Е. В. Кислицын, Е. И. Шишков

Разработка приложений на языке Java

Екатеринбург
2017

Министерство образования и науки Российской Федерации

Уральский государственный экономический университет

Е. В. Кислицын, Е. И. Шишков

РАЗРАБОТКА ПРИЛОЖЕНИЙ НА ЯЗЫКЕ JAVA

Рекомендовано
Советом по учебно-методическим вопросам
и качеству образования
Уральского государственного экономического университета
в качестве учебного пособия

Екатеринбург
2017

УДК 681:004.43(075.8)
ББК 39.973-018.1(я73)
К44

Рецензенты:

факультет экономики и менеджмента Уральского института управления – филиала Российской академии народного хозяйства и государственной службы при Президенте РФ;

доктор физико-математических наук,
профессор кафедры прикладной математики УралЭНИН
Уральского федерального университета
имени первого Президента России Б. Н. Ельцина
А. Н. Сесекин

Авторский коллектив:
Е. В. Кислицын (введение, глава 1), *Е. И. Шишков* (глава 2)

Кислицын, Е. В.

К44 Разработка приложений на языке Java [Текст] : учеб. пособие / Е. В. Кислицын, Е. И. Шишков ; М-во образования и науки РФ, Урал. гос. экон. ун-т. – Екатеринбург : [Изд-во Урал. гос. экон. ун-та], 2017. – 86 с.

В учебном пособии раскрываются основные понятия и методы разработки приложений на языке Java, в том числе для ОС Android. Излагаются основы языка программирования Java и объектно-ориентированного программирования. Содержатся основные сведения по созданию Android-приложений. Приведены задания для самостоятельного выполнения.

Пособие рекомендуется студентам всех форм обучения по направлениям подготовки бакалавриата 09.03.01 «Информатика и вычислительная техника», 09.03.03 «Прикладная информатика», 02.03.03 «Математическое обеспечение и администрирование информационных систем», а также по направлению подготовки магистров 09.04.03 «Прикладная информатика», изучающим курсы «Разработка кроссплатформенных приложений», «Объектно-ориентированное программирование», «Разработка программных приложений». Также может быть интересно студентам, магистрантам и аспирантам других направлений подготовки, желающим повысить свой образовательный уровень в области разработки программных и кроссплатформенных приложений.

УДК 681:004.43(075.8)
ББК 39.973-018.1(я73)

Введение

Язык Java – это объектно-ориентированный, платформенно-независимый язык программирования, используемый для разработки распределенных приложений, работающих в сети Internet. Проект Java был представлен корпорацией Sun Microsystems в 1995 г. Система программирования Java позволяет использовать World Wide Web (WWW) для распространения небольших интерактивных прикладных программ – апплетов. Они размещаются на серверах Internet, транспортируются клиенту по сети, автоматически устанавливаются и запускаются на месте как часть документа WWW. Апплет имеет весьма ограниченный доступ к ресурсам компьютера клиента, поэтому он может предоставлять произвольный мультимедийный интерфейс и выполнять сложные вычисления без риска повреждения данных на диске. Другим видом программ являются приложения Java, представляющие переносимые коды, которые могут выполняться на любом компьютере независимо от его архитектуры. Генерируемый при этом виртуальный код представляет набор инструкций для выполнения на интерпретаторе виртуального кода – виртуальной Java-машине (JVM – Java Virtual Machine). Широкое распространение получили сервлеты и JSP (Java Server Pages), предоставляющие клиентам возможность доступа к базам данных и приложениям на сервере.

Данное пособие состоит из двух глав, в конце которых приведены контрольные вопросы и задания для самостоятельного выполнения студентами. В *первой главе* вкратце изложены основные принципы языка программирования Java, основы объектно-ориентированного программирования и некоторые другие

особенности. Примеры, приведенные в первой главе, реализованы в среде IntelliJ IDEA Community 2016, являющейся доступной в сети Интернет. Руководствуясь данным пособием, практиковаться в программировании на языке Java можно и в других средах, таких как Eclipse, NetBeans и др. Отличия в самих кодах программ отсутствуют. *Вторая глава* посвящена разработке приложений под операционную систему Android. Примеры программ во второй главе представлены в среде Android Studio. Пособие обладает достаточно малым объемом и имеет своей целью знакомство читателя с основами программирования на языке Java и разработки Android-приложения. Для более подробного знакомства с данными технологиями необходимо обратиться к источникам, представленным в конце книги.

Для эффективного изучения языка программирования Java необходимо не только освоить материал данного пособия, но и обратиться к другим источникам, а также выполнить все упражнения, представленные в книге. Для решения практических задач нужно установить на свой компьютер Java SE Development Kit и Java Runtime Environment, которые можно бесплатно скачать с официального сайта компании Oracle. Также необходимо установить интегрированную среду разработки (IntelliJ IDEA или любую другую).

Глава 1

Язык программирования Java

1.1. Базовые особенности языка Java

1.1.1. Первая программа

Язык программирования Java является полностью объектно-ориентированным. Отсюда следует, что для составления любой программы, какой бы величины она ни была, необходимо прежде всего описать класс. Создадим простейшую программу «Hello, world!» в среде IntelliJ IDEA. При запуске IDE появляется стартовое окно, представленное на рис. 1.1.

Рис. 1.1. Стартовое окно

Выберите пункт «Create New Project». Далее необходимо подключить SDK, который предварительно нужно установить на ПК (рис. 1.2).

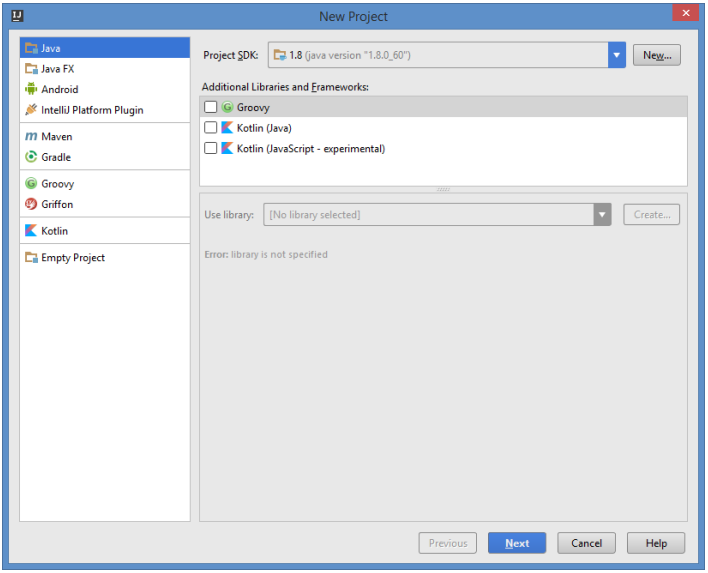

Рис. 1.2. Второй шаг настройки

На третьем шаге выберите пункт «Create project from template». Наконец, на последнем шаге введите имя вашего проекта (Hello). Также здесь вы можете настроить путь сохранения вашего проекта.

Теперь напишите вашу первую программу, как показано на рис. 1.3.

```
1  package com.company;
2
3  public class Main {
4
5      public static void main(String[] args) {
6          System.out.println("Hello, World!");
7      }
8  }
```

Рис. 1.3. Код программы «Hello, World!»

После компиляции и запуска программы в среде IntelliJ IDEA в окне вывода появится сообщение: «Hello, World!»

(рис. 1.4). Для того чтобы скомпилировать приложение, нажмите Ctrl+F9, а для запуска Shift+F10.

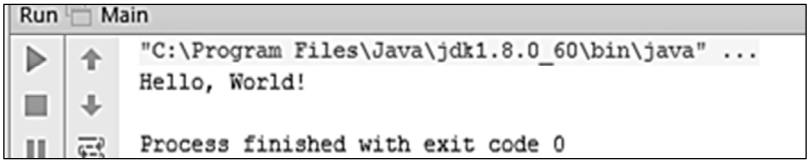

Рис. 1.4. Работа программы «Hello, World!»

Проанализируем написанный код. Фигурными скобками в языке Java отмечаются блоки командного кода. Весь командный код размещается между открывающейся и закрывающейся фигурными скобками. Внешние скобки используются для определения программного кода класса, а внутренние – для метода main.

Описание любого класса начинается с ключевого слова `class`. После этого слова идет само имя класса, которое программист придумывает самостоятельно, исходя из условий программы. Соответственно, все, что находится в фигурных скобках после конструкции `class Hello { }`, относится к этому классу.

Программный код класса Hello состоит только из одного метода main(). Здесь важно запомнить, что выполнение Java-программы всегда начинается с вызова метода main. В этом методе представлен код, который будет выполняться в результате вызова программы. Любая программа, за исключением апплетов, может содержать только один метод main.

Перейдем к описанию спецификаций метода main: public – метод доступен за пределами класса; static – метод является статическим и для его вызова нет необходимости создавать экземпляр класса; void – метод не возвращает результат. Инструкция String[] args, находящаяся в круглых скобках после названия метода, означает тип аргумента метода, т. е. формальное название аргумента – args, который является текстовым массивом.

Все команды программного кода заканчиваются точкой с запятой. Для вывода информации на экран в консольных приложениях используется метод println() объекта out объекта пото-

ка стандартного вывода System. Соответственно, чтобы вывести на экран какое-либо сообщение, достаточно написать:

```
System.out.println("Ваше сообщение");
```

1.1.2. Комментарии в Java-программе

В языке Java используется три вида комментариев.

1. Однострочный комментарий начинается с символа //. Все, что находится справа от данного символа и до конца строки, является комментарием.

2. Многострочный комментарий начинается с последовательности символов /* и заканчивается */. Все, что находится внутри, является комментарием и игнорируется компилятором.

3. Многострочный комментарий документационной информации начинается последовательностью символов /** и заканчивается */. Используется для выделения в качестве комментария данных справочного характера. Он может содержать дескрипторы вида:

@author – задает сведения об авторе;

@exception – задает имя класса исключения;

@param – описывает параметры, передаваемые методу;

@return – описывает тип, возвращаемый методом;

@throws – описывает исключение, генерируемое методом.

Из java-файла, содержащего такие комментарии, соответствующая утилита javadoc.exe может извлекать информацию для документирования классов и сохранения ее в виде HTML-документа.

1.1.3. Простые типы данных и операторы

Все данные в Java делятся на простые и ссылочные. Разница состоит в том, что простые типы данных передаются по значению, а ссылочные через ссылку. В языке Java выделяют 8 базовых простых типов данных:

1) *byte* – целые числа в диапазоне от –128 до 127 (8 бит);

2) *short* – целые числа в диапазоне от –32768 до 32767 (16 бит);

3) *int* – целые числа в диапазоне от –2147483648 до 2147483647 (32 бита);

4) *long* – целые числа в диапазоне от –9223372036854775808 до 9223372036854775807 (64 бита);

5) *float* – действительные числа в диапазоне от $3{,}4*10^{-38}$ до $3{,}4*10^{38}$ (32 бита);

6) *double* – действительные числа двойной точности в диапазоне от $1{,}7*10^{-308}$ до $1{,}7*10^{308}$ (64 бита);

7) *char* – символьный тип для представления символьных значений в диапазоне от 0 до 65536;

8) *boolean* – логический тип, принимающая одно из двух значений: true или false.

После указания типа переменной ей необходимо присвоить значение. Это делается с помощью литералов. *Литерал* – это постоянное значение, предназначенное для восприятия человеком, которое не может быть изменено в программе. В действительных числах дробная часть отделяется точкой. Символы вводятся в одинарных кавычках. Для ввода логических значений используются ключевые слова true и false. При объявлении переменной перед ее именем обязательно указывается идентификатор типа (см. выше). Затем ей может быть присвоено значение с помощью оператора =.

Все операторы в Java делятся на 4 группы: арифметические, логические, побитовые и сравнения, основные из которых представлены в табл. 1.1.

Таблица 1.1

Операторы в Java

Оператор	Название	Оператор	Название
Арифметические		*Логические*	
+	Сложение	&&	Логическое И
–	Вычитание	||	Логическое ИЛИ
*	Умножение	!	Логическое отрицание
/	Деление	*Сравнения*	
%	Остаток	<	Меньше
+=	Сложение с присваиванием	<=	Меньше или равно
–=	Вычитание с присваиванием	>	Больше
*=	Умножение с присваиванием	>=	Больше или равно
++	Инкремент	!=	Не равно
--	Декремент	==	Равно

1.1.4. Операторы управления

Оператор if позволяет условное выполнение оператора или условный выбор двух операторов, выполняя один или другой, но не оба сразу.

```
if (boolexp) { /*операторы*/}
else { /*операторы boolexp */ }//может отсутствовать
```

Оператор switch передает управление одному из нескольких операторов в зависимости от значения выражения.

```
switch(exp) {
case exp1:/*операторы, если exp==exp1*/
break;
case exp2:/*операторы, если exp==exp2*/
break;
default: /* операторы Java */
}
```

При совпадении условий вида *exp==exp1* выполняются подряд все блоки операторов до тех пор, пока не встретится оператор break.

