принимает в качестве аргумента код клавиши и возвра- щает описание этой клавиши.

1. Отредактируйте копию программы *Window.java*, заменив имя клас- са **Window** в объявлении, конструкторе и экземпляре на **Keystrokes**. Затем добавьте в начало импортирование функциональности паке- та **java.awt.event**.

### import java.awt.event.\* ;

1. Отредактируйте объявление класса, чтобы добавить в программу интерфейс **KeyListener**.

### class Keystrokes extends JFrame implements KeyListener

1. Передконструктором **Keystrokes()** создайтекомпонент **JTextField** и компонент **JTextArea**.

### JTextField field = new JTextField( 38 ) ; JTextArea txtArea = new JTextArea( 5 , 38 ) ;

1. Вставьте операторы для добавления этих двух компонентов на кон- тейнер **JPanel**.

### pnl.add( field ) ; pnl.add( txtArea ) ;

1. В конструкторе **Keystrokes()** добавьте оператор, который застав- ляет компонент **JTextField** генерировать события **KeyEvent**.

### field.addKeyListener( this ) ;

1. После метода конструктора добавьте обработчик событий, кото- рый распознает нажатие клавиш.

### public void keyPressed( KeyEvent event )

### {

### txtArea.setText( "Нажата клавиша" ) ;

### }



**Внимание**

1. Добавьте еще один обработчик события, который отображает сим- вол клавиши после того, как она нажата.

### public void keyTyped( KeyEvent event )

### {

### txtArea.append( "\nСимвол : "

### + event.getKeyChar() ) ;

### }

1. Добавьте третий обработчик события, который будет отображать код клавиши и текст, когда клавиша отпущена.

### public void keyReleased( KeyEvent event )

### {

Метод **getKeyCode()** воз- вращает код клавиши

в случае, если вызван обработчик **keyPressed()** или **keyReleased()**, но не **keyTyped()**.

**161**

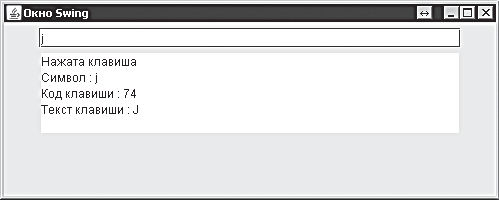
### int keyCode = event.getKeyCode() ; txtArea.append( "\nКод клавиши : "

### + event.getKeyCode() ) ; txtArea.append( "\nТекст клавиши : "

### + event.getKeyText( keyCode ) ) ;

### }



1. Сохраните программу под именем *Keystrokes.java*, затем скомпи- лируйте и запустите, пробуя набрать что-нибудь в текстовом поле сверху.

**Совет**

Запустите программу

и нажмите несимвольную клавишу, например **Пробел**, чтобы увидеть ее текстовое название.

**Ответ на события мыши**

**9. Распознавание событий**

Компоненты библиотеки Swing могут распознавать действия пользова- теля с мышью при помощи интерфейса **MouseListener**, который пере- дает событие **MouseEvent** следующим пяти методам-обработчикам.

|  |  |
| --- | --- |
| **Обработчик события Описание** | |
| **mousePressed( MouseEvent )** | Нажата кнопка мыши |
| **mouseReleased( MouseEvent )** | Освобождена кнопка мыши |
| **mouseClicked( MouseEvent )** | Выполнен щелчок кнопкой мыши |
| **mouseEntered( MouseEvent )** | Указатель входит в область |
| **mouseExited( MouseEvent )** | Указатель выходит из области |

Движение мыши можно распознавать при помощи интерфейса **Mouse- MotionListener**, который передает событие **MouseEvent** двум обработ- чикам:

**162**

|  |  |
| --- | --- |
| **Обработчик события Описание** | |
| **mouseMoved( MouseEvent )** | Перемещение указателя |
| **mouseDragged( MouseEvent )** | Перетаскивание (с нажатой кнопкой) |



**JAVA**

*Mouse.java*

Если в программе реализованы интерфейсы **MouseListener** или **Mouse- MotionListener**, то нужно объявлять все методы-обработчики (даже если они не используются).

У объекта **MouseEvent**, который передается интерфейсу **MouseMotion- Listener**, существуют методы **getX()** и **getY()**, которые возвращают текущие координаты указателя мыши относительно графического ком- понента, сгенерировавшего событие.

1. Отредактируйте копию файла *Window.java*, заменив имя класса **Window** в объявлении, конструкторе и операторе экземпляра на **Mouse**. Затем добавьте начальный оператор для импортирования функциональности пакета **java.awt.event**.

### import java.awt.event.\* ;

1. Отредактируйте объявление класса, чтобы добавить интерфейсы

**MouseListener** и **MouseMotionListener** в программу.

### class Mouse extends JFrame

### implements MouseListener , MouseMotionListener

1. Перед конструктором **Mouse()** создайте компонент **JTextArea**, а также две целочисленные переменные для хранения координат указателя.

### JTextArea txtArea = new JTextArea( 8 , 38 ) ; int x , y ;

1. В конструкторе **Mouse()** вставьте операторы для добавления ком-

понентов **JTextArea** на контейнер **JPanel**, а также для генерации событий **MouseEvent**.

### pnl.add( txtArea ) ; txtArea.addMouseMotionListener( this ) ; txtArea.addMouseListener( this ) ;

1. После метода конструктора добавьте два обработчика событий для

интерфейса **MouseMotionListener**.

### public void mouseMoved( MouseEvent event )

### { x = event.getX() ; y = event.getY() ; } public void mouseDragged( MouseEvent event ) { }

1. Добавьте пять обработчиков событий для интерфейса **MouseLis-**

### tener.

### public void mouseEntered( MouseEvent event )

### { txtArea.setText( "\nНажата кнопка мыши" ) ; } public void mousePressed( MouseEvent event )

### { txtArea.append( "\nКнопка нажата, когда указатель в позиции X: " +x+ "Y: " +y ) ; } public void mouseReleased( MouseEvent event )

### { txtArea.append( "\nКнопка мыши отпущена" ) ; } public void mouseClicked(MouseEvent event ) { } public void mouseExited(MouseEvent event ) { }

1. Сохраните программу под именем *Mouse.java*, затем скомпилируйте

и запустите, щелкнув указателем мыши по компоненту **JTextArea**.

### Совет

При помощи за- мены изображений

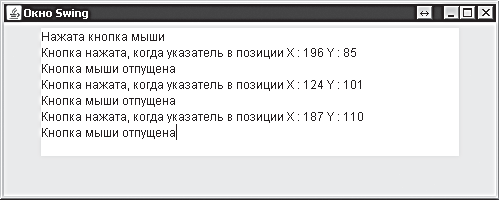


в методах-обработчиках

### mouseEntered()

и **mouseExited()** можно до- стичь эффекта перекатыва- ния (ролловера).

**163**



**9. Распознавание событий**



**JAVA**

*Messages.java*

**164**

**Вывод сообщений**

В библиотеке Swing для создания стандартного диалогового окна суще- ствует класс **JOptionPane**. Метод **showMessageDialog()** данного класса отображает сообщение пользователю; в сообщении размещается раз- личная информация, предупреждение или описание ошибок. Отобра- жаемое сообщение центрируется относительно родительского окна.

Метод **showMessageDialog()** может принимать четыре аргумента:

* родительский объект — обычно на него ссылается ключевое слово

### this;

* строковое сообщение для отображения;
* строковый заголовок диалога;
* одна из констант: **INFORMATION\_MESSAGE**, **WARNING\_MESSAGE** или

### ERROR\_MESSAGE.

В зависимости от того, какая указана константа, в диалоговом окне бу- дет отображаться соответствующая иконка.

1. Отредактируйте копию файла *Window.java*, заменив имя класса **Window** в объявлении, конструкторе и операторе экземпляра на **Messages**.
2. Добавьте начальный оператор для импортирования функциональ- ности пакета **java.awt.event**.