Циклы:

```
1. while (boolexpr) { /*операторы*/ }
2. do { /*операторы*/ }
while (boolexp);
3. for(exp1;boolexp; exp3){ /*операторы*/ }
4. for(int a:X) { /*операторы*/}
```

Циклы выполняются, пока булевское выражение *boolexp* равно true.

В *операторе for* предусмотрены места для всех четырех частей цикла. Ниже приведена общая форма оператора записи for.

```
for (инициализация; завершение; итерация) тело;
```

Также *оператор break* используется для досрочного выхода из цикла, а *оператор continue* – для перехода на следующую итерацию цикла.

1.1.5. Массивы

Массив представляет собой совокупность однотипных переменных с общим для обращения к ним именем. В Java массивы могут быть как одномерными, так и многомерными.

Одномерный массив представляет собой список связанных переменных. Для объявления одномерного массива обычно применяется следующая общая форма:

```
тип имя_массива[] = new тип[размер];
```

где тип объявляет конкретный тип элемента массива. Тип элемента, называемый также базовым, определяет тип данных каждого элемента, составляющего массив. А размер определяет число элементов массива. В связи с тем что массивы реализованы в виде объектов, создание массива происходит в два этапа. Сначала объявляется переменная, ссылающаяся на массив, затем выделяется память для массива, а ссылка на нее присваивается переменной массива. Следовательно, память для массивов в Java динамически распределяется с помощью оператора new.

Среди многомерных массивов наиболее простыми являются двумерные. Двумерный массив, по существу, представляет собой ряд одномерных массивов. Для того чтобы объявить двумерный целочисленный табличный массив table размерами 10×20, следует написать такое выражение:

```
int table[][] = new int[10][20];
```

Выделяя память под многомерный массив, достаточно указать лишь первый (крайний слева) размер. А память под остальные размеры массива можно выделять по отдельности. Например, в приведенном ниже фрагменте кода память выделяется только под первый размер двумерного массива table. А под второй его размер она выделяется вручную.

```
int table [][] = new int[3][];
table [0] = new int [4];
table[1] = new int[4];
table[2] = new int[4];
```

в Java допускаются массивы размерностью больше двух. Ниже приведена общая форма объявления многомерного массива.

```
тип имя_массива[][]...[] = new тип[размер_1][размер_2] ...
[размер_N];
```

1.2. Основы объектно-ориентированного программирования

1.2.1. Инкапсуляция, наследование и полиморфизм

Язык программирования Java является полностью объектно-ориентированным. Это означает, что программа, написанная на языке Java, должна строго соответствовать парадигме объектно-ориентированного программирования (ООП). Обычно выделяют три фундаментальных принципа, которые составляют основу ООП: инкапсуляция, полиморфизм и наследование. Кроме этих принципов, вводятся также два важных понятия – класс и объект.

Концепция процедурного программирования могла бы быть сформулирована как система составления программного кода, действующего на данные. В этом случае приоритет остается за программным кодом. В ООП предпочтение отдается данным. Именно данные управляют доступом к программному коду. В зависимости от того, какие данные обрабатываются, определяются методы для их обработки. В объектно-ориентированных языках программирования эта концепция реализуется через механизмы инкапсуляции, полиморфизма и наследования.

Инкапсуляция позволяет объединить данные и код обработки этих данных в одно целое. В результате получается нечто наподобие «черного ящика», в котором содержатся все необходимые данные и код. Указанным способом создаются объекты, являющиеся именно той конструкцией, которая поддерживает и через которую реализуется механизм инкапсуляции.

Полиморфизм позволяет использовать один и тот же интерфейс для выполнения различных действий. Здесь действует принцип «один интерфейс – много методов». Благодаря поли-

морфизму программы становятся менее сложными, так как для определения и выполнения однотипных действий служит единый интерфейс. Такой единый интерфейс применяется пользователем или программистом к объектам разного типа, а выбор конкретного метода для реализации соответствующей команды осуществляется компьютером в соответствии с типом объекта, для которого выполняется команда.

Наследование позволяет усовершенствовать эволюционным способом программный код, сохраняя при этом на приемлемом уровне сложность программы. Наследование – это механизм, с помощью которого один объект может получить свойства другого объекта. Это позволяет создавать на основе уже существующих объектов новые объекты с новыми свойствами, сохраняя при этом свойства старых объектов. Например, для создания нового класса «Домашняя кошка», который от класса «Кошка» отличается наличием поля «награды на выставках», в общем случае пришлось бы заново создавать класс, описывая в явном виде все его поля и методы. В рамках ООП с помощью механизма наследования можно создать новый класс «Домашняя кошка» на основе уже существующего класса «Кошка», добавив в описании класса только новые свойства – старые наследуются автоматически.

1.2.2. Классы и объекты

Класс представляет собой шаблон, по которому определяется форма объекта. В нем указываются данные и код, который будет оперировать этими данными. В Java используется спецификация класса для построения объектов, которые являются экземплярами класса. Следовательно, класс, по существу, представляет собой ряд схематических описаний способа построения объекта. При этом очень важно подчеркнуть, что класс является логической абстракцией. Физическое представление класса появится в оперативной памяти лишь после того, как будет создан объект этого класса. Следует также иметь в виду, что методы и переменные, составляющие класс, принято называть членами класса. А члены данных называются переменными экземпляра.

В теле класса перечисляются с указанием типа переменные – поля класса. Что касается методов, то это обычные функции, только описанные в рамках класса. На рис. 1.5 представлен общий вид класса.

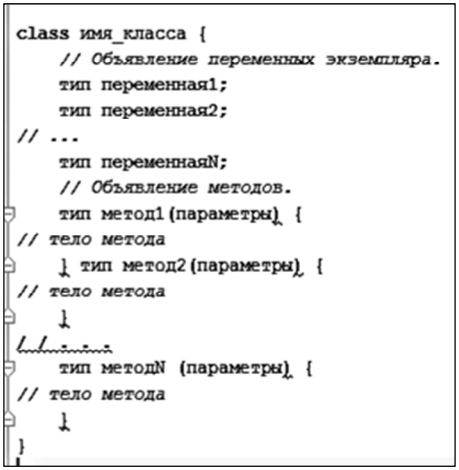

Рис. 1.5. Обобщенная структура класса

Проиллюстрируем создание класса на примере сведений о работниках предприятия. Назовем этот класс Worker, и в нем будут храниться следующие данные: стаж работы, тарифная ставка, количество отработанных за месяц часов. На рис. 1.6 представлен первоначальный вариант класса Worker, в котором определены три переменные экземпляра: experience, salary и time.

```
 6   public class Worker {
 7       int experience;
 8       double salary;
 9       double time;
10
11   }
```

Рис. 1.6. Первоначальный вид класса Worker

Слово class указывает на создание нового типа данных. В данном случае этот тип называется Worker. Пользуясь этим именем, можно теперь создавать объекты типа Worker. Но указанный выше код не приводит к появлению объектов типа Worker. Для того чтобы создать реальный объект, потребуется оператор, аналогичный следующему:

```
Worker manager = new Worker(); // создать объект manager типа Worker
```

После выполнения этого оператора объект manager станет экземпляром класса Worker. Иными словами, класс обретет физическое воплощение. Всякий раз, когда создается экземпляр класса, строится объект, содержащий копии всех переменных экземпляра, определенных в классе. Для обращения к переменным используется оператор-точка (.). Этот оператор связывает имя объекта с именем члена класса. На рис. 1.7 приведен пример обращения к переменным класса Worker.

```
6  public class WorkerDemo {
7      public static void main (String args[]) {
8          Worker manager = new Worker();
9          double plata;
10         manager.experience=3;
11         manager.salary=85.5;
12         manager.time=168;
13         // рассчитать месячную заработную плату менеджера
14         plata=manager.salary*manager.time;
15         System.out.println("В этом месяце с тарифной ставкой "+manager.salary
16                 +" руб./час менеджер получит "+plata+" рублей.");
17     }
18 }
```

Рис. 1.7. Класс WorkerDemo

После выполнения программы мы получим результат (рис. 1.8).

```
"C:\Program Files\Java\jdk1.8.0_60\bin\java" ...
В этом месяце с тарифной ставкой 85.5 руб./час менеджер получит 14364.0 рублей.

Process finished with exit code 0
```

Рис. 1.8. Результат выполнения программы

Каждый объект содержит свои копии переменных экземпляра, определенные в его классе. Следовательно, содержимое переменных в одном объекте может отличаться от содержимого тех же самых переменных в другом объекте. Между объектами нет никакой связи, за исключением того, что они относятся к одному и тому же типу. Так, если имеются два объекта типа Worker, каждый из них содержит собственную копию переменных salary, time и т. д., причем значения одноименных переменных в этих двух объектах могут различаться.

В операции присваивания переменные ссылочного типа действуют иначе, чем переменные такого простого типа. Когда одна переменная ссылки на объект присваивается другой, ситуация несколько усложняется, поскольку такое присваивание приводит к тому, что переменная, находящаяся в левой части оператора присваивания, ссылается на тот же самый объект, на который ссылается переменная, находящаяся в правой части этого оператора. Сам же объект не копируется. В силу этого отличия присваивание переменных ссылочного типа может привести к несколько неожиданным результатам. В качестве примера рассмотрим следующий фрагмент кода:

```
Worker menl = new Worker();
Worker men2 = menl;
```

На первый взгляд, переменные menl и men2 ссылаются на совершенно разные объекты, но на самом деле они ссылаются на один и тот же объект.

1.2.3. Методы

Описание метода состоит из сигнатуры и тела метода. Сигнатура метода, в свою очередь, состоит из ключевого слова, которое обозначает тип возвращаемого методом результата, имени метода и списка аргументов в круглых скобках после имени метода. Аргументы разделяются запятыми, для каждого аргумента перед формальным именем аргумента указывается его тип. Тело метода заключается в фигурные скобки и содержит код, определяющий функциональность метода. В качестве значения методы в Java могут возвращать значения простых (ба-

зовых) или ссылочных типов (объекты). Если метод не возвращает результат, в качестве идентификатора типа указывается ключевое слово void. Значение, возвращаемое методом, указывается после инструкции return.

Для иллюстрации добавим в класс Worker метод, рассчитывающий заработную плату работника за месяц (рис. 1.9, 1.10).

```
6   public class Worker {
7       int experience;
8       double salary;
9       double time;
10
11      void plata() {
12          System.out.println("За месяц работник получит "+salary*time+" рублей.");
13      }
14  }
```

Рис. 1.9. Добавление метода в класс Worker

```
6   public class WorkerDemo {
7       public static void main (String args[]) {
8           Worker manager = new Worker();
9           Worker operatorPC = new Worker();
10          // присвоить значения полям в объекте manager
11          manager.experience=3;
12          manager.salary=85.5;
13          manager.time=168;
14          // присвоить значения полям в объекте operatorPC
15          operatorPC.experience=5;
16          operatorPC.salary=78.6;
17          operatorPC.time=176;
18          // рассчитать месячную заработную плату менеджера и оператора ПК
19          System.out.print("Менеджер работает "+manager.experience+" года/лет.");
20          manager.plata();
21          System.out.print("Оператор ПК работает "+manager.experience+" года/лет.");
22          operatorPC.plata();
23
24      }
25  }
```

Рис. 1.10. Использование методов в классе WorkerDemo

Выполните программу и проверьте результат. В качестве типа, возвращаемого методом plata, указано ключевое слово void. Это значит, что данный метод не возвращает никаких данных вызывающей части программы. Сам метод располагается в фигурных скобках. После этого управление передается обрат-

но вызывающей части программы. Рассмотрим строку кода в методе main:

```
Manager.plata();
```

В этой строке кода вызывается метод plata() для объекта manager. Для вызова метода относительно объекта перед его именем указываются имя объекта и оператор-точка. При вызове метода ему передается управление. Когда метод завершит свое действие, управление будет возвращено вызывающей части программы, и ее выполнение продолжится со строки кода, следующей за вызовом этого метода. В данном случае в результате вызова manager.plata() отображается месячная заработная плата, определяемая объектом manager. Аналогично при вызове operatorPC.plata() на экран выводится зарплата, определяемая объектом operatorPC.

Необходимо отметить следующую особенность метода plata(): в нем выполняется непосредственное обращение к переменным экземпляра salary и time, т. е. перед ними не указываются имя объекта и оператор-точка. Если в методе используется переменная экземпляра, определенная в его классе, обращаться к ней можно напрямую, не указывая объект.

Возврат из метода осуществляется при выполнении одного из двух условий. Первое, когда признаком завершения метода и возврата из него служит закрывающая круглая скобка. Вторым условием является выполнение оператора return. Существуют две разновидности оператора return: одна – для методов типа void, не возвращающих значение, а другая – для методов, возвращающих значение вызывающей части программы. Рассмотрим первую. Организовать немедленное завершение метода типа void и возврат из него можно с помощью следующей формы оператора return:

```
return;
```

При выполнении этого оператора управление будет возвращено вызывающей части программы, а оставшийся в методе код будет проигнорирован. Рассмотрим в качестве примера метод, представленный на рис. 1.11.

Здесь переменная цикла for принимает лишь значения от 0 до 10. Как только значение переменной i становится равным 10, цикл завершается и происходит возврат из метода. В одном методе допускается несколько операторов return. Необходимость в них возникает в том случае, если в методе организовано несколько ветвей выполнения.

```
void myMeth() {
    int i;
    for(i=0; i<25; i++) {
        if(i == 10) return; // завершить цикл на значении 10
        System.out.println();
    }
}
```

Рис. 1.11. Пример метода с невозвращаемым значением

Способность возвращать значение является одним из самых полезных свойств метода. Методы возвращают значения вызывающей части программы, используя следующую форму оператора return:

return значение;

где значение – конкретное возвращаемое значение. Данная форма оператора return может быть использована только в тех методах, тип которых отличается от типа void. Изменим метод plata() из предыдущего примера, чтобы он возвращал значение. Вместо того чтобы отображать месячную зарплату в методе plata(), ограничимся ее расчетом и возвратом полученного значения. Преимущество такого подхода заключается, в частности, в том, что возвращаемое значение может быть использовано при выполнении других расчетов. Измененный метод plata() представлен на рис. 1.12.

На рис. 1.13 представлен метод main(), в котором используется измененный метод plata().

Проверьте правильность выполнения программы. Обратите внимание на то, что вызов метода plata() в данной программе указывается в правой части оператора присваивания, тогда как в левой его части – переменная, которая принимает значение, возвращаемое этим методом. Несмотря на то что приведенная

выше программа компилируется и выполняется без ошибок, ее эффективность можно повысить. В частности, переменные range1 и range2 в ней не нужны. Вызов метода plata() можно непосредственно указать в качестве параметра метода println().

```
6   public class Worker {
7       int experience;
8       double salary;
9       double time;
10  // Возвращает месячную заработную плату сотрудника
11      double plata() {
12          return salary*time;
13      }
14  }
```

Рис. 1.12. Класс Worker с методом, возвращающим значение

```
6   public class WorkerDemo {
7       public static void main (String args[]) {
8           Worker manager = new Worker();
9           Worker operatorPC = new Worker();
10          double range1, range2;
11          // присвоить значения полям в объекте manager
12          manager.experience=3;
13          manager.salary=85.5;
14          manager.time=168;
15          // присвоить значения полям в объекте operatorPC
16          operatorPC.experience=5;
17          operatorPC.salary=78.6;
18          operatorPC.time=176;
19          // рассчитать месячную заработную плату менеджера и оператора ПК
20          range1 = manager.plata();
21          range2 = operatorPC.plata();
22          //Вывести зарплату сотрудников
23          System.out.print("Менеджер работает "+manager.experience+" года/лет и " +
24                  "получает "+range1+" рублей в месяц.");
25          System.out.print("Оператор ПК работает "+manager.experience+" года/лет и " +
26                  "получает "+range2+" рублей в месяц.");
27      }
28  }
```

Рис. 1.13. Применение методов, возвращающих значение

При вызове метода ему можно передать одно или несколько значений. Значение, передаваемое методу, называется аргументом. А переменная, получающая аргумент, называется формальным параметром или просто параметром. Параметры объявляются в скобках после имени метода. Синтаксис объявления параметров такой же, как и у переменных. А областью действия параметров является тело метода. За исключением

особых случаев передачи аргументов методу параметры действуют так же, как и любые другие переменные. Добавим в класс Worker параметризированный метод, который будет рассчитывать заработную плату с учетом персональной надбавки (в процентах) и премии, как показано на рис. 1.14.

```
6   public class Worker {
7       int experience;
8       double salary;
9       double time;
10  // Возвращает месячную заработную плату сотрудника
11      double plata() {
12          return salary*time;
13      }
14  // Возвращает месячную заработную плату сотрудника
15  // с учетом премии и надбавки
16      double plataPlus(int plus, int premia) {
17          return (salary*time*(1+plus/100)+premia);
18      }
19  }
```

Рис. 1.14. Добавление нового метода в класс Worker

Главный класс будет выглядеть, как показано на рис. 1.15.

```
    public static void main (String args[]) {
        Worker manager = new Worker();
        Worker operatorPC = new Worker();
        double range1, range2;
        // присвоить значения полям в объекте manager
        manager.experience=3;
        manager.salary=85.5;
        manager.time=168;
        // присвоить значения полям в объекте operatorPC
        operatorPC.experience=5;
        operatorPC.salary=78.6;
        operatorPC.time=176;
        // добавить переменные с надбавками
        int nadbavkaManager = 10;
        int nadbavkaOperator = 15;
        int premia = 3000;
        //Вывести зарплату сотрудников
        System.out.print("Менеджер получил "
                +manager.plataPlus(nadbavkaManager, premia)+" рублей за месяц.");
        System.out.print("Оператор ПК получил "
                +operatorPC.plataPlus(nadbavkaOperator, premia)+" рублей за месяц.");
    }
}
```

Рис. 1.15. Измененный метод main()

1.2.4. Конструкторы

Конструктор инициализирует объект при его создании. У конструктора такое же имя, как и у его класса, а с точки зрения синтаксиса он подобен методу. Но у конструкторов нет возвращаемого типа, указываемого явно. Как правило, конструкторы используются для задания первоначальных значений переменных экземпляра, определенных в классе, или же для выполнения любых других установочных процедур, которые требуются для создания полностью сформированного объекта. У всех классов имеются конструкторы, независимо от того, определите вы их или нет, поскольку в Java автоматически предоставляется конструктор, используемый по умолчанию и инициализирующий все переменные экземпляра их значениями по умолчанию. Для большинства типов данных значением по умолчанию является нулевое, для типа bool – логическое значение false, а для ссылочных типов – пустое значение null. Но как только вы определите свой собственный конструктор, конструктор по умолчанию больше не используется. Причем конструктор, как и метод, может либо иметь параметры, либо не иметь. На рис. 1.16 представлен пример конструктора для класса Worker.

```
public class Worker {
    int experience;
    double salary;
    double time;
// Конструктор
    Worker(int e, double s, double t) {
        experience = e;
        salary = s;
        time = t;
    }

    // Возвращает месячную заработную плату сотрудника
    double plata() {
        return salary*time;
    }
// Возвращает месячную заработную плату сотрудника
// с учетом премии и надбавки
    double plataPlus(int plus, int premia) {
        return (salary*time*(1+plus/100)+premia);
    }
}
```

Рис. 1.16. Конструктор класса Worker

На рис. 1.17 представлен пример использования созданного конструктора.

```
public class WorkerDemo {
    public static void main (String args[]) {
        Worker manager = new Worker(3,85.5,168);
        Worker operatorPC = new Worker(5,78.6,176);
        double range1, range2;
        // добавить переменные с надбавками
        int nadbavkaManager = 10;
        int nadbavkaOperator = 15;
        int premia = 3000;
        //Вывести зарплату сотрудников
        System.out.print("Менеджер получил "
                +manager.plataPlus(nadbavkaManager, premia)+" рублей за месяц.");
        System.out.print("Оператор ПК получил "
                +operatorPC.plataPlus(nadbavkaOperator, premia)+" рублей за месяц.");
    }
}
```

Рис. 1.17. Применение конструкторов с параметрами

1.2.5. Ключевое слово this

Когда метод вызывается, ему автоматически передается ссылка на вызывающий объект, т. е. тот объект, для которого вызывается данный метод. Эта ссылка обозначается ключевым словом this. Следовательно, ключевое слово this обозначает именно тот объект, по ссылке на который действует вызываемый метод. Добавим this в конструктор класса Worker, как показано на рис. 1.18.

```
// Конструктор
    Worker(int experience, double salary, double time) {
        this.experience = experience;
        this.salary = salary;
        this.time = time;
    }
```

Рис. 1.18. Применение ключевого слова this

1.2.6. Модификаторы доступа

Управление доступом к членам класса в Java осуществляется с помощью трех модификаторов доступа: public, private

и protected. Если модификатор не указан, то принимается тип доступа по умолчанию. Когда член класса обозначается модификатором public, он становится доступным из любого другого кода в программе, включая и методы, определенные в других классах. Когда же член класса обозначается модификатором private, он может быть доступен только другим членам этого же класса. Следовательно, методы из других классов не имеют доступа к закрытому члену данного класса. Если все классы в программе относятся к одному пакету, то отсутствие модификатора доступа равнозначно указанию модификатора public по умолчанию. *Пакет* представляет собой группу классов, предназначенных как для организации классов, так и для управления доступом. Модификатор доступа указывается перед остальной частью описания типа отдельного члена класса. Это означает, что именно с него должен начинаться оператор объявления члена класса.

1.2.7. Перегрузка методов

Несколько методов одного класса могут иметь одно и то же имя, отличаясь лишь набором параметров. Перегрузка методов является одним из способов реализации принципа полиморфизма в Java. Для того чтобы перегрузить метод, достаточно объявить его новый вариант, отличающийся от уже существующих, а все остальное сделает компилятор. Нужно лишь соблюсти одно условие: тип и(или) число параметров в каждом из перегружаемых методов должны быть разными. Некоторые считают, что два метода могут различаться лишь типом возвращаемого значения, но это заблуждение. Возвращаемый тип не предоставляет достаточных сведений для принятия решения о том, какой именно метод должен быть использован. Конечно, перегружаемые методы могут иметь разные возвращаемые типы, но при вызове метода выполняется лишь тот его вариант, в котором параметры соответствуют передаваемым аргументам.

На рис. 1.19 и 1.20 приведен простой пример программы, демонстрирующий перегрузку методов.

```
public class Peregruzka {
    // Первый вариант метода.
    void ovlDemo() {
        System.out.println("Нет параметров");
    }
    // перегрузить метод ovlDemo с одним параметром типа int.
// Второй вариант метода.
    void ovlDemo(int a) {
        System.out.println("Один параметр: " + a) ;
    }
    // перегрузить метод ovlDemo с двумя параметрами типа int.
// Третий вариант метода.
    int ovlDemo(int a, int b) {
        System.out.println("Два параметра: " + a + " " + b);
        return a + b;
    }
    // перегрузить метод ovlDemo с двумя параметрами типа doTible.
// Четвертый вариант метода.
    double ovlDemo(double a, double b) {
        System.out.println("Два вещественных параметра: " +
                a + " "+ b);
        return a + b;
    }
```

Рис. 1.19. Перегрузка методов

```
public class PeregruzkaDemo {
    public static void main(String args[]) {
        Peregruzka ob = new Peregruzka();
        int resI;
        double resD;
// вызвать все варианты метода ovlDemo()
        ob.ovlDemo();
        System.out.println ();
        ob.ovlDemo(2);
        System.out.println ();
        resI = ob.ovlDemo(4, 6);
        System.out.println("Result of ob.ovlDemo(4, 6): " + resI);
        System.out.println();
        resD = ob.ovlDemo(1.1, 2.32);
        System.out.println("Result of ob.ovlDemo(1.1, 2.32): " +
                resD);
    }
}
```

Рис. 1.20. Использование перегрузки методов

Как видите, метод ovlDemo() перегружается четырежды. В первом его варианте параметры не предусмотрены, во втором – определен один целочисленный параметр, в третьем – два целочисленных параметра, в четвертом – два параметра типа double. Обратите внимание на то, что первые два варианта метода ovlDemo () имеют тип void, а два других возвращают значение.

Перегрузка методов представляет собой механизм воплощения полиморфизма, т. е. способ реализации в Java принципа «один интерфейс – множество методов». Для того чтобы стало понятнее, как и для чего это делается, необходимо принять во внимание следующее соображение: в языках программирования, не поддерживающих перегрузку методов, каждый метод должен иметь уникальное имя. Но в ряде случаев требуется выполнять одну и ту же последовательность операций с разными типами данных. Главная ценность перегрузки заключается в том, что она обеспечивает доступ к связанным вместе методам по общему имени. Благодаря полиморфизму несколько имен сводятся к одному. Для установления взаимосвязи между перегружаемыми методами не существует какого-то твердого правила, но с точки зрения правильного стиля программирования перегрузка методов подразумевает подобную взаимосвязь. Использовать одно и то же имя для не связанных друг с другом методов не рекомендуется, хотя это и возможно. На практике перегружать следует только тесно связанные операции.

Как и методы, конструкторы также могут перегружаться. Это дает возможность конструировать объекты самыми разными способами.

1.2.8. Ключевое слово static

Иногда требуется определить такой член класса, который будет использоваться независимо от всех остальных объектов этого класса. Как правило, доступ к члену класса организуется посредством объекта этого класса, но в то же время можно создать член класса для самостоятельного применения без ссылки на конкретный экземпляр объекта. Для того чтобы создать такой член класса, достаточно указать в самом начале его объявления ключевое слово static. Если член класса объявляется как static,

он становится доступным до создания любых объектов своего класса и без ссылки на какой-либо объект. С помощью ключевого слова static можно объявлять как переменные, так и методы. Наиболее характерным примером члена типа static служит метод main(), который объявляется таковым потому, что он должен вызываться виртуальной машиной Java в самом начале выполняемой программы.