### import java.awt.event.\* ;

1. Отредактируйте объявление класса, чтобы добавить интерфейс

**ActionListener** в программу.

### class Messages extends JFrame implements ActionListener

1. Перед конструктором **Messages()** создайте три компонента

**JButton**. **JButton btn1= new JButton( "Показать информационное сообщение" ) ; JButton btn2= new JButton( "Показать предупреждение" ) ; JButton btn3= new JButton( "Показать сообщение об ошибке" ) ;**

1. Вставьте операторы для добавления компонентов кнопок на кон- тейнер **JPanel**.

### pnl.add( btn1 ) ; pnl.add( btn2 ) ; pnl.add( btn3 ) ;

1. В конструкторе **Messages()** вставьте операторы, генерирующие со- бытия **ActionEvent** для каждой кнопки.

### btn1.addActionListener( this ) ; btn2.addActionListener( this ) ; btn3.addActionListener( this ) ;

1. После метода конструктора добавьте метод-обработчик событий

для интерфейса **ActionListener**.

### public void actionPerformed( ActionEvent event ) { }

1. Внутри фигурных скобок метода-обработчика вставьте условные операторы, отображающие диалог при нажатии клавиш.

### if ( event.getSource() == btn1 ) JOptionPane.showMessageDialog( this , "Информация..." , "Диалоговое сообщение", JOptionPane.INFORMATION\_MESSAGE ) ; if ( event.getSource() == btn2 ) JOptionPane.showMessageDialog( this , "Предупреждение..." , "Диалоговое сообщение" , JOptionPane.WARNING\_MESSAGE ) ;

### if ( event.getSource() == btn3 ) JOptionPane.showMessageDialog( this , "Ошибка..." , "Диалоговое сообщение" , JOptionPane.ERROR\_MESSAGE ) ;

1. Сохраните программу под именем *Messages.java*, затем скомпили-

руйте и запустите, щелкнув указателем по каждой из кнопок.

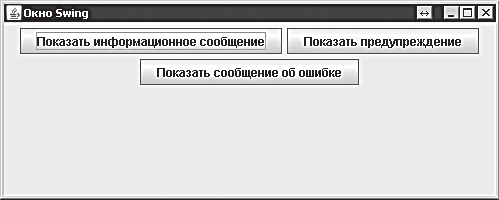
### Совет

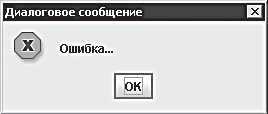
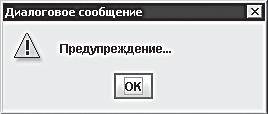
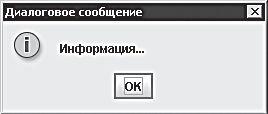
Вы можете указать только два аргумента — родитель- ское окно и сообщение, тем самым создав диалог,



в котором будут использо- ваться значок и заголовок по умолчанию.

**165**





**9. Распознавание событий**



**JAVA**

*Request.java*

**166**

**Запрос пользовательского ввода**

В библиотеке Swing существует класс **JOptionPane**, позволяющий за- прашивать информацию у пользователя при помощи двух методов, открывающих диалоговое окно — метод **showConfirmationDialog()**, запрашивающий подтверждение, а также метод **showInputDialog()**, который предлагает ввести пользователю строку.

Оба эти метода могут принимать четыре аргумента:

* родительский объект — ссылка на него определяется ключевым словом **this**;
* строка запроса для отображения;
* строка, определяющая заголовок диалогового окна;
* одна из констант **JOptionPane**, таких как **PLAIN\_MESSAGE** и опреде- ляющая кнопки для подтверждения, **YES\_NO\_CANCEL\_OPTION**.

Диалоговое окно будет возвращать строку ввода в случае с диалогом ввода или целое число в результате нажатия на кнопки подтвержде- ния — **0** для подтверждения, **1** для отказа или **2** для отмены.

1. Отредактируйте копию файла *Window.java*, заменив имя класса **Window** в объявлении, конструкторе и операторе экземпляра на **Request**. После этого добавьте начальный оператор для импорти- рования функциональности пакета **java.awt.event**.

### import java.awt.event.\* ;

1. Отредактируйте объявление класса, чтобы добавить в программу интерфейс **ActionListener**.

### class Request extends JFrame implements ActionListener

1. Перед конструктором **Request()** создайте компоненты **JTextField**

и два компонента **JButton**.

### JTextField field = new JTextField( 38 ) ;

### JButton btn1 = new JButton( "Запрос подтверждения" ) ; JButton btn2 = new JButton( "Запрос ввода" ) ;

1. Добавьте все компоненты на контейнер **JPanel**.

### pnl.add( field ) ; pnl.add( btn1 ) ; pnl.add( btn2 ) ;

1. В конструкторе **Request()** вставьте операторы, которые будут гене- рировать событие **ActionEvent** для каждой кнопки.

### btn1.addActionListener( this ) ; btn2.addActionListener( this ) ;

1. После метода конструктора добавьте метод обработчика событий для интерфейса **ActionListener**.

### public void actionPerformed( ActionEvent event ) { }



1. Внутри фигурных скобок обработчика событий вставьте условный оператор, для того чтобы организовать ответ на нажатие кнопок.

### if ( event.getSource() == btn1 )

### {

### int n = JOptionPane.showConfirmDialog( this , "Вы согласны?" , "Диалог подтверждения" , JOptionPane.YES\_NO\_CANCEL\_OPTION ) ;

### switch( n )

### {

**Совет**

Константа **OK\_CANCEL** пред- лагает две кнопки для под- тверждения: **OK** возвраща- ет 0, а **CANCEL** возвращает 2. Обратитесь к документации, чтобы узнать информацию о полном наборе констант.

### case 0 : field.setText( "Согласен" ) ; break ; case 1 : field.setText( "Не согласен" ) ; break ; case 2 : field.setText( "Отменено" ) ; break ;

### }

### }

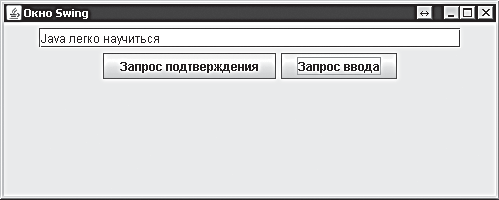
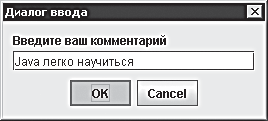
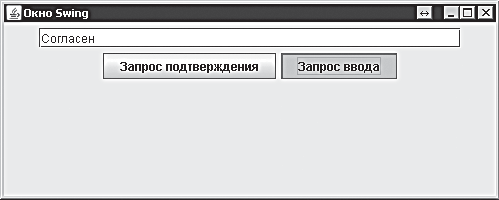
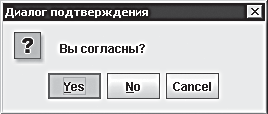
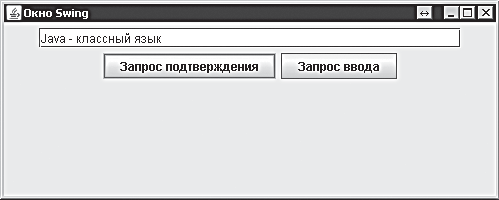
1. Вставьте условный оператор для обработки пользовательского ввода.

### if ( event.getSource() == btn2 )

### field.setText( JOptionPane.showInputDialog( this , "Введите ваш комментарий" , "Диалог ввода" , JOptionPane.PLAIN\_MESSAGE ) ) ;

**167**

Сохраните программу под именем *Request.java*, затем скомпили- руйте и запустите, выбрав указателем каждую из кнопок.



Подробнее о загрузке фай- лов изображений при помо- щи **ClassLoader** см. в главе 8.