Для того чтобы воспользоваться членом типа static за пределами класса, достаточно указать имя этого класса с оператором-точкой. Но создавать объект для этого не нужно. В действительности член типа static оказывается доступным не по ссылке на объект, а по имени своего класса. Так, если требуется присвоить значение 10-й переменной count типа static, являющейся членом класса Timer, то для этой цели можно воспользоваться следующей строкой кода:

```
Timer.count = 10;
```

Эта форма записи подобна той, что используется для доступа к обычным переменным экземпляра посредством объекта, но в ней указывается имя класса, а не объекта. Аналогичным образом можно вызвать метод типа static, используя имя класса и оператор-точку. Переменные, объявляемые как static, по существу являются глобальными. Когда же объекты объявляются в своем классе, копия переменной типа static не создается. Вместо этого все экземпляры класса совместно пользуются одной и той же переменной типа static.

1.2.9. Вложенные и внутренние классы

В Java определены вложенные классы. *Вложенным* называется такой класс, который объявляется в другом классе. Вложенный класс не может существовать независимо от объемлющего класса, потому что последний ограничивает область его действия. Если вложенный класс объявлен в пределах области действия объемлющего класса, он становится членом последнего. Имеется также возможность объявить вложенный класс, который станет локальным в пределах блока. Существует два типа вложенных классов. Одни вложенные классы объявляются

с помощью модификатора доступа static, а другие – без него. Классы такого типа называются внутренними. Внутренний класс имеет доступ ко всем переменным и методам внешнего класса и может обращаться к ним непосредственно, как и все остальные нестатические члены внешнего класса. Иногда внутренний класс используется для предоставления услуг объемлющему классу.

1.2.10. Наследование

В языке Java наследуемый класс принято называть суперклассом, а наследующий от него класс – подклассом. Следовательно, подкласс – это специализированный вариант суперкласса. Он наследует все переменные и методы, определенные в суперклассе, дополняя их своими элементами.

Наследование одних классов от других отражается в Java при объявлении класса. Для этой цели служит ключевое слово extends. Подкласс дополняет суперкласс, расширяя его. Рассмотрим пример программы, демонстрирующий некоторые свойства наследования. В этой программе определен суперкласс TwoDShape (рис. 1.21), хранящий сведения о ширине и высоте двумерного объекта.

```
// Класс, описывающий двумерные объекты
public class TwoDShape {
    double width;
    double height;
    void showDim() {
        System.out.println("Ширина и высота фигуры равны " +
                width + " и " + height);
    }
}
```

Рис. 1.21. Класс TwoDShape

Там же определен его подкласс Triangle (рис. 1.22), хранящий сведения о треугольнике.

Обращение к объектам класса Triangle реализовано в классе Shapes (рис. 1.23).

```
// Подкласс класса TwoDShape для представления треугольников.
// Класс Triangle наследуется от класса TwoDShape
public class Triangle extends TwoDShape{
    String style;
    double area() {
//Из класса Triangle можно обращаться к членам класса
// TwoDShape таким же образом, как и к собственным членам.
        return width * height / 2;
    }
    void showStyle() {
        System.out.println("Triangle is " + style);
    }
}
```

Рис. 1.22. Класс Triangle

```
public class Shapes {
    public static void main(String args[]) {
        Triangle t1 = new Triangle();
        Triangle t2 = new Triangle();
// Все члены класса Triangle, даже унаследованные от класса
// TwoDShape, доступны из объектов типа Triangle.
        t1.width = 4.0;
        t1.height = 4.0;
        t1.style = "равнобедренный";
        t2.width = 8.0;
        t2.height = 12.0;
        t2.style = "прямоугольный";
        System.out.println("Info for t1: ");
        t1.showStyle ();
        t1. showDim() ;
        System.out.println ("Площадь фигуры " + t1.area());
        System.out.println ();
        System.out.println("Info for t2; ");
        t2.showStyle();
        t2.showDim();
        System.out.println("Площадь фигуры " + t2.area());
    }
}
```

Рис. 1.23. Класс Shapes

Результат выполнения программы представлен на рис. 1.24.

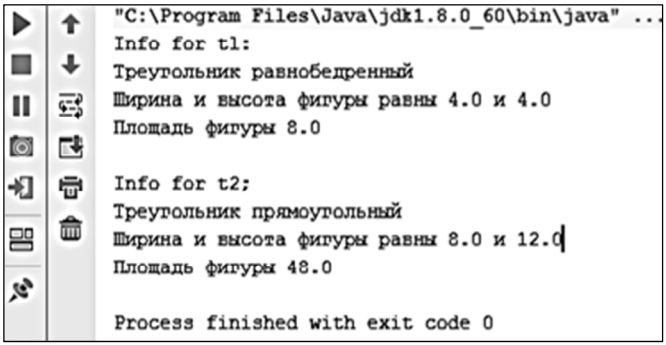

Рис. 1.24. Результат выполнения программы

Здесь в классе TwoDShape определены атрибуты обобщенной двумерной фигуры, конкретным воплощением которой может быть квадрат, треугольник, прямоугольник и т. д. Класс Triangle представляет конкретную разновидность объекта типа TwoDShape, в данном случае – треугольник. Класс Triangle включает в себя все элементы класса TwoDObject, а в дополнение к ним – поле style и методы area() и showStyle(). Описание треугольника хранится в переменной экземпляра style, метод area() вычисляет и возвращает площадь треугольника, а метод showStyle() отображает геометрическую форму треугольника. В класс Triangle входят все члены суперкласса TwoDShape, и поэтому в теле метода area() доступны переменные экземпляра width и height. Кроме того, с помощью объектов tl и t2 в методе main() можно непосредственно обращаться к переменным width и height, как будто они принадлежат классу Triangle.

Члены класса зачастую объявляются закрытыми, чтобы исключить их несанкционированное или незаконное использование. Наследование класса не отменяет ограничения, накладываемые на доступ к закрытым членам класса. Поэтому даже если в подкласс входят все члены его суперкласса, то в нем все равно оказываются недоступными те члены суперкласса, которые являются закрытыми. Так, если сделать закрытыми переменные экземпляра width и height в классе TwoDShape, они станут недоступными в классе Triangle. Для того чтобы члены суперкласса были доступны только самому суперклассу и его подклассам, используется модификатор доступа protected.

1.2.11. Пакеты

Иногда взаимозависимые части программ оказывается удобно объединить в группу. В Java для этой цели предусмотрены пакеты. Во-первых, пакет предоставляет механизм объединения взаимосвязанных частей программы. При обращении к классам, входящим в пакет, указывается имя пакета. Таким образом, пакет предоставляет средства для именования коллекции классов. И во-вторых, пакет является частью механизма управления доступом в Java. Классы могут быть объявлены как закрытые для всех пакетов, кроме того, в который они входят. Следовательно, пакет предоставляет также средства для инкапсуляции классов.

Как правило, при именовании класса для него выделяется имя в пространстве имен. Пространство имен определяет область объявлений. В Java нельзя присваивать двум классам одинаковые имена из одного и того же пространства имен. Иными словами, в пределах пространства имен каждый класс должен иметь уникальное имя. В крупных программах бывает нелегко выбрать уникальное имя для класса. Более того, при использовании библиотек и кода, написанного другими программистами, приходится принимать специальные меры для предотвращения конфликтов имен. Для разрешения подобных затруднений служат пакеты, позволяющие разделить пространство имен на отдельные области. Если класс определен в пакете, то имя пакета присоединяется к имени класса, в результате чего исключается конфликт между двумя классами, имеющими одинаковые имена, но принадлежащими к разным пакетам. Пакет обычно содержит логически связанные классы, и поэтому в Java определены специальные права доступа к содержимому пакета. Так, в пакете можно определить один код, доступный другому коду из того же самого пакета, но недоступный из других пакетов. Это позволяет создавать автономные группы связанных вместе классов и делать операции в них закрытыми.

Каждый класс в Java относится к тому или иному пакету. Если оператор package отсутствует в коде, то используется глобальный пакет, выбираемый по умолчанию. Пакет по умолчанию не имеет имени, что упрощает его применение. Однако ес-

ли пакет по умолчанию подходит для очень простых программ, служащих в качестве примера, то для реальных приложений он малопригоден. Как правило, для разрабатываемого кода приходится определять один или несколько пакетов. Для создания пакета достаточно поместить оператор package в начало файла с исходным кодом программы на Java. В результате классы, определенные в этом файле, будут принадлежать указанному пакету. А поскольку пакет определяет пространство имен, имена классов, содержащихся в файле, войдут в это пространство имен как составные его части. Общая форма оператора package выглядит следующим образом:

```
package имя_пакета;
```

Для управления пакетами в Java используется файловая система, в которой для хранения содержимого каждого пакета выделяется отдельный каталог. Подобно другим именам в Java, имена пакетов зависят от регистра символов. Это означает, что каталог, предназначенный для хранения пакета, должен иметь имя, в точности совпадающее с именем пакета. Пакеты всегда именуются прописными буквами. В разных файлах могут содержаться одинаковые операторы package. Этот оператор лишь определяет, какому именно пакету должны принадлежать классы, код которых содержится в данном файле, и не запрещает другим классам входить в состав того же самого пакета. Как правило, пакеты реальных программ распространяются на большое количество файлов. В Java допускается создавать иерархию пакетов. Для этого достаточно разделить имена пакетов точками.

1.2.12. Интерфейсы

Иногда в объектно-ориентированном программировании полезно определить, что именно должен делать класс, но не то, как он должен это делать. В абстрактном методе определяются возвращаемый тип и сигнатура метода, но не предоставляется его реализация. А в подклассе должна быть обеспечена своя собственная реализация каждого абстрактного метода, определенного в его суперклассе. Таким образом, абстрактный метод определяет интерфейс, но не реализацию метода. Конечно, аб-

страктные классы и методы приносят известную пользу, но положенный в их основу принцип может быть развит далее. В Java предусмотрено разделение интерфейса класса и его реализации с помощью ключевого слова interface.

С точки зрения синтаксиса интерфейсы подобны абстрактным классам. Но в интерфейсе ни у одного из методов не должно быть тела. Это означает, что в интерфейсе вообще не предоставляется никакой реализации. В нем указывается только, что именно следует делать, но не как это делать. Как только интерфейс будет определен, он может быть реализован в любом количестве классов. Кроме того, в одном классе может быть реализовано любое количество интерфейсов. Для реализации интерфейса в классе должны быть предоставлены тела методов, описанных в этом интерфейсе. Каждому классу предоставляется полная свобода для определения деталей своей собственной реализации интерфейса. Следовательно, один и тот же интерфейс может быть реализован в различных классах по-разному. Тем не менее в каждом из них должен поддерживаться один и тот же ряд методов данного интерфейса. А в том коде, где известен такой интерфейс, могут использоваться объекты любого из этих двух классов, поскольку интерфейс для всех этих объектов остается одинаковым. Благодаря поддержке интерфейсов в Java может быть в полной мере реализован главный принцип полиморфизма: «один интерфейс – множество методов».

Интерфейсы определяются с помощью ключевого слова interface. Ниже приведена упрощенная форма объявления интерфейса.

```
доступ interface имя {
возвращаемый_тип имя_метода_1 (список_параметров) ;
возвращаемый_тип имя_метода_2 (список_параметров) ;
тип переменная_1 = значение;
тип переменная_2 = значение;
// . . .
возвращаемый_тип имя_метода_N(список_параметров) ;
тип переменная_N = значение;
}
```

Здесь доступ обозначает тип доступа, который определяется модификатором доступа public или вообще не указывается. Если модификатор доступа отсутствует, применяется правило, предусмотренное по умолчанию, т. е. интерфейс доступен только членам своего пакета. Ключевое слово public указывает на то, что интерфейс может использоваться в любом другом пакете. А имя интерфейса может быть любым допустимым идентификатором. При объявлении методов указываются их сигнатуры и возвращаемые типы. Эти методы являются, по существу, абстрактными. Реализация метода не может содержаться в составе интерфейса. Каждый класс, в определении которого указан интерфейс, должен реализовать все методы, объявленные в интерфейсе. Методы, объявленные в интерфейсе, неявно считаются открытыми (public). Переменные, объявленные в интерфейсе, не являются переменными экземпляра. Они неявно обозначаются ключевыми словами public, final и static и обязательно подлежат инициализации. По существу, они являются константами. Ниже приведен пример определения интерфейса. Предполагается, что этот интерфейс должен быть реализован в классе, где формируется последовательный ряд числовых значений.

```
public interface Series {
int getNext(); // возвратить следующее по порядку число
void reset() ; // начать все с самого сначала
void setStart(int x); // задать начальное значение
}
```

Этот интерфейс объявляется открытым (public), а следовательно, он может быть реализован в классе, принадлежащем любому пакету.

1.3. Обработка исключительных ситуаций

1.3.1. Основные положения обработки исключений

Исключение – это ошибка, возникающая в процессе выполнения программы. Используя подсистему обработки исключений Java, можно контролировать реакцию программы на появление ошибок в ходе ее выполнения. Преимущество обработ-

ки исключений в том, что она автоматически предусматривает реакцию на многие ошибки, избавляя от необходимости писать вручную соответствующий код.

В Java определены стандартные исключения для наиболее часто встречающихся программных ошибок, в том числе деления на нуль или попытки открыть несуществующий файл. Для того чтобы обеспечить требуемую реакцию на конкретную ошибку, в программу следует включить соответствующий обработчик событий. Исключения широко применяются в библиотеке Java API.

В Java все исключения представлены отдельными классами. Все классы исключений являются потомками класса Throwable. Так, если в программе возникнет исключительная ситуация, будет сгенерирован объект класса, соответствующего определенному типу исключения. У класса Throwable имеются два непосредственных подкласса: Exception и Error. Исключения типа Error относятся к ошибкам, возникающим в виртуальной машине Java, а не в прикладной программе. Контролировать такие исключения невозможно, поэтому реакция на них в прикладной программе, как правило, не предусматривается.

Ошибки, связанные с выполнением действий в программе, представлены отдельными подклассами, производными от класса Exception. К этой категории, в частности, относятся ошибки деления на нуль, выхода за границы массива и обращения к файлам. Подобные ошибки следует обрабатывать в самой программе. Важным подклассом, производным от Exception, является класс RuntimeException, который служит для представления различных видов ошибок, часто встречающихся при выполнении программ.

Для обработки исключений в Java предусмотрены пять ключевых слов: try, catch, throw, throws и finally. Они образуют единую подсистему, в которой использование одного ключевого слова почти всегда автоматически влечет за собой употребление другого. Операторы, в которых требуется отслеживать появление исключений, помещаются в блок try. Если в блоке try будет сгенерировано исключение, его можно перехватить и обработать нужным образом. Системные исключения генерируются автоматически. А для того чтобы сгенерировать исключение

вручную, следует воспользоваться ключевым словом throw. Иногда возникает потребность обрабатывать исключения за пределами метода, в котором они возникают, и для этой цели служит ключевое слово throws. Если же некоторый фрагмент кода должен быть выполнен обязательно и независимо от того, возникнет исключение или нет, его следует поместить в блок finally.

1.3.2. Ключевые слова try и catch

Основными языковыми средствами обработки исключений являются ключевые слова try и catch. Они используются совместно. Это означает, что нельзя указать ключевое слово catch в коде, не указав ключевое слово try.

На рис. 1.25 представлена общая форма записи блоков try/catch.

```
try {
// Блок кода, в котором должны отслеживаться ошибки
        }
        catch (тип_исключения_1 объект_исключения) {
// Обработчик исключения тип_исключения_1
        }
        catch (тип_исключения_2 объект_исключения) {
// Обработчик исключения тип_исключения_2
        }
        catch (тип_исключения_3 объект_исключения) {
// Обработчик исключения тип_исключения_3
        }
```

Рис. 1.25. Общая форма записи блоков try/catch

В скобках, следующих за ключевым словом catch, указываются тип исключения и переменная, ссылающаяся на объект данного типа. Когда возникает исключение, оно перехватывается соответствующим оператором catch, обрабатывающим это исключение. Как видно из рис. 1.25, с одним блоком try может быть связано несколько операторов catch. Тип исключения определяет, какой именно оператор catch будет выполняться. Так, если тип исключения соответствует типу оператора catch, то именно он и будет выполнен, а остальные операторы catch –

пропущены. При перехвате исключения переменной, указанной в скобках после ключевого слова catch, присваивается ссылка на «объект_исключения». Если исключение не генерируется, блок try завершается обычным образом и ни один из его операторов catch не выполняется. Выполнение программы продолжается с первого оператора, следующего за последним оператором catch. Таким образом, операторы catch выполняются только при появлении исключения.

Рассмотрим пример, в котором будет обработана ошибка выхода за пределы массива. Попытка обратиться за границы массива приводит к ошибке, и виртуальная машина Java генерирует соответствующее исключение ArraylndexOutOfBoundsException. На рис. 1.26 приведен код программы, демонстрирующей данный пример.

```
public class ExampleException {
    // генерация исключения
    static void genException() {
        int mass[] = new int[4];
        System.out.println("До генерации исключения.");
        // Генерация исключения в связи с обращением
        // за пределы массива
        mass[7]=10;
        System.out.println("Это не отобразится.");

    }
}

class ExampleTest {
    public static void main(String args[]) {
        // Попытка выполнить метод genException
        try {
            ExampleException.genException();
        }
        // Перехват исключения
        catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Индекс не существует!");
        }
        System.out.println("После блока catch");
    }
}
```

Рис. 1.26. Пример обработки исключения выхода за пределы массива

Результат выполнения программы представлен на рис. 1.27.

```
До генерации исключения.
Индекс не существует!
После блока catch

Process finished with exit code 0
```

Рис. 1.27. Результат программы с обработкой исключения

В данном примере показаны несколько важных особенностей обработки исключений:

1) код, подлежащий проверке на наличие ошибок, помещается в блок try;

2) когда возникает исключение, выполнение блока try прерывается и управление получает блок catch. Следовательно, явного обращения к блоку catch не происходит, но переход к нему осуществляется лишь при определенном условии, возникающем в ходе выполнения программы.

Так, оператор вызова метода println(), следующий за выражением, в котором происходит обращение к несуществующему элементу массива, вообще не выполняется. По завершении блока catch выполнение программы продолжается с оператора, следующего за этим блоком. Таким образом, обработчик исключений предназначен для устранения программных ошибок, приводящих к исключительным ситуациям, а также для обеспечения нормального продолжения исполняемой программы.

Одно из главных преимуществ обработки исключений в том, что она позволяет вовремя отреагировать на ошибку в программе и затем продолжить ее выполнение. Рассмотрим другой пример, в котором элементы одного массива делятся на другой. Если при этом происходит деление на ноль, то генерируется исключение ArithmeticException. Обработка подобного исключения состоит в том, что программа уведомляет об ошибке и затем продолжает свое выполнение (рис. 1.28).

```
class ExampleDel {
    public static void main(String args[]) {
        // Объявление массивов
        int mass1[] = {8,16,24,32,64,128};
        int mass2[] = {2,0,4,0,8,16};
        // Цикл, в котором осуществляется деление
        for(int i=0;i<mass1.length;i++) {
            // Попытка поделить одно число на другое
            try {
                System.out.println(mass1[i] + " / " + mass2[i] + " = " + mass1[i]/mass2[i]);
            }
            // Перехват исключения
            catch(ArithmeticException e) {
                System.out.println("На ноль делить нельзя!");
            }
        }

    }
}
```

Рис. 1.28. Программа, осуществляющая деление элементов двух массивов

В результате получим итог, представленный на рис. 1.29.

```
8 / 2 = 4
На ноль делить нельзя!
24 / 4 = 6
На ноль делить нельзя!
64 / 8 = 8
128 / 16 = 8

Process finished with exit code 0
```

Рис. 1.29. Результат выполнения программы

Данный пример демонстрирует еще одну важную особенность: обработанное исключение удаляется из системы. Иными словами, на каждом шаге цикла блок try выполняется в программе заново, а все возникшие ранее исключения считаются обработанными. Благодаря этому в программе могут обрабатываться повторяющиеся ошибки.

Как пояснялось ранее, с блоком try можно связать несколько операторов catch. Обычно разработчики так и поступают на практике. Каждый из операторов catch должен перехватывать отдельный тип исключений. Выражения с операторами

catch проверяются в том порядке, в котором они встречаются в программе, и выполняется лишь тот из них, который соответствует типу возникшего исключения. Остальные блоки операторов catch просто игнорируются.

1.3.3. Генерация исключений

Генерировать исключения можно вручную, используя для этого оператор throw. Общая форма этого оператора имеет следующий вид:

```
throw объект_исключения;
```

где `объект_исключения` должен быть объектом класса, производного от класса Throwable. На рис. 1.30 представлен пример программы, где исключение ArithmeticException генерируется вручную.

```
class ThrowExample {
    public static void main(String args[]) {
        try {
            System.out.println("До применения оператора throw.");
            // Генерация исключения
            throw new ArithmeticException();
        }
        catch (ArithmeticException exc) {
// перехватить исключение
            System.out.println("Исключение обработано.");
        }
        System.out.println("После блока try/catch.");
    }
}
```

Рис. 1.30. Пример программы с ручной генерацией исключения

Выполнение этой программы дает результат, представленный на рис. 1.31.

Исключение ArithmeticException генерируется с помощью ключевого слова new в операторе throw. Оператор throw генерирует исключение в виде объекта. Поэтому после ключевого слова throw недостаточно указать только тип исключения, нужно еще создать объект для этой цели.

```
До применения оператора throw.
Исключение обработано.
После блока try/catch.

Process finished with exit code 0
```

Рис. 1.31. Результат выполнения программы с ручной генерацией исключения

Исключение, перехваченное блоком catch, может быть повторно сгенерировано для обработки другим аналогичным блоком. Чаще всего повторное генерирование исключений применяется с целью предоставить разным обработчикам доступ к исключению. Например, повторное генерирование имеет смысл в том случае, если один обработчик оперирует одним свойством исключения, а другой обработчик ориентирован на другое его свойство. Повторно сгенерированное исключение не может быть перехвачено тем же самым блоком catch. Оно распространяется в другие блоки catch.

1.3.4. Ключевые слова finally и throws

Иногда требуется определить кодовый блок, который должен выполняться по завершении блока try/catch. Допустим, в процессе работы программы возникло исключение, требующее ее преждевременного завершения. Но в программе открыт файл или установлено сетевое соединение, а следовательно, файл нужно закрыть, а соединение разорвать. Для выполнения подобных операций нормального завершения программы необходимо использовать блок finally. Для того чтобы определить код, который должен выполняться по завершении блока try/catch, нужно указать блок finally в конце последовательности операторов try/catch. На рис. 1.32 приведена общая форма записи блока try/catch вместе с блоком finally.

Блок finally выполняется всегда по завершении блока try/catch независимо от того, какое именно условие к этому привело. Следовательно, блок finally получит управление как при нормальной работе программы, так и при возникновении ошибки. Более того, он будет вызван даже в том случае, если в блоке

try или в одном из блоков catch будет присутствовать оператор return для немедленного возврата из метода.

```
try {
// Блок кода, в котором отслеживаются ошибки.
}
catch (тип_исключения_1 объект_исключения) {
// Обработчик исключения тип_исключения_1
}
catch (тип_исключения_2 объект_исключения) {
// Обработчик исключения тип_исключения_2
    }
// . . .
finally {
// Код блока finally
    }
```

Рис. 1.32. Общая форма записи блоков обработки исключений

Иногда исключения нецелесообразно обрабатывать в том методе, в котором они возникают. В таком случае их следует указывать с помощью ключевого слова throws. На рис. 1.33 приведена общая форма объявления метода, в котором присутствует ключевое слово throws.

```
возвращаемый_тип имя_метода(список_параметров) throws список_исключений {
// Тело метода
}
```

Рис. 1.33. Общая форма метода
с использованием ключевого слова throws

В списке исключений через запятую указываются исключения, которые может генерировать метод. Кроме того, исключения, генерируемые подклассом Error или RuntimeException, можно и не указывать в списке оператора throws. Исполняющая система Java по умолчанию предполагает, что метод может их генерировать. А исключения всех остальных типов следует непременно объявить с помощью ключевого слова throws. Если этого не сделать, возникнет ошибка при компиляции.

1.4. Работа с файлами

1.4.1. Организация ввода-вывода в Java. Байтовые потоки

Ввод-вывод в программах на Java осуществляется посредством потоков. *Поток* – это некая абстракция производства или потребления информации. С физическим устройством поток связывает система ввода-вывода. Все потоки действуют одинаково, даже если они связаны с разными физическими устройствами. Поэтому классы и методы ввода-вывода могут применяться к самым разным типам устройств. Например, методами вывода на консоль можно пользоваться и для вывода в файл на диске. Для реализации потоков используется иерархия классов, содержащихся в пакете java.io.

В современных версиях Java определены два типа потоков: байтовые и символьные. Байтовые потоки предоставляют удобные средства для ввода и вывода байтов. Они используются, например, при чтении и записи двоичных данных. В особенности они полезны для обращения с файлами. А символьные потоки ориентированы на обмен символьными данными. В них применяется кодировка в уникоде (Unicode), а следовательно, программы, в которых используются символьные потоки, легко поддаются локализации на разные языки мира. В некоторых случаях символьные потоки обеспечивают более высокую эффективность по сравнению с байтовыми. Необходимость поддерживать два разных типа потоков ввода-вывода привела к созданию двух иерархий классов. Следует иметь в виду, что на самом нижнем уровне все средства ввода-вывода имеют байтовую организацию. А символьные потоки лишь предоставляют удобные и эффективные инструменты для обработки символов.

На вершине иерархии байтовых потоков находятся классы InputStream и OutputStream. При возникновении ошибок в процессе выполнения методы из классов InputStream и OutputStream могут генерировать исключения типа lOException. Методы, определенные в этих двух абстрактных классах, доступны во всех подклассах. Таким образом, они формируют минимальный

набор функций ввода-вывода, общих для всех байтовых потоков.