**9. Распознавание событий**

**Воспроизведение звука**

Пакет **java.applet** содержит класс с именем **AudioClip**, в который встроены методы для проигрывания аудиофайлов форматов *.au*, *.aiff*,

*.wav*, а также *.mid*. Совместимость с компонентами библиотеки Swing поддерживается при помощи класса **JApplet**, который предлагает ме- тод **newAudioClip()**, возвращающий URL-адрес аудиофайла.



**На заметку**

Аудиофайл должен быть загружен в качестве ресурса в объект **AudioClip** при помощи метода **getResource()** объекта **ClassLoader** — точно так же, как и файл изображения.

В ответ на пользовательские действия воспроизведение аудиофайла может быть начато и прекращено при помощи методов **play()** и **stop()** объекта **AudioClip**. Кроме того, аудиофайл может быть воспроизведен непрерывно при помощи метода **loop()**.

* 1. Отредактируйте копию файла *Window.java*, заменив имя класса **Window** в объявлении, конструкторе и операторе экземпляра на **Sound**.



**JAVA**

*Sound.java*

**168**



*music.wav*

* 1. Добавьте начальный оператор для импортирования функциональ- ности всех классов пакета **java.awt.event**.

### import java.awt.event.\* ;

* 1. Отредактируйте объявление класса для добавления интерфейса

**ActionListener** в программу.

### class Sound extends JFrame implements ActionListener

* 1. Перед конструктором **Sound()** создайте объект **ClassLoader**. **ClassLoader ldr = this.getClass().getClassLoader() ;**
  2. Создайте объект **AudioClip** и загрузите файл звукового ресурса при помощи объекта **ClassLoader**.

### java.applet.AudioClip audio =

### JApplet.newAudioClip( ldr.getResource( "music.wav" ) ) ;

* 1. Создайте два компонента **JButton** для управления воспроизведе- нием звука.

### JButton playBtn = new JButton( "Играть" ) ; JButton stopBtn = new JButton( "Стоп" ) ;

* 1. Добавьте кнопки на контейнер **JPanel**. **pnl.add( playBtn ) ;**

**169**

### pnl.add( stopBtn ) ;

* 1. В конструкторе **Sound()** вставьте операторы, чтобы генерировать событие **ActionEvent** для каждой кнопки при их нажатии.

### playBtn.addActionListener( this ) ; stopBtn.addActionListener( this ) ;

* 1. После метода конструктора добавьте обработчик события для ин-

терфейса **ActionListener**.

### public void actionPerformed( ActionEvent event ) { }

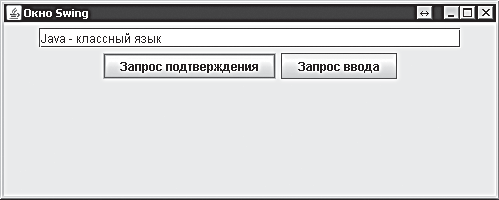
* 1. Внутри фигурных скобок обработчика событий вставьте оператор для проигрывания аудиофайла по нажатию кнопки воспроизве- дения.

### if ( event.getSource() == playBtn ) audio.play() ;

* 1. Вставьте оператор для остановки воспроизведения аудиофайла по нажатию кнопки остановки.

### if ( event.getSource() == stopBtn ) audio.stop() ;

* 1. Сохраните программу под именем *Sound.java*, затем скомпилируй- те и запустите, используя кнопки для контроля над воспроизведе- нием аудиофайла.



**Совет**

Можно воспроизводить не- сколько звуковых объектов одновременно.



**На заметку**

Каждый раз, когда вызыва- ется метод **play()**, воспро- изведение стартует с начала файла.

**Заключение**

**9. Распознавание событий**

* Для добавления одного или нескольких интерфейсов **EventListener** в объявление класса используется ключевое слово **implements**.
* Чтобы заставить компонент сгенерировать событие **ActionEvent** при его активации, используется метод компонента **addActionLis- tener()**, который принимает в качестве аргумента ключевое слово **this**.
* Интерфейс **ActionListener** передает сгенерированное событие **ActionEvent** в качестве аргумента методу-обработчику события **actionPerformed()**, который может реагировать на нажатие кноп- ки пользователем.
* Для того чтобы идентифицировать компонент, который сгенери- ровал событие, используется метод **getSource()**.
* Интерфейс **ItemListener** создает сгенерированное собы- тие **ItemEvent** в качестве аргумента методу-обработчику **itemStateChanged()**, который может реагировать на выбор элемен- та пользователем.
* Для идентификации компонента, сгенерировавшего событие, ис- пользуется метод **getItemSelectable()**.

**170**

* Интерфейс **KeyListener** передает сгенерированное событие **KeyEvent** в качестве аргумента трем обязательным методам-обра- ботчикам событий, которые могут реагировать на нажатие кнопки пользователем и возвращать символ, соответствующий нажатой кнопке.
* Интерфейс **MouseListener** передает сгенерированное событие **MouseEvent** в качестве аргумента пяти обязательным методам- обработчикам, которые могут реагировать на действия пользова- теля с мышью.
* Интерфейс **MouseMotionListener** передает сгенерированное собы- тие **MouseEvent** в качестве аргумента двум обязательным обработ- чикам событий, которые могут реагировать на движения указателя мыши.
* Метод **showMessageDialog()** класса **JOptionPane** создает диалого- вое окно, отображающее пользователю сообщение, а для запроса пользовательского ввода используются методы **showInputDialog()** и **showConfirmationDialog()**.
* Звуковые ресурсы могут представляться при помощи класса **AudioClip** пакета **java.applet** и проигрываются при помощи мето- да этого класса **play()**.

**Развертывание программ**

**10**

***В данной главе демонстрируется, как развертывать Java-программы при помощи настольных***

***приложений и встроенных в веб-страницы апплетов.***

* **Методы развертывания**
* **Распространение программ**
* **Построение архивов**
* **Развертывание приложений**
* **Подписывание jar-файлов**
* **Использование технологии Web Start**
* **Изготовление апплетов**
* **Преобразование веб-страниц**
* **Развертывание апплетов**
* **Заключение**

**Методы развертывания**

**10. Развертывание программ**

Технология Java предлагает два различных способа для развертывания программ:



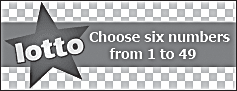
* + настольные приложения, которые могут быть запущены на любой из платформ, где установлено соответствующее окружение Java Runtime Environment;
  + апплеты для веб-страниц, которые выполняются на любом веб- браузере, где установлен соответствующий Java-плагин.

Принцип выбора метода развертывания обычно определяется назна- чением программ. Например, апплеты используются в легких програм- мах, которые должны быть интегрированы в содержимое веб-страниц, в то время как настольные приложения предпочтительнее использо- вать для более серьезных программ.

Как апплеты, так и настольные приложения могут быть созданы из- начально из общей программы, наподобие представленной ниже про- граммы *Lotto.java*.

Конструктор **Lotto()** генерирует простой интерфейс Swing, пред- ставляющий собой панель, содержащую одну метку, одно тексто- вое поле и одну кнопку. Данная панель помещена на окно размерами 260 × 200 пикселов. Компонент «метка» просто отображает описатель- ное графическое изображение:

**172**





**Внимание**

Программы в данной книге скомпилированы при помо- щи Java 8 — для корректно- го выполнения приложения требуется установка соответ- ствующего Java 8 JRE, а для выполнения апплетов — плагин Java 8.

*Lotto.png* — области с шахматным рисунком являются прозрачными.

Метод-обработчик для кнопки запускает алгоритм выбора последова- тельности из шести случайных чисел в диапазоне от 1 до 49, которые будут отображаться в компоненте «текстовое поле».

Программа *Lotto.java*, представленная здесь, используется на протяже- нии всей главы для создания, как приложения, так и апплета.

### import javax.swing.\* ; import java.awt.event.\* ;

### public class Lotto extends JFrame implements ActionListener

### {

### // Компоненты.

### ClassLoader ldr = this.getClass().getClassLoader() ; java.net.URL iconURL = ldr.getResource( "Lotto.png" ) ;

### ImageIcon icon = new ImageIcon( iconURL ) ; JLabel img = new JLabel( icon ) ;

### JTextField txt = new JTextField( "" , 18 ) ;

### JButton btn = new JButton( "Показать счастливые номера" ) ; JPanel pnl = new JPanel() ;

### // Конструктор. public Lotto()

### {



**JAVA**

*Lotto.java*

### super( "Приложение Lotto" ) ; setSize( 260 , 200 ) ; setDefaultCloseOperation( JFrame.EXIT\_ON\_CLOSE ) ; pnl.add( img ) ; pnl.add( txt ) ; pnl.add( btn ) ; btn.addActionListener( this ) ;

### add( pnl ) ; setVisible( true ) ;

### }

### // Обработчик событий.

### public void actionPerformed( ActionEvent event )

### {

### if ( event.getSource() == btn )

### {

**173**

### int[] nums = new int[50] ; String str = "" ;

### for ( int i = 1 ; i < 50 ; i++ ) { nums[i] = i ; } for ( int i = 1 ; i < 50 ; i++ )

### { int r = (int) ( 49 \* Math.random() ) + 1 ;

### int temp= nums[i] ; nums[i]= nums[r] ; nums[r]= temp ; } for ( int i = 1 ; i < 7 ; i++ )



### { str += " " + Integer.toString( nums[i] ) + " " ; } txt.setText( str ) ;

### }

### }

### // Точка входа.

### public static void main ( String[] args )

### { Lotto lotto = new Lotto() ; }

### }

**Совет**

Данный обработчик событий перемешивает целые числа в массиве, а затем первые шесть чисел заносит в стро- ковый массив.

Файл изображения должен быть именован в точности так: *Lotto.png* (а не *lotto.png*).

**174**

**10. Развертывание программ**

**Распространение программ**

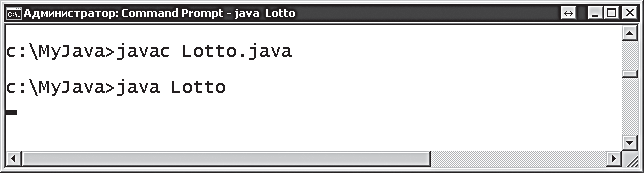
Объект **ClassLoader** в программе *Lotto.java* с предыдущей страницы ожидает, что графический файл *Lotto.png* размещен в том же каталоге, что и файл программы, поэтому, прежде чем попытаться скомпилиро- вать программу, нужно убедиться, что файлы размещены именно та- ким образом.



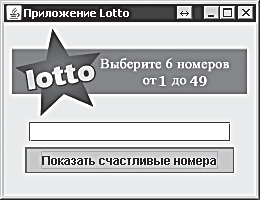
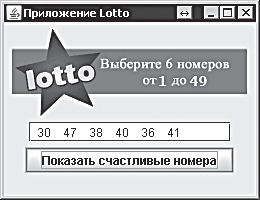
**Внимание**



Убедившись, что файлы расположены так, как это требуется, можно запускать компилятор **javac** для создания файла *Lotto.class*, а затем ин- терпретатор **java** для запуска программы.



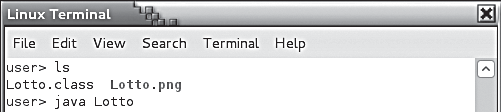
Программа **Lotto** открывает новое окно определенного размера, содер- жащее компоненты интерфейса Swing. Каждый раз, когда пользователь нажимает на кнопку, обработчик события отображает шесть новых случайных чисел в диапазоне от 1 до 49 в текстовом поле.

Программа **Lotto**, как и другие примеры данной книги, скомпилиро- вана для Java 8. Ее можно распространять для исполнения на других компьютерах, где установлено Java 8 Runtime Environment независимо от операционной системы.

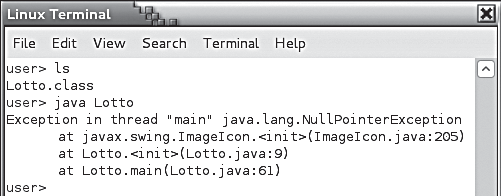


Например, на скриншотах ниже файлы *Lotto.class* и *Lotto.png* скопиро- ваны на рабочий стол компьютера, на котором установлена операци- онная система Linux и Java 8 Runtime Environment. Поэтому программа **Lotto** может быть запущена при помощи интерпретатора **java** точно так же, как и на исходной операционной системе Windows:





При распространении таким образом Java-программ существует, од- нако, опасность, что программа не выполнится, если файлы ресурсов станут недоступны — в данном случае отсутствие файла *Lotto.png* при- ведет к следующей ошибке:



**Совет**

Для распространения про- граммы не требуется вклю- чать файл исходных кодов

*.java*. Необходимы только файлы *.class*, а также ресурс- ные файлы.

**175**



**На заметку**

В больших программах может использоваться большое количество файлов ресурсов, корректное рас- положение которых может быть случайно нарушено пользователем. Решение, описанное ниже, заключа- ется в упаковке программы и всех ее ресурсов в один ис- полняемый архивный файл.

**Построение архивов**

**10. Развертывание программ**

Пакет JDK содержит утилиту **jar**, которая позволяет упаковывать фай- лы программ, а также все ресурсные файлы в один файловый Java-архив (JAR). Данная утилита сжимает все файлы, используя популярный формат ZIP, и создает файл с расширением *.jar*. JAR-архивы являются эффективным методом и помогают избавиться от случайного удаления ресурсных файлов. Интерпретатор **java** полностью поддерживает JAR- архивы, так что приложения и апплеты могут быть запущены без необ- ходимости извлекать отдельные файлы из архива. Инструмент **jar**, так же, как и интерпретатор **java** и компилятор **javac**, размещен в каталоге *bin*, и его можно запускать из командной строки для выполнения сле- дующих основных операций с архивами.

|  |  |
| --- | --- |
| **Синтаксис команды** | **Операция** |
| **jar cf jar-файл входные файлы** | Создание JAR-архива |
| **jar cfe jar-файл точка\_входа входные файлы** | Создание JAR-архива с ука- занием точки входа в прило- жение |
| **jar tf jar-файл** | Просмотр содержимого JAR-архива |
| **jar uf jar-файл** | Обновление содержимого JAR-архива |
| **jar ufm jar-файл файл\_атрибутов** | Обновление содержимого манифеста JAR-архива с добав- лением в него атрибутов |
| **jar xf jar-файл** | Извлечение содержимого JAR-архива |
| **jar xf jar-файл архивный\_файл** | Извлечение определенно- го файла из содержимого JAR-архива |



**Совет**

В случае с большими про- граммами для архивации множественных файлов можно использовать сим- вол шаблона \*. Например, команда **jar cf Program. jar \*.class** архивирует все файлы классов в текущем каталоге.

**176**

*Lotto.jar*

Выполните следующие шаги, чтобы создать JAR-архив для программы

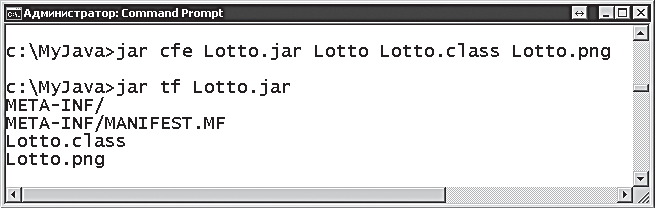
**Lotto**, описанной в начале главы.