Поток System.in является экземпляром класса InputStream, и благодаря этому обеспечивается автоматический доступ к методам, определенным в классе InputStream. К сожалению, в классе InputStream определен только один метод ввода – read(), предназначенный для чтения байтов. Ниже приведены разные формы объявления этого метода.

```
int read () throws IOException
int read(byte data[]) throws IOException
int read(byte data[], int start, int max) throws IOException
```

Чтение завершается по достижении конца потока, по заполнении массива или при возникновении ошибки. Метод read() возвращает количество прочитанных байтов или значение –1, если достигнут конец потока. Третья форма данного метода позволяет разместить прочитанные данные в массиве data начиная с элемента, обозначаемого индексом start. Максимальное количество байтов, которые могут быть введены в массив, определяется параметром max. При возникновении ошибки в каждой из этих форм метода read() генерируется исключение IOException. Условие конца потока ввода System.in устанавливается при нажатии клавиши Enter.

System.out является байтовым потоком вывода и по-прежнему широко используется для побайтового вывода данных на консоль. Вывести данные на консоль проще всего с помощью уже знакомых нам методов print() и println(). Эти методы определены в классе PrintStream. Несмотря на то что System.out является байтовым потоком вывода, пользоваться им вполне допустимо для организации элементарного вывода данных на консоль. Класс PrintStream представляет собой выходной поток, производный от класса OutputStream, и поэтому в нем также реализуется метод write() низкоуровневого вывода. Следовательно, этот метод может быть использован для вывода данных на консоль. Самая простая форма метода write(), определенного в PrintStream, имеет следующий вид:

```
void write(int byteval)
```

Этот метод записывает в поток байтовое значение, указываемое в качестве параметра byteval. Несмотря на то что этот параметр объявлен как int, учитываются только 8 младших битов его значения.

1.4.2. Чтение и запись в файлы из байтовых потоков

В Java предоставляется большое количество классов и методов, позволяющих читать и записывать данные в файлы. В Java все файлы имеют байтовую организацию, и поэтому для побайтового чтения и записи данных в такие файлы предоставляются соответствующие методы. Кроме того, для байтовых потоков ввода-вывода в файлы в Java разрешено создавать специальные оболочки в виде символьных объектов. Для того чтобы создать байтовый поток и связать его с файлом, следует воспользоваться классом FileInputStream или FileOutputStream. А для открытия файла достаточно создать объект одного из этих классов, передав имя файла конструктору в качестве параметра. В открытый файл можно записывать данные или читать их из него.

Файл открывается для ввода созданием объекта типа FileInputStream. Для этой цели чаще всего используется приведенная ниже форма объявления конструктора данного класса.

```
FileInputStream(String имя_файла) throws FileNotFoundException
```

В качестве параметра этому конструктору передается `имя_файла`, который требуется открыть. Если указанный файл не существует, генерируется исключение FileNotFoundException.

Для чтения данных из файла служит метод read(). Ниже приведена форма объявления этого метода, которой мы будем пользоваться в дальнейшем.

```
int read() throws IOException
```

При каждом вызове метод read() читает байт из файла и возвращает его как целочисленное значение. По достижении конца файла этот метод возвращает значение –1. При возникновении ошибки метод генерирует исключение IOException.

В этой форме метод read() выполняет те же самые действия, что и одноименный метод для ввода данных с консоли.

Завершив операции с файлом, следует закрыть его с помощью метода close(), общая форма объявления которого выглядит следующим образом:

```
void close () throws IOException
```

При закрытии файла освобождаются связанные с ним системные ресурсы, чтобы использовать их для работы с другим файлом. Если же файл не будет закрыт, могут произойти утечки памяти из-за того, что часть памяти остается выделенной для неиспользуемых ресурсов.

Для того чтобы открыть файл для вывода, следует создать объект типа FileOutputStream. Ниже приведены два наиболее часто употребляемых конструктора этого класса:

```
FileOutputStream{String имя_файла) throws FileNotFoundException
FileOutputStream(String имя_файла, boolean append) throws FileNotFoundException
```

Если файл не может быть создан, возникает исключение FileNotFoundException. В первой форме конструктора при открытии файла удаляется существовавший ранее файл с таким именем. Вторая форма отличается наличием параметра append. Если этот параметр принимает логическое значение true, записываемые данные добавляются в конец файла. В противном случае старые данные в файле перезаписываются новыми.

Для того чтобы записать данные в файл, следует вызвать метод write(). Наиболее простая форма этого метода приведена ниже:

```
void write(int byteval) throws IOException
```

Этот метод записывает в поток байтовое значение, указанное в качестве параметра byteval. Несмотря на то что этот параметр объявлен как int, учитываются только 8 младших битов его значения. Если в процессе записи возникнет ошибка, будет сгенерировано исключение IOException.

По завершении работы с файлом его нужно закрыть с помощью метода close(). Объявление этого метода выглядит следующим образом:

```
void close () throws IOException
```

При закрытии файла освобождаются связанные с ним системные ресурсы, которые могут быть использованы для работы с другим файлом. Процедура закрытия файла также гарантирует, что данные, оставшиеся в буфере, будут записаны на диск.

1.4.3. Символьные потоки в Java

На вершине иерархии классов, поддерживающих символьные потоки, находятся абстрактные классы Reader и Writer. Методы, определенные в указанных абстрактных классах Reader и Writer, доступны во всех их подклассах. Таким образом, они образуют минимальный набор функций ввода-вывода, необходимых для всех символьных потоков.

Наиболее подходящим для ввода с консоли является класс BufferedReader, поддерживающий буферизованный поток ввода. Но объект типа BufferedReader нельзя построить непосредственно из потока стандартного ввода System.in. Сначала нужно преобразовать байтовый поток в символьный. И для этой цели служит класс InputStreamReader, преобразующий байты в символы. Для того чтобы получить объект типа InputStreamReader, связанный с потоком стандартного ввода System.in, нужно воспользоваться следующим конструктором:

```
InputStreamReader(InputStream inputStream)
```

Поток ввода System.in является экземпляром класса InputStream, и поэтому его можно указать в качестве параметра inputStream данного конструктора. Затем на основании объекта типа InputStreamReader можно создать объект типа BufferedReader, используя следующий конструктор:

```
BufferedReader(Reader inputReader)
```

где inputReader – это поток, который связывается с создаваемым экземпляром класса BufferedReader. Объединяя обращения

к указанным выше конструкторам в одну операцию, мы получаем приведенную ниже строку кода. В ней создается объект типа BufferedReader, связанный с клавиатурой.

```
BufferedReader br = new BufferedReader (new InputStreamReader(System.in));
```

После выполнения этого оператора присваивания переменная br будет содержать ссылку на символьный поток, связанный с консолью через поток ввода System.in.

Прочитать символы из потока ввода System.in можно с помощью метода read(), определенного в классе BufferedReader. Ниже приведены общие формы объявления трех вариантов метода read(), предусмотренных в классе BufferedReader.

```
int read () throws IOException
int read(char data[]) throws IOException
int read(char data[], int start, int max) throws IOException
```

В первом варианте метод read() читает один символ в уникоде. По достижении конца потока этот метод возвращает значение –1. Во втором варианте метод read() читает данные из потока ввода и помещает их в массив. Чтение оканчивается по достижении конца потока, по заполнении массива data символами или при возникновении ошибки. В этом случае метод возвращает число прочитанных символов, а если достигнут конец потока – значение –1. В третьем варианте метод read() помещает прочитанные символы в массив data начиная с элемента, определяемого индексом start. Максимальное число символов, которые могут быть записаны в массив, определяется параметром max. В данном случае метод возвращает число прочитанных символов или значение –1, если достигнут конец потока. При возникновении ошибки в каждом из перечисленных выше вариантов метода read() генерируется исключение IOException. При чтении данных из потока ввода System.in конец потока устанавливается нажатием клавиши Enter.

Для ввода символьной строки с клавиатуры следует воспользоваться методом readLine() из класса BufferedReader. Ниже приведена общая форма объявления этого метода:

```
string readLine() throws IOException
```

Этот метод возвращает объект типа String, содержащий прочитанные символы. При попытке прочитать символьную строку по окончании потока метод возвращает пустое значение null.

На практике чаще всего приходится обращаться с файлами, имеющими байтовую организацию, тем не менее для этой цели можно пользоваться символьными потоками. Преимущество символьных потоков заключается в том, что они оперируют непосредственно символами в юникоде. Так, если требуется сохранить текст в юникоде, для этой цели лучше всего воспользоваться символьными потоками. Как правило, для файлового ввода-вывода символов служат классы FileReader и FileWriter. Класс FileWriter представляет поток, через который можно осуществлять запись данных в файл. В классе FileReader создается объект типа Reader, который можно использовать для чтения содержимого файла.

1.5. Графический пользовательский интерфейс

1.5.1. Основы библиотеки Swing

Консольные приложения, рассматриваемые нами ранее, прекрасно подходят для изучения основ языка Java. Однако на практике чаще всего разрабатываются приложения с графическим интерфейсом (GUI). Библиотека Swing представляет собой коллекцию классов и интерфейсов для создания визуальных компонентов, таких как кнопки, полосы прокрутки, таблицы и т. д.

В более ранних версиях Java средства Swing отсутствовали, но была оконная подсистема – библиотека AWT (Abstract Window Toolkit). В библиотеке AWT определен базовый набор компонентов для создания пользовательских интерфейсов с ограниченными возможностями. Одним из ограничений AWT стало превращение ее визуальных компонентов в платформенно-зависимые эквиваленты. Это означает, что внешний вид компонентов AWT определялся не средствами Java, а используемой платформой. А поскольку в компонентах AWT применялся в качестве ресурса собственный код, то они назывались тяжеловесными. Применение собственных равноправных компонентов

послужило причиной целого ряда осложнений. Во-первых, из-за отличий в операционных системах компоненты по-разному выглядели и даже иначе вели себя на разных платформах. Это было прямым нарушением главного принципа Java: код, написанный один раз, должен работать везде. Во-вторых, внешний вид каждого компонента был фиксированным, и изменить его было очень трудно. И в-третьих, на применение тяжеловесных компонентов накладывался целый ряд обременительных ограничений. В частности, такие компоненты всегда имели прямоугольную форму и были непрозрачны.

В результате определенных доработок в 1997 г. появилась библиотека компонентов Swing, включенная в состав библиотеки классов JFC (Java Foundation Classes). Swing устраняет ограничения, присущие компонентам AWT, благодаря использованию двух основных средств: легковесных компонентов и подключаемых стилей оформления. Эти средства составляют основные принципы конструирования Swing и в значительной мере обусловливают возможности и удобство применения этой библиотеки.

Все компоненты Swing, за небольшим исключением, являются легковесными, а это означает, что они полностью написаны на Java и не зависят от конкретной платформы, поскольку не опираются на платформенно-зависимые равноправные компоненты. Легковесные компоненты обладают рядом существенных преимуществ, к числу которых относятся эффективность и гибкость. Каждому пользователю Swing изначально доступны различные стили оформления, к числу которых относятся металлический (Metal) и Motif. Металлический стиль называется также стилем оформления Java. Это платформенно-независимый стиль оформления, доступный во всех средах выполнения программ на Java. Реализация подключаемых стилей оформления в Swing стала возможной благодаря тому, что при создании Swing был использован видоизмененный вариант классической архитектуры «модель – представление – контроллер» (MVC). В терминологии MVC модель определяет состояние компонента. В Swing применяется вариант MVC, в котором представление и контроллер объединены в единую логическую сущность, называемую представителем пользовательского интерфейса.

В связи с этим принятый в Swing подход называется архитектурой «модель – представитель», а иначе – архитектурой с разделенной моделью. Таким образом, компоненты Swing нельзя рассматривать как классическую реализацию архитектуры MVC, хотя их архитектура и опирается на нее.

В состав графического пользовательского интерфейса, создаваемого средствами Swing, входят две основные разновидности элементов: компоненты и контейнеры. Такое разделение во многом условно, поскольку контейнеры являются в то же время компонентами, а отличаются от них своим назначением. Компонент – это независимый элемент (например, кнопка или поле ввода текста), а контейнер может содержать в себе несколько компонентов. Для того чтобы отобразить компонент на экране, его следует поместить в контейнер. Поэтому в графическом пользовательском интерфейсе должен присутствовать хотя бы один контейнер. А так как контейнеры являются в то же время компонентами, то один контейнер может содержать в себе другой. Это дает возможность сформировать так называемую иерархию контейнеров, на вершине которой должен находиться контейнер верхнего уровня.

Подавляющее большинство компонентов Swing, кроме четырех контейнеров верхнего уровня, создаются с помощью классов, производных от класса JComponent. В классе JComponent реализуются функциональные возможности, общие для всех компонентов, такие как поддержка подключаемых стилей. Этот класс наследует свойства классов Container и Component из библиотеки AWT. Таким образом, компоненты Swing создаются на основе компонентов AWT и совместимы с ними. Классы, представляющие все компоненты Swing, входят в пакет javax.swing.