1. Откройте командную строку (терминальное окно) и перейдите в каталог, в котором размещены файлы программы Lotto — *Lotto. class* и *Lotto.png*.



**JAR**

1. В командной строке наберите **jar cfe Lotto.jar Lotto Lotto. class Lotto.png**, затем нажмите клавишу **Enter** для создания архи- ва *Lotto.jar*.
2. Теперь наберите **jar tf Lotto.jar** для просмотра содержимого JAR-архива.



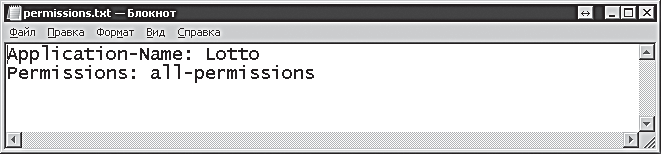
Обратите внимание, что инструмент **jar** автоматически создает ката- лог *META-INF* рядом с архивными файлами. Он содержит текстовый файл манифеста с именем *MANIFEST.MF*. В данный файл манифеста добавляется атрибут **permissions**, который определяет доступ к ресур- сам приложения.

1. Запустите простой текстовый редактор и наберите в нем следу- ющее:

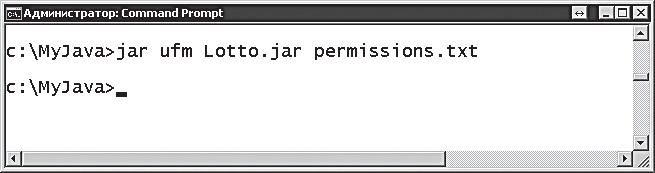
### Application-Name: Lotto Permissions: all-permissions

Затем сохраните файл под именем *permissions.txt* рядом с JAR-ар-

хивом.



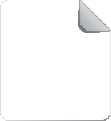
1. Наберите команду **jar ufm Lotto.jar permissions.txt**, затем на- жмите клавишу **Enter** для добавления атрибута **permissions** в ма- нифест.



Файл *permissions.txt* должен оканчиваться символом новой строки — нажмите клавишу **Enter** после послед- ней строки перед сохране- нием файла.



**Внимание**



*permissions.txt*

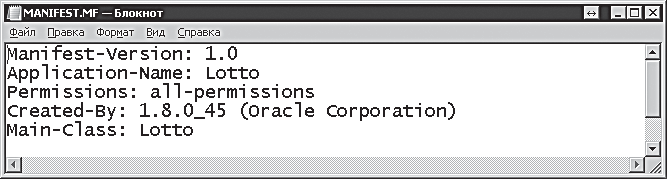


**Совет**

Извлеките копию каталога *META-INF*, используя команду **jar xf Lotto.jar META- INF**, чтобы изучить файл *MANIFEST.MF* — он должен выглядеть точно так же,

как показано на последнем рисунке.

**177**

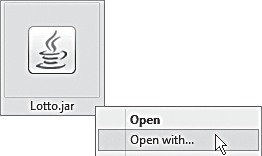


### Внимание



**10. Развертывание программ**

При запуске JAR-файлов расширение *.jar* необходимо указывать.

Установите для файлов JAR запускаемым по умолчанию приложением JRE.



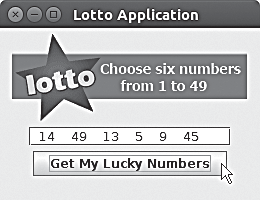
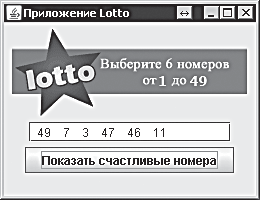
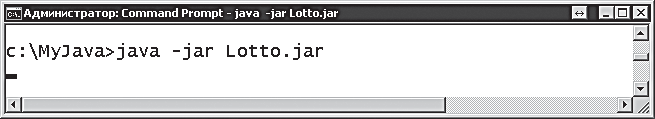
**Совет**

**178**

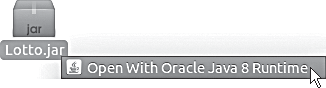
**Развертывание приложений**

Файлы JAR-архивов являются исполняемыми на любой системе, на ко- торой установлена соответствующая версия Java Runtime.

1. В командной строке перейдите в каталог, в котором расположен файл *Lotto.jar*, затем наберите команду **java -jar Lotto.jar** и на- жмите клавишу **Enter** для запуска приложения **Lotto**.

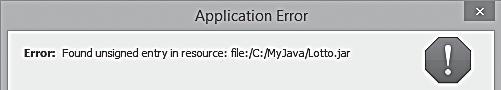


1. Альтернативным методом запуска является двойной щелчок мы- шью по файлу *Lotto.jar*, либо щелчок правой кнопкой мыши и вы- бор команды **Открыть с помощью** (Open with) и **Java Runtime**.



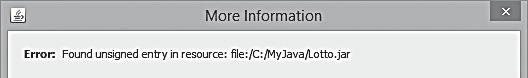


**Подписывание jar-файлов**

В случае если Java-программа предназначена для широкого распростра- нения, JAR-архивы подписывают цифровой подписью, для того чтобы пользователи, запускающие их, могли удостовериться, что программа выпущена доверенным источником. Попытка запустить неподписан- ное Java-приложение, распространяемое через Интернет при помощи технологии Java Web Start или через Java-апплет, встроенный в веб- страницу, будет выводить сообщение об ошибке:



**Внимание**



Сертификаты, предназначенные для подписи кода, выпускаются до- веренными центрами сертификации (Certificate Authority или CA), та- кими как VeriSign, DigiCert или GlobalSign. Сертификаты выпускаются после проверки центрами авторства разработчиков. Таким образом, пользователи могут быть уверены, что программа, подписанная сер- тификатом, получена из надежного источника. Центры сертификации обеспечивают разработчикам персональный файл, содержащий закры- тый ключ, с помощью которого разработчик подписывает JAR-файлы, а в пакете JDK для этих целей существует утилита **jarsigner**.

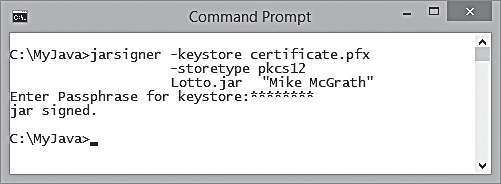
* 1. Отройте командную строку (терминальное окно) и перейдите в ка- талог, где размещен файл *Lotto.jar*.
  2. Введите команду **jarsigner**, которая идентифицирует файл ключа, тип хранения, файл JAR-архива и имя разработчика; затем нажми- те клавишу **Enter**.

### jarsigner -keystore certificate.pfx

### -storetype pkcs12 Lotto.jar "Mike McGrath"

* 1. Введите парольную фразу, которую вы отправите в центр серти-

фикации для своего сертификата. Дальше вы увидите ответ, под- тверждающий подпись JAR-файла.



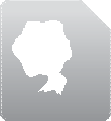
В пакете JDK существует

утилита **keytool**, при помо- щи которой можно соз- дать самоподписываемые

сертификаты, но они больше не поддерживаются для

JAR-файлов.

**179**



*certificate.pfx*

Данный файл не включен в архив файлов примеров

к этой книге, поскольку вам нужен будет собственный сертификат.



**Совет**

Проверить, подписан ли JAR- файл, вы можете с помощью команды **jarsigner -verify Lotto.jar**.



**JNLP**

*Lotto.jnlp*

**180**

**10. Развертывание программ**

Помните, что в файле JNLP нужно установить параметр безопасности

**all-permissions**, чтобы при- вести файл в соответствие

с манифестом.

**Использование технологии Web Start**

Технология Java Web Start позволяет пользователям запускать Java- приложения простым нажатием на гиперссылку, находящуюся на веб-странице. Ссылка указывает на файл JNLP (протокол Java Network Launching Protocol), который предоставляет информацию о приложе- нии в формате XML. Выполните следующие шаги, чтобы запустить приложение **Lotto** при помощи ссылки с веб-страницы.