В Swing определены две разновидности контейнеров. К первой из них относятся следующие контейнеры верхнего уровня: JFrame, JApplet, JWindow и JDialog. Они не наследуют переменные и методы от класса JComponent, тем не менее являются производными от классов Component и Container. В отличие от других, легковесных компонентов Swing, контейнеры верхнего уровня являются тяжеловесными. Именно поэтому контейнеры верхнего уровня составляют отдельную группу в библиотеке Swing. Как следует из названия контейнеров верх-

него уровня, они должны находиться на вершине иерархии контейнеров и не могут содержаться в других контейнерах. Более того, любая иерархия должна начинаться именно с контейнера верхнего уровня. В прикладных программах чаще всего используется контейнер типа JFrame, а в апплетах – контейнер типа JApplet. Контейнеры второго рода являются легковесными и производными от класса JComponent. В качестве примера легковесных контейнеров можно привести классы JPanel, JScrollPane и JRootPane. Легковесные контейнеры могут содержаться в других контейнерах, и поэтому они нередко используются для объединения группы взаимосвязанных компонентов.

1.5.2. Менеджеры компоновки

Менеджер компоновки управляет размещением компонентов в контейнере. В Java определено несколько таких диспетчеров. Большинство из них входит в состав AWT, но в Swing также предоставляется ряд дополнительных диспетчеров компоновки. Все менеджеры компоновки являются экземплярами классов, реализующих интерфейс LayoutManager. В табл. 1.2 перечислены основные менеджеры компоновки.

Таблица 1.2

Менеджеры компоновки

Название	Характеристика
FlowLayout	Располагает компоненты слева направо и сверху вниз
BorderLayout	Располагает компоненты по центру и по краям контейнера. Принимается по умолчанию
GridLayout	Располагает компоненты сеткой, как в таблице
GridBagLayout	Располагает компоненты разных размеров настраиваемой сеткой, как в таблице
BoxLayout	Располагает компоненты в блоке по вертикали или по горизонтали
SpringLayout	Располагает компоненты с учетом ряда ограничений

Рассмотрим два менеджера компоновки: BorderLayout и FlowLayout. Для панели содержимого по умолчанию принимается менеджер компоновки BorderLayout. Этот менеджер компоновки определяет в составе контейнера пять областей, в которые могут помещаться компоненты. Первая область располага-

ется посередине и называется центральной. Остальные четыре размещаются по сторонам света и соответственно называются северной, южной, восточной и западной. По умолчанию компонент, добавляемый на панели содержимого, располагается в центральной области. Для того чтобы расположить компонент в другой области, следует указать ее имя. Несмотря на то что возможностей, предоставляемых менеджером компоновки BorderLayout, зачастую оказывается достаточно, иногда возникает потребность в других. К числу самых простых относится менеджер компоновки FlowLayout. Он размещает компоненты построчно: слева направо и сверху вниз. Заполнив текущую строку, он переходит к следующей. Такая компоновка предоставляет лишь ограниченный контроль над расположением компонентов, хотя и проста в употреблении.

1.5.3. Создание простого оконного приложения

Swing-программы не просто настраивают применение компонентов Swing, обеспечивающих взаимодействие с пользователем, но и должны отвечать особым требованиям, связанным с организацией поточной обработки. Рассматриваемый ниже пример наглядно демонстрирует один из приемов написания Swing-приложений. В данной программе (рис. 1.34) используются два компонента Swing: классы JFrame и JLabel. Класс JFrame представляет собой контейнер верхнего уровня, нередко применяемый в Swing-приложениях, а класс JLabel – компонент Swing, с помощью которого создается метка, используемая для отображения информации. Метка является самым простым компонентом Swing, поскольку она не реагирует на действия пользователя, а только помечает отображаемую информацию. Контейнер JFrame служит для размещения экземпляра компонента JLabel. С помощью метки отображается короткое текстовое сообщение.

Результат выполнения программы представлен на рис. 1.35.

```
package com.company;
import javax.swing.*;

class SwingDemo {
    SwingDemo() {
// Создание нового контейнера JFrame.
        JFrame jfrm = new JFrame( title: "Простое диалоговое окно");
// Установка начальных размеров рамки окна
        jfrm.setSize ( width: 375,  height: 120);
// При закрытии окна программа должна завершиться
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
// Создание текстовой метки с помощью компонента Swing
        JLabel jlab = new JLabel( text: " Демонстрация работы элементов GUI");
// Добавление метки на панели содержимого
        jfrm.add(jlab);
// Отображение рамки окна
        jfrm.setVisible(true);
    }
    public static void main(String args[]) {
// Объект SwingDemo должен быть создан в потоке
// диспетчеризации событий
        SwingUtilities.invokeLater(new Runnable() {
            public void run () {
                new SwingDemo();
                    }
                }) ;
    }
}
```

Рис. 1.34. Листинг программы простого оконного приложения

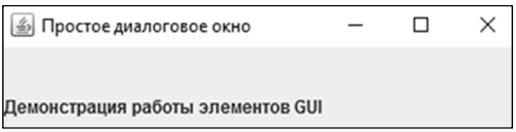

Рис. 1.35. Выполнение программы с применением GUI

Разберем особенности написания кода программы SwingDemo. В начале программы осуществляется импорт пакета javax.swing. Этот пакет содержит компоненты и модели Swing. В частности, в нем определены классы, реализующие метки, кнопки, поля ввода текста и меню. Этот пакет должен быть непременно включен в каждую программу, использующую библиотеку Swing. Далее в программе объявляется класс SwingDemo и его конструктор, в котором выполняется большая

часть действий программы. Код конструктора начинается с создания объекта типа JFrame – jfrm, который определяет прямоугольное окно, содержащее строку заголовка, кнопки сворачивания, разворачивания и закрытия, а также системное меню. Далее задаются размеры окна в пикселях. По умолчанию, когда окно верхнего уровня закрывается (например, по щелчку на кнопке в его верхнем правом углу), оно удаляется с экрана, но приложение не завершает на этом свою работу. Чтобы приложение завершилось, применяется метод setDefaultCloseOperation(). Помимо константы JFrame.EXIT_ON_CLOSE, данному методу можно передавать константы: JFrame.DISPOSE_ON_CLOSE, JFrame.HIDE_ON_CLOSE, JFrame.DO_NOTHING_ON_CLOSE. Имена констант отражают их назначение: освободить, скрыть, ничего не делать после закрытия окна соответственно. Все они определены в интерфейсе WindowConstants, входящем в пакет javax.swing. Далее создается компонент JLabel из библиотеки Swing. Компонент JLabel – самый простой в использовании среди всех компонентов Swing, поскольку он не предполагает обработку событий, связанных с действиями пользователя, а только отображает информацию: текст, изображение или и то и другое. Затем, в следующей строке кода метка вводится на панели содержимого рамки окна. Каждый контейнер верхнего уровня включает в себя панель содержимого, на которой располагаются компоненты. Поэтому для размещения компонента внутри рамки окна его следует добавить на панели содержимого. С этой целью вызывается метод add() из класса JFrame (в данном случае ссылка на объект типа JFrame содержится в переменной jfrm). Последний оператор в конструкторе класса SwingDemo обеспечивает отображение окна.

В методе main() создается объект класса SwingDemo, в результате чего окно и метка отображаются на экране. Объект типа SwingDemo создается не в основном потоке приложения, а в потоке диспетчеризации событий. Такое решение принимается по ряду причин. Прежде всего, Swing-программы, как правило, управляются событиями. Так, если пользователь активизирует компонент пользовательского интерфейса, формируется соответствующее событие. Оно передается прикладной программе путем вызова обработчика событий, определенного в этой

программе. Но этот обработчик выполняется в специальном потоке диспетчеризации событий, формируемом средствами Swing, а не в главном потоке прикладной программы. Таким образом, поток, в котором выполняется этот обработчик событий, создается другими средствами, хотя все обработчики событий определяются в самой программе. Во избежание осложнений, связанных, например, с попытками двух потоков одновременно обновить один и тот же компонент, все компоненты пользовательского интерфейса из библиотеки Swing должны создаваться и обновляться не в основном потоке приложения, а в потоке диспетчеризации событий. Но метод main() выполняется в основном потоке, и поэтому в нем нельзя непосредственно создавать объект класса SwingDemo. Сначала следует построить объект типа Runnable, выполняемый в потоке диспетчеризации событий, а затем предоставить ему возможность построить графический пользовательский интерфейс. Для того чтобы активизировать код построения графического пользовательского интерфейса в потоке диспетчеризации событий, следует воспользоваться одним из двух методов, определенных в классе SwingUtilities: invokeLater() или invokeAndWait().

1.5.4. Компонент JButton

Нажимаемая кнопка представлена в Swing экземпляром класса JButton. Этот класс является производным от абстрактного класса AbstractButton, в котором определены функции общие для всех кнопок. На кнопке может отображаться текст надписи, изображение или то и другое. Класс JButton содержит три конструктора. Один из них имеет следующий вид:

```
JButton(String сообщение)
```

Параметр сообщение определяет символьную строку, которая должна отображаться в виде надписи на кнопке. После щелчка на кнопке формируется событие ActionEvent. Класс ActionEvent определен в библиотеке AWT, но используется также в библиотеке Swing. В классе JButton предоставляются методы, позволяющие зарегистрировать приемник событий или отменить его регистрацию:

```
void addActionListener(ActionListener al)
void removeActionListener(ActionListener al)
```

где параметр al задает объект, который будет уведомляться о наступлении событий. Объект должен представлять собой экземпляр класса, реализующего интерфейс ActionListener. В интерфейсе ActionListener определен только один метод: actionPerformed(). Данный метод вызывается при щелчке на кнопке. Следовательно, в этом методе осуществляется обработка событий, связанных с действиями пользователя над кнопкой. Реализуя метод actionPerformed(), необходимо позаботиться о том, чтобы он быстро выполнял свои функции и возвращал управление. С помощью объекта типа ActionEvent, передаваемого методу actionPerformed(), можно получить важные сведения о событии, связанном с щелчком на кнопке. Получить команду действия можно, вызвав метод getActionCommand() для объекта события, который объявляется следующим образом:

```
string getActionCommand()
```

Команда действия идентифицирует кнопку. Когда в пользовательском интерфейсе приложения имеется несколько кнопок, команда действия позволяет достаточно просто определить, какая из них была выбрана.

На рис. 1.36 представлен пример программы, демонстрирующий использование кнопки, а на рис. 1.37 результат выполнения этой программы.

Пакет java.awt необходим, поскольку он содержит класс диспетчера компоновки FlowLayout, а пакет java.awt.event потому, что в нем определены интерфейс ActionListener и класс ActionEvent. Далее в программе объявляется класс ButtonDemo, который реализует интерфейс ActionListener. Это означает, что объекты типа ButtonDemo могут быть использованы для приема и обработки событий действия. Затем объявляется ссылка на объект типа JLabel. Она будет использована в методе actionPerformed() для отображения сведений о том, какая именно кнопка была нажата. Конструктор класса ButtonDemo начинается с создания контейнера jfrm типа JFrame. Затем в качестве диспетчера компоновки для панели содержимого контейнера jfrm устанавливается FlowLayout, как показано ниже.

```
package com.company;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
class ButtonDemo implements ActionListener {
    JLabel jlab;
    ButtonDemo() {
        JFrame jfrm = new JFrame( title: "Столица Великобритании?");
        jfrm.setLayout(new FlowLayout () ) ;
        jfrm.setSize( width: 260,  height: 120);
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // Создание двух кнопок
        JButton jbtnL = new JButton( text: "London");
        JButton jbtnP = new JButton( text: "Paris");
        // Добавление приемников событий
        jbtnL.addActionListener( l: this) ;
        jbtnP.addActionListener( l: this) ;
        // Добавление кнопок на панели содержимого
        jfrm.add(jbtnL);
        jfrm.add(jbtnP);
        // создать метку
        jlab = new JLabel( text: "Нажми на кнопку");
        // добавить метку в рамке окна
        jfrm.add(jlab);
        // отобразить рамку окна
        jfrm.setVisible(true);    }
    // Обработка событий от кнопки
    public void actionPerformed(ActionEvent ae) {
    // Для определения нажатой кнопки используется команда действия
        if(ae.getActionCommand().equals("London"))
            jlab.setText("Правильный ответ!");
        else
            jlab.setText("Почитайте учебник географии...");
    }
    public static void main(String args[]) {
// создать рамку окна в потоке диспетчеризации событий
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new ButtonDemo();            }
        });    } }
```

Рис. 1.36. Программа с кнопками

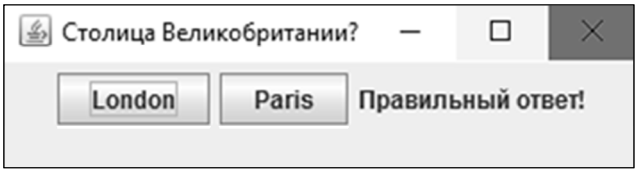

Рис. 1.37. Результат выполнения программы с кнопками

По умолчанию на панели содержимого в качестве диспетчера компоновки используется BorderLayout, но для многих приложений больше подходит диспетчер компоновки FlowLayout. После установки размеров рамки окна и определения операции, завершающей программу, в конструкторе ButtonDemo() создаются две кнопки. На первой кнопке отображается надпись London, а на второй – Paris. Далее с кнопками связывается приемник событий действия, в роли которого выступает экземпляр класса ButtonDemo, а ссылка на него передается с помощью ключевого слова this. В результате объект, создающий кнопки, будет также получать уведомление об их нажатии. Всякий раз, когда кнопка нажимается, формируется событие, о котором зарегистрированные приемники уведомляются в результате вызова метода actionPerformed(). Объект типа ActionEvent, представляющий событие от манипуляции с кнопкой, передается этому методу в качестве параметра. Для передачи события служит параметр ae. В теле метода извлекается команда действия, которая соответствует кнопке, сформировавшей событие. Для получения команды действия вызывается метод getActionCommand(). В зависимости от содержания символьной строки, представляющей команду действия, устанавливается текст надписи на кнопке. Следует также иметь в виду, что метод actionPerformed() вызывается в потоке диспетчеризации событий. Он должен завершаться как можно быстрее, чтобы не замедлять работу приложения.