1. Откройте простой текстовый редактор, например Блокнот (Note- pad), и скопируйте следующее содержимое в формате XML для соз- дания файла протокола JNLP.

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

### <jnlp spec = "1.0+"codebase = "file:///C:/MyJava/" href = "Lotto.jnlp" >

### <information>

### <title>Lotto Application</title>

### <vendor>Java in easy steps</vendor>

### <homepage href = ["http://www.ineasysteps.com"](http://www.ineasysteps.com/) />

### <offline-allowed />

### </information><security><all-permissions/></security>

### <resources>

### <jar href = "Lotto.jar" />

### <j2se version = "1.6+"href = ["http://java.sun.com/products/autodl/j2se"](http://java.sun.com/products/autodl/j2se) />

### </resources>

### <application-desc main-class = "Lotto" />

### </jnlp>

1. Сохраните новый файл JNLP под именем *Lotto.jnlp* рядом с файлом

*Lotto.jar*.



1. В тот же самый каталог добавьте файл *Lotto.html* и включите в его содержимое следующую гиперссылку.

**Совет**

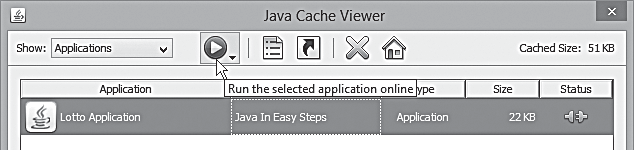
### <a href = "Lotto.jnlp" >Запустить приложение Lotto</a>

1. Откройте файл *Lotto.html* в вашем веб-браузере и щелкните по ссылке для того чтобы запустить приложение **Lotto**, используя тех- нологию Web Start.

Во время загрузки на короткое время появляется заставка Java Web Start, а потом загружается ваше приложение.

Для того чтобы задействовать технологию Web Start, необходимо снача- ла сконфигурировать веб-сервер таким образом, чтобы он поддерживал формат JNLP. Для этого нужно добавить в описание MIME-типа следую- щую строку: **application/x-java-jnlp-file JNLP**. Кроме того, нужно за- дать атрибут **codebase** с указанием расположения вашего приложения на сервере, например **codebase= "http://www.myserver/java-apps/"**. После загрузки необходимых файлов JNLP, JAR и HTML в это место, Java Web Start может загружать приложение по ссылке, как указывалось выше.

Во время первой загрузки приложения информация из JNLP-файла ав- томатически сохраняется в локальном кэше Java. Она может быть ис- пользована для последующей загрузки приложения, а также для созда- ния ярлыков на рабочем столе для запуска приложений.

1. Чтобы открыть кэш Java на компьютере под управлением операци- онной системы Windows, щелкните по иконке **Java** в окне панели управления, а затем по кнопке **View** (Показать) на вкладке **General** (Общие) в открывшемся диалоговом окне.
2. Выберите программу **Lotto** и щелкните по иконке запуска , для того чтобы открыть приложение.
3. Нажмите кнопку **Install shortcuts** (Установить ярлык) для добав- ления на рабочий стол ярлыка. После этого попробуйте запустить приложение, используя созданный ярлык.



**HTML**

*Lotto.html*



**Внимание**

В атрибуте **codebase** должен быть точно указан каталог

с файлами на сервере, иначе Web Start не сможет найти их и программа не запустится.

**181**



**На заметку**

Кэш Java можно открыть из командной строки как

в среде Windows, так и Linux с помощью команды **javaws**

### -viewer.



**182**

**10. Развертывание программ**

Объявления **init** в апплете должны включать ключевое слово **void** — апплет не мо- жет возвращать значение.

**Создание апплетов**

В качестве альтернативы Java-приложениям, которые выполняются при помощи среды Java Runtime Environment, установленной в системе, программу можно разрабатывать в виде встроенных в веб-страницы Java-апплетов, которые исполняются Java-плагином в браузере.

Java-апплеты отличаются от приложений Java двумя основными аспек- тами:

* апплетам не нужно отдельное окно — они располагаются внутри веб-страницы при помощи кода HTML;
* апплеты не содержат метод **main** — вместо него они используют ме- тод **init()**.

Чтобы понять разницу между апплетом и приложением, выполните следующие шаги, преобразуя приложение *Lotto.java* в апплет.

1. Измените имя класса с **Lotto** на **LottoApplet**, а затем в объявлении класса замените **JFrame** на **JApplet**.
2. Поменяйте конструктор на метод апплета **init()** путем замены

### public Lotto() на public void init().

1. Удалите все операторы, определяющие свойство окна в бывшем конструкторе, убрав следующие строки.

### super( "Приложение Lotto" ) ; setSize( 260 , 200 ) ;

### setDefaultCloseOperation( JFrame.EXIT\_ON\_CLOSE ) ; setVisible( true ) ;



**Внимание**

1. Добавьте возможность задания цвета фона апплета при помощи

шестнадцатеричного параметра, устанавливаемого в коде HTML.

### String bgStr = getParameter( "BgColor" ) ; int bgHex = Integer.parseInt( bgStr , 16 ) ;

### pnl.setBackground( new java.awt.Color( bgHex ) );

1. Удалите метод **main**, убрав следующий блок кода.

### public static void main( String[] args )

### { Lotto lotto = new Lotto() ; }

1. Сохраните полученный файл под именем *LottoApplet.java* — его со- держимое должно выглядеть следующим образом.

### import javax.swing.\* ; import java.awt.event.\* ;

### public class LottoApplet extends JApplet implements ActionListener



**JAVA**

### {

### // Компоненты.

### ClassLoader ldr = this.getClass().getClassLoader() ; java.net.URL iconURL = ldr.getResource( "Lotto.png" ) ; ImageIcon icon = new ImageIcon( iconURL ) ;

### JLabel img = new JLabel( icon ) ;

### JTextField txt = new JTextField( "" , 18 ) ;

### JButton btn = new JButton( "Показать счастливые номера" ) ; JPanel pnl = new JPanel() ;

### // Точка входа апплета. public void init()

### {

*LottoApplet.java*

### pnl.add( img ) ; pnl.add( txt ) ; pnl.add( btn ) ; btn.addActionListener( this ) ;

### String bgStr = getParameter( "BgColor" ) ; int bgHex = Integer.parseInt( bgStr , 16 ) ;

### pnl.setBackground( new java.awt.Color( bgHex ) ) ; add( pnl ) ;

**183**

### }

### // Обработчик событий.

### public void actionPerformed( ActionEvent event )

### {

### if ( event.getSource() == btn )

### {

### int[] nums = new int[50] ; String str = "" ;

### for ( int i = 1 ; i < 50 ; i++ ) { nums[ i ] = i ; } for ( int i = 1 ; i < 50 ; i++ )

### {

### int r = (int) Math.ceil( 49 \* Math.random() ) + 1 ; int temp=nums[i]; nums[i]=nums[r]; nums[r]=temp;

### }

### for ( int i = 1 ; i < 7 ; i++ )

**На заметку**

Компоненты **LottoApplet**,

а также обработчик событий



остались теми же самыми.

### { str += " " + Integer.toString( nums[ i ] ) + " " ; } txt.setText( str ) ;

### }

### }

### }

**10. Развертывание программ**



**HTML**

*LottoApplet.html*

**184**

**Встраивание апплетов в код веб-страницы**

Для того чтобы встроить Java-апплет в содержимое веб-страницы, в исходный код этой страницы нужно добавить некоторый дополни- тельный код HTML, определяющий пространство, в котором будет за- пускаться апплет.

В начале при помощи HTML-тега **<object>** сам Java-апплет опреде- ляется как объект при помощи назначения атрибуту **type** значения **"application/x-java-applet"**. После этого размер апплета на веб- странице определяется заданием числовых значений (в пикселях) атрибутам **width** и **height**.

Затем параметры, используемые апплетом, указываются в виде атрибутов **name** и **value** тегов **<param>**, входящих внутрь тегов **<object> </object>**. Обязательным должен быть тег **<param>**, который присваивает атрибуту **name** значение **"code"**, а атрибуту **value** — имя файла Java-апплета. Эле- мент **<object>** может также содержать всплывающее сообщение, отобра- жающееся на веб-странице в случае невозможности запустить апплет.

Каталог *bin* пакета JDK содержит инструмент **appletviewer**, который можно использовать для предварительного просмотра апплета из ука- занного документа HTML.

1. Откройте простой текстовый редактор, например Блокнот (Note- pad), затем скопируйте следующее содержимое для создания файла апплета.

### <!DOCTYPE HTML>

### <html>

### <head>

### <meta charset = "UTF-8" >

### <title>Lotto Applet Host</title>

### </head>

### <body>

### <object type = "application/x-java-applet" width = "260" height = "160" >

### <param name = "code" value = "LottoApplet.class" >

### <param name = "BgColor" value = "FFFF00" > [ Java Applet

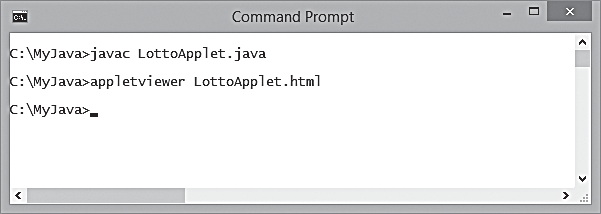
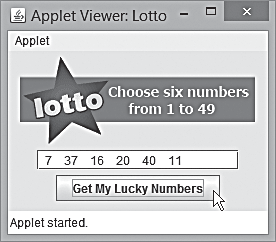
### - Requires Java Plugin ]

### </object>

### </body>

### </html>

1. Сохраните содержимое в файле с именем *LottoApplet.html* рядом с файлами программ.
2. В командной строке для создания файла *LottoApplet.class* скомпи- лируйте приведенную выше программу *LottoApplet.java*, а затем используйте инструмент **appletviewer** для предварительного про- смотра апплета.

Фон апплета *LottoApplet.class* установлен параметром **BgColor** (**FFFF00** — Желтый), а в остальном он выглядит так же, как и приложение **Lotto**. Так же, как и в случае с Java-приложениями, все файлы Java-апплетов рекомендуется собирать в один JAR-архив, чтобы, во-первых, случайно не потерять какой-нибудь ресурсный файл, а, во-вторых, уменьшить общий размер файлов. Для этого следует добавлять еще один HTML- элемент **<param>** с атрибутом **name**, имеющим значение **"archive"**, и атрибутом **value**, в котором указывается имя JAR-архива.

1. Добавьте следующий тег внутрь элемента **<object>** файла

*LottoApplet.html* и заново сохраните файл.

### <param name = "archive" value = "LottoApplet.jar" >

1. В том же каталоге, что и файл HTML, создайте архив *LottoApplet.jar*, содержащий файлы *Lotto.png* и *LottoApplet.class*, а затем просмотри- те апплет с помощью инструмента **appletviewer**, чтобы убедиться, что он запускается так же, как и раньше.



**110100101010**

**000101101011**

**110100101010**

**000101101011**

**110100101011**

**CLASS**

*LottoApplet.class*

**185**



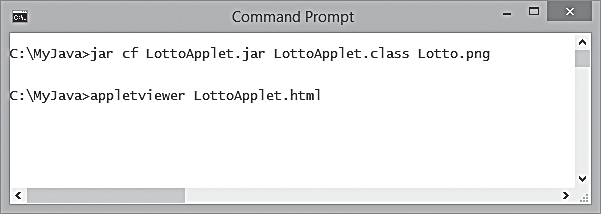
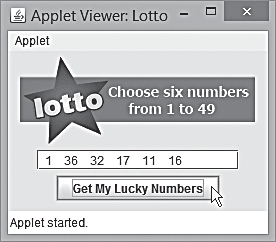
**Внимание**

Параметр **code** по-прежнему необходим, чтобы опреде- лить класс, который со- держит точку входа в про- грамму.



**JAR**

*LottoApplet.jar*



**На заметку**

Процесс задания разреше- ний и подписи JAR-файлов ранее в этой главе.

**186**

**10. Развертывание программ**



**HTML**

*PlayLotto.html*

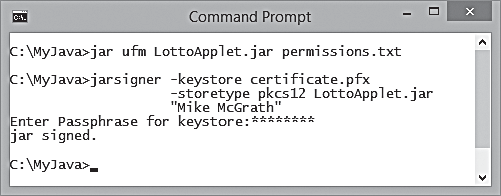
**Развертывание апплетов**

Поскольку Java-апплеты предназначены для широкого распростране- ния через Интернет, для JAR-файлов требуется задавать определенные разрешения, а также дополнять их цифровой подписью.

1. Наберите команду **jar ufm LottoApplet.jar permissions.txt**, на- жмите клавишу **Enter** для добавления атрибута **permissions** в ма- нифест.
2. Введите команду **jarsigner**, задающую файл ключа, тип хране- ния, имя JAR-файла и имя разработчика; затем нажмите клавишу **Enter**.

### jarsigner -keystore certificate.pfx

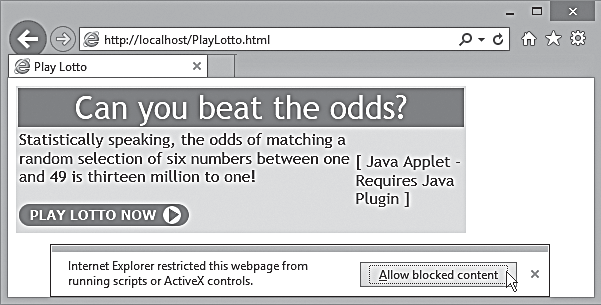
### -storetype pkcs12 LottoApplet.jar "Mike McGrath"



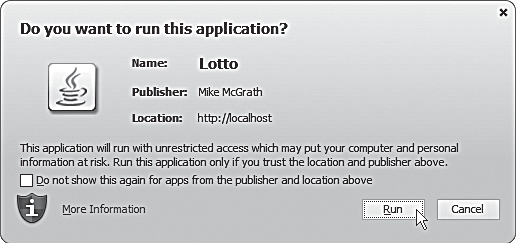
При просмотре апплетов на различных веб-браузерах и платфор- мах могут возникать различные неожиданные проблемы, поэтому хорошей практикой является тестирование апплета перед распро- странением на множестве различных окружений.

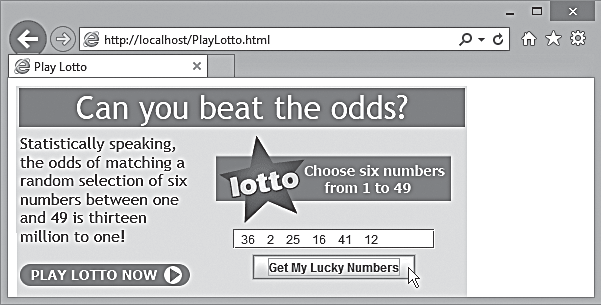
1. Скопируйте HTML-код апплета в желаемое место на веб-странице, которую нужно распространить.
2. Настройте все необходимые атрибуты HTML для размещения страницы — для **LottoApplet** установите шестнадцатеричный па- раметр **BgColor**, соответствующий цвету фона на странице.
3. Сохраните настроенную веб-страницу на компьютере и просмо- трите изменения в вашем браузере.
4. Если вы удовлетворены видом апплета, загрузите веб-страницу и JAR-файл на ваш веб-сервер, затем протестируйте его произво- дительность в различных окружениях.

**187**



Если в веб-браузер не установлен Java-плагин или Java не включе- на в настройках, то на странице вместо апплета может отображаться определенное всплывающее сообщение. В тех случаях, когда опция Java включена в браузере, апплет предоставляет на веб-странице ту же са- мую функциональность, что и Java-приложение:





Текст, заключенный между HTML-тегами **<object>** и **</ object>**, является сообще- нием, которое будет отобра- жаться в браузере в случае неудачной загрузки апплета.



**На заметку**



**Совет**

Цвет фона апплета **LottoApplet** настроен здесь таким образом, чтобы соот- ветствовать фону таблицы, в которой он появляется на веб-странице.

**Заключение**

**10. Развертывание программ**

* + Java-программы могут быть распространены в виде отдельно за- пускающихся настольных приложений, которые работают на соот- ветствующих версиях Java Runtime Environment.
  + Альтернативным способом распространения Java-программ явля- ются встроенные в веб-страницы апплеты, которые работают с со- ответствующими версиями Java-плагинов, встроенных в браузер.
  + Файлы приложений могут распространяться для дальнейшего ис- полнения на других системах с использованием подходящего ин- терпретатора **java**.
  + Упаковка всех необходимых программ в один JAR-архив позволяет избегать случайной потери ресурсных файлов.
  + JAR-архивы, которые должны быть распространены в виде на- стольных приложений, должны включать указание входной точки программы в файле манифеста.
  + Файлы манифеста JAR-архивов должны содержать атрибут Permissions.
  + JAR-приложения могут быть запущены из командной строки при помощи команды **java –jar** либо двойным нажатием по соответ- ствующей иконке.

**188**

* + Для широкого распространения Java-программ необходимо добав- лять к ним цифровые подписи.
  + Технология Java Web Start позволяет запускать приложения с помо- щью гиперссылки на веб-страницу.
  + Информация о Java-приложении хранится в формате XML в файле JNLP.
  + Перед тем как распространять Java-приложение через Интернет, веб-серверы должны быть сконфигурированы для поддержки тех- нологии Web Start.
  + Используя Java Cache Viewer, можно повторно запускать приложе- ния при помощи технологии Web Start, а также создавать ярлыки на рабочем столе для последующего запуска.
  + Java-апплеты не требуют компонентов окон и вместо метода **main**

используют метод **init**.

Параметры Java-апплета можно установить в коде HTML, тем самым настраивая апплет таким образом, чтобы он подходящим образом вписывался в окружающую его веб-страницу.

**Предметный указатель**

## G

GUI, 136

## J

**Д**

декремент, 28*,* 53 деление по модулю, 28 длина массива, 76

JAR, 139*,* 176

Java API, 82

Java-плагин, 172

## L

Lambda-выражения, 128

## А

апплет, 12*,* 172*,* 182

аргумент, 64*,* 84

архив, 176

атрибут, 108

## Б

бесконечный цикл, 54

библиотека Java, 82

бит, 42

блок кода, 52

булев, 34

## В

внешний цикл, 58

внутренний цикл, 58

всплывающая подсказка, 140

выпадающий список, 144

выход из цикла, 58*,* 59

## Г

главный метод, 51

**И**

индекс, 66

инициализатор, 55

инициализация, 35*,* 48

инкапсуляция, 112

инкремент, 28*,* 53

**189**

интерпретатор, 11

исключение, 78

итерация, 52

## К

класс, 82

ключевое слово, 50

кнопка, 138

код, 31

командная строка, 68

комментарии, 23

компилятор, 10*,* 16*,* 64

компиляция, 16*,* 64

конкатенация, 29

константа, 22*,* 84

конструктор, 110

контейнер, 136

## Л

литерал, 40*,* 90

логическое значение, 32*,* 46 локальный кэш Java, 181

## М

**10. Предметный указатель**

массив, 66

менеджер шаблона, 136

метка, 60*,* 140

метод, 64

модификатор, 54

## О

область видимости, 102

обработка событий, 154

общедоступный метод, 113

объявление, 35*,* 48*,* 66

окно, 136

округление, 86

операнд, 28

оператор, 15*,* 46

операция, 28

ошибка компиляции, 64

## П

**190**

пакет, 82

перегрузка метода, 101

переключатель, 50

переменная, 18*,* 19

переменная-счетчик, 52

побитовый оператор, 42

подкласс, 106

подстрока, 94

полоса прокрутки, 142

постфиксный, 53

префиксный, 53

приложение, 172

приоритет, 38

присваивание, 57*,* 66

проверочное выражение, 56

## Р

равенство, 30

развертывание программ, 172

размер массива, 66

**С**

свойство, 66

сертификат, 179

символ, 32

символьный, 64

синтаксис, 36

случайное число, 88

слушатель событий, 154 сообщение об ошибке, 31 список, 66

ссылка, 180

строка, 28

строковый тип, 64

суперкласс, 106

счетчик, 72

счетчик цикла, 32

## Т

тег, 184

текстовое поле, 142 технология Java Web Start, 180 тип данных, 20*,* 64

точечная запись, 66*,* 82

## У

унарный оператор, 34

условное ветвление, 32

## Ф

файл манифеста, 177

флажок, 144

## Ц

цикл, 32*,* 52*,* 72

цифровая подпись, 179

## Ч

числовой код ASCII, 64

число с плавающей точкой, 64

## Э

экземпляр класса, 90*,* 108

элемент массива, 66

*Все права защищены. Книга или любая ее часть не может быть скопирована, воспроизведена в электронной или механической форме, в виде фотокопии, записи в память ЭВМ, репродукции или каким-либо иным способом, а также использована в любой информационной системе без получения разрешения от издателя. Копирование, воспроизведение и иное использование книги или ее части без согласия издателя является незаконным и влечет уголовную, административную и гражданскую ответственность.*

Производственно-практическое издание ПРОГРАММИРОВАНИЕ ДЛЯ НАЧИНАЮЩИХ

МакГрат Майк

ПРОГРАММИРОВАНИЕ НА JAVA ДЛЯ НАЧИНАЮЩИХ

(орыс тілінде)

Директор редакции *Е. Капьёв* Ответственный редактор *В. Обручев* Художественный редактор *В. Брагина*

В оформлении обложки использована иллюстрация: Myimagine / Shutterstock.com

Используется по лицензии от Shutterstock.com

ООО «Издательство «Э»

123308, Москва, ул. Зорге, д. 1. Тел. 8 (495) 411-68-86.

Андіруші: «Э» А]Б Баспасы, 123308, М–скеу, Ресей, Зорге кИшесі, 1 зй.

Тел. 8 (495) 411-68-86.

Тауар белгісі: «Э»

]азаьстан Республикасында дистрибьютор ж–не Инім бойынша арыз-талаптарды ьабылдаушыны« Икілі «РДЦ-Алматы» ЖШС, Алматы ь., Домбровский кИш., 3«а», литер Б, офис 1.

Тел.: 8 (727) 251-59-89/90/91/92, факс: 8 (727) 251 58 12 вн. 107.

Анімні« жарамдылыь мерзімі шектелмеген.

Сертификация туралы аьпарат сайтта Андіруші «Э»

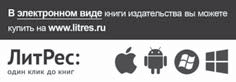
Сведения о подтверждении соответствия издания согласно законодательству РФ о техническом регулировании можно получить на сайте Издательства «Э»

Андірген мемлекет: Ресей Сертификация ьарастырылмауан

Подписано в печать 27.01.2016. Формат 84x1081/16.

Печать офсетная. Усл. печ. л. 20,16.

Тираж экз. Заказ

ИНТЕРНЕ Т-МАГАЗИН



И Н Т Е Р Н Е Т - М А Г А З И Н

И Н Т Е Р Н Е Т - М А Г А З И Н

ИНТЕРНЕ Т-МАГАЗИН