Контрольные вопросы

1. Каковы три основных принципа объектно-ориентированного программирования?

2. С чего начинается выполнение программы на Java?

3. Что такое переменная?

4. Как создать однострочный комментарий? Как создать многострочный комментарий?

5. Как выглядит общая форма условного оператора if? Как выглядит общая форма цикла for?

6. Как создать кодовый блок?

7. Если при вводе кода программы вы допустите опечатку, то какого рода сообщение об ошибке получите?

8. Имеет ли значение, в каком именно месте строки находится оператор?

9. Почему в Java строго определены диапазоны допустимых значений и области действия простых типов?

10. Что собой представляет символьный тип в Java и чем он отличается от символьного типа в ряде других языков программирования?

11. Переменная типа boolean может иметь любое значение, поскольку любое ненулевое значение интерпретируется как истинное. Верно ли это?

12. К какому типу приводятся типы byte и short при вычислении выражения?

13. Когда возникает потребность в явном приведении типов?

14. Оказывают ли лишние скобки влияние на эффективность выполнения программ?

15. Определяет ли кодовый блок область действия переменных?

16. Какова общая форма многоступенчатой конструкции if – else – if?

17. Допустим, имеется следующий фрагмент кода:

```
if(x < 10)
if(y 100) {
if(!done) x = z;
```

```
else y = z;
}
else System.out.println("error"); // что если?
```

С каким из операторов if связан последний оператор else?

18. Напишите цикл for, в котором перебирались бы значения от 1000 до 0 с шагом 2.

19. Какие действия выполняет оператор break? Опишите оба варианта этого оператора.

20. В чем отличие класса от объекта?

21. Как определяется класс?

22. Чью собственную копию содержит каждый объект?

23. Покажите, как объявить объект counter класса MyCounter, используя два отдельных оператора.

24. Как должен быть объявлен метод myMeth, принимающий два параметра, a и b, типа int и возвращающий значение типа double?

25. Как должно завершаться выполнение метода, возвращающего некоторое значение?

26. Каким должно быть имя конструктора?

27. Какие действия выполняет оператор new?

28. Что означает ключевое слово this?

29. Может ли конструктор иметь параметры?

30. Если метод не возвращает значения, то как следует объявить тип этого метода?

31. Допустим, все объекты класса должны совместно пользоваться одной и той же переменной. Как объявить такую переменную?

32. Для чего может понадобиться статический блок?

33. Что такое внутренний класс?

34. Допустим, требуется член класса, к которому могут обращаться только другие члены этого же класса. Какой модификатор доступа следует использовать в его объявлении?

35. Какой класс находится на вершине иерархии исключений?

36. Что произойдет, если исключение не будет перехвачено?

37. Исключения какого типа необходимо явно объявлять с помощью оператора throws, включаемого в объявление метода?

38. Какими тремя способами можно сгенерировать исключение?

39. Компоненты AWT являются тяжеловесными. А какими являются компоненты Swing?

40. Может ли изменяться стиль оформления компонента Swing? Если да, то какое средство позволяет это сделать?

41. Какой контейнер верхнего уровня чаще всего используется в приложениях?

42. Контейнер верхнего уровня содержит несколько панелей. На какой панели размещаются компоненты?

43. В каком потоке должно происходить все взаимодействие с компонентами графического пользовательского интерфейса?

44. Какая команда действия связывается по умолчанию с компонентом JButton? Как изменить команду действия?

45. Какое событие формируется при нажатии кнопки?

Задания для самостоятельного выполнения

1. Дано целое число. Если оно является положительным, то прибавить к нему 1; в противном случае не изменять его. Вывести полученное число.

2. Дано целое число, лежащее в диапазоне 1–999. Вывести его строку-описание вида «четное двузначное число», «нечетное трехзначное число» и т. д.

3. Дано целое число K. Вывести строку-описание оценки, соответствующей числу K (1 – «плохо», 2 – «неудовлетворительно», 3 – «удовлетворительно», 4 – «хорошо», 5 – «отлично»). Если K не лежит в диапазоне 1–5, то вывести строку «ошибка».

4. В восточном календаре принят 60-летний цикл, состоящий из 12-летних подциклов, обозначаемых названиями цвета: зеленый, красный, желтый, белый и черный. В каждом подцикле годы носят названия животных: крысы, коровы, тигра, зайца, дракона, змеи, лошади, овцы, обезьяны, курицы, собаки и сви-

ньи. По номеру года определить его название, если 1984 г. – начало цикла: «год зеленой крысы».

5. Дано целое число N (> 0). Найти сумму 1 + 1/2 + 1/3 + … + 1/N (вещественное число).

6. Дано целое число N (> 0). Последовательность вещественных чисел A_K определяется следующим образом:

$$A_0 = 1,\ A_K = (A_{K-1} + 1) / K,\ \ K = 1, 2, \ldots .$$

Вывести элементы $A_1, A_2, \ldots, A_N$.

7. Дано вещественное число X (|X| < 1) и целое число N (> 0). Найти значение выражения

$$1 + X / 2 – 1 * X2 / (2 * 4) + 1 * 3 * X3 / (2 * 4 * 6) − \ldots +$$

$$+ (−1)N – 1 * 1 * 3 *\ldots* (2 * N − 3) * XN / (2 * 4 *\ldots* (2 * N)).$$

Полученное число является приближенным значением функции $(1 + X)^{1/2}$.

8. Дано целое число N (> 1). Вывести наименьшее из целых чисел K, для которых сумма 1 + 2 + … + K будет больше или равна N, и саму эту сумму.

9. Дано вещественное число ε (> 0). Последовательность вещественных чисел A_K определяется следующим образом:

$$A_1 = 1,\ A_2 = 2,\ A_K = (A_{K-2} + 2 * A_{K-1}) / 3,\ K = 3, 4, \ldots .$$

Найти первый из номеров K, для которых выполняется условие $|A_K − A_{K–1}| < \varepsilon$, и вывести этот номер, а также числа A_{K-1} и A_K.

10. Описать класс, реализующий шестнадцатеричный счетчик, который может увеличивать или уменьшать свое значение на единицу в заданном диапазоне. Предусмотреть инициализацию счетчика значениями по умолчанию и произвольными значениями. Счетчик имеет два метода: увеличения и уменьшения, – и свойство, позволяющее получить его текущее состояние. При выходе за границы диапазона выбрасываются исключения. Написать программу, демонстрирующую все разработанные элементы класса.

11. Описать класс «процессор», содержащий сведения о марке, тактовой частоте, объеме кэша и стоимости. Преду-

смотреть инициализацию с проверкой допустимости значений полей. В случае недопустимых значений полей выбрасываются исключения. Описать свойства для получения состояния объекта. Описать класс «материнская плата», включающий класс «процессор» и объем установленной оперативной памяти. Предусмотреть инициализацию с проверкой допустимости значений поля объема памяти. В случае недопустимых значений поля выбрасывается исключение. Описать свойства для получения состояния объекта. Написать программу, демонстрирующую все разработанные элементы классов.

12. Описать класс «студенческая группа». Предусмотреть возможность работы с переменным числом студентов, поиска студента по какому-либо признаку (например, по фамилии, имени, дате рождения), добавления и удаления записей, сортировки по разным полям, доступа к записи по номеру. Написать программу, демонстрирующую все разработанные элементы класса.

13. Описать класс «товар», содержащий следующие закрытые поля: название товара; название магазина, в котором продается товар; стоимость товара в рублях. Предусмотреть свойства для получения состояния объекта. Описать класс «склад», содержащий закрытый массив товаров. Обеспечить следующие возможности: вывод информации о товаре по номеру с помощью индекса; вывод на экран информации о товаре, название которого введено с клавиатуры; если таких товаров нет, выдать соответствующее сообщение; сортировку товаров по названию магазина, по наименованию и по цене; перегруженную операцию сложения товаров, выполняющую сложение их цен. Написать программу, демонстрирующую все разработанные элементы классов.

Глава 2

Введение в Android

2.1. Особенности ОС Android

Android – одна из операционных систем нового поколения, созданных для работы с аппаратным обеспечением современных мобильных устройств. На сегодняшний день Windows Mobile, Apple iPhone и Palm Pre предлагают достаточно мощные и более простые в использовании среды разработки мобильных приложений. Однако в отличие от Android это запатентованные операционные системы, в которых в определенных случаях приоритет отдается встроенному ПО, а не приложениям сторонних программистов. Кроме того, эти операционные системы ограничивают возможности взаимодействия приложений с данными телефона, а также ограничивают или контролируют процесс распространения сторонних приложений, созданных для данных платформ.

Android дает новые возможности для мобильных приложений, предлагая открытую среду разработки, построенную на открытом ядре Linux. У всех приложений есть доступ к аппаратным средствам устройства, для чего используются специальные серии API-библиотек. Кроме того, здесь включена полная и контролируемая поддержка взаимодействия приложений. На платформе Android все программы имеют одинаковый статус. Сторонние приложения написаны на том же API, что и встроенное ПО, при этом во всех программах одинаковое время исполнения. Пользователи могут удалять или заменять встроенные ПО на альтернативные сторонние разработки, будь то номеронабиратель или рабочий стол.

Хотя многие возможности Android появились раньше, он стал первой средой, которая совместила в себе следующие особенности:

– по-настоящему открытая, бесплатная платформа разработки ПО, основанная на Linux и открытом коде;

– архитектура, основанная на компонентах, несущая в себе идеи, распространенные в Интернете;

– множество изначально встроенных возможностей;

– автоматическое управление жизненным циклом приложения;

– высококачественные графика и звук;

– переносимость между широким диапазоном существующего и будущего аппаратного обеспечения.

Android предлагает по-новому взглянуть на взаимодействие мобильных приложений с пользователем и техническую базу, делающую это возможным.

2.1.1. Основные элементы приложения

В основе архитектуры Android идея многократного использования компонентов, благодаря чему вы можете публиковать и делиться Активностями, Сервисами (Services) и данными с другими приложениями с возможностью управления доступом с помощью политик безопасности, которые вы назначаете прямо на месте. Аналогичный механизм, с помощью которого вы можете создать собственный менеджер контактов или программу для набора номера, позволяет предоставлять другим программистам свои компоненты – на их основе они смогут создавать новые пользовательские интерфейсы или функциональные расширения, а также строить свои приложения.

Перечислим службы приложений, которые можно назвать базовыми составляющими архитектуры всех приложений для Android, а также основой фреймворка:

– Менеджер Активностей. Контролирует жизненный цикл приложения и поддерживает возможности пользовательской навигации по стеку деятельностей. Activity представляет собой визуальный пользовательский интерфейс для приложения – окно;

– Представления. Используются при создании пользовательских интерфейсов для Активностей;

– Менеджер уведомлений. Обеспечивает работу унифицированных ненавязчивых уведомлений для пользователей;

– Источники данных. Позволяют приложениям открывать доступ к данным;

– Менеджер ресурсов. Обеспечивает отображение некодированных ресурсов, таких как текстовые строки или изображения.

Приложение в Android:

– упаковано в архив с расширением .apk;

– представляет собой набор компонент;

– не имеет стартовой точки типа метода main( ).

Окружение приложения представлено через контекст.

Каждое приложение выполняется в отдельном процессе под отдельным пользователем в многопользовательской среде с минимальными привилегиями.

2.1.2. Виды приложений

Большинство создаваемых программ относятся к одной из перечисленных категорий:

– программы переднего плана. Такое приложение работает, когда оно видимо на экране, в противном случае его выполнение приостанавливается. Пример – игры или картографические мэшапы;

– фоновые. Приложения, с которыми пользователи практически не взаимодействуют, за исключением их настройки. Большую часть времени они находятся в скрытом состоянии. Пример – службы экранирования звонков и SMS-автоответчики;

– смешанные. Предполагают некоторую степень интерактивности, однако большую часть времени работают в фоновом режиме. Как правило, после настройки незаметны. Лишь при необходимости уведомляют пользователя о каких-либо событиях. Пример – мультимедийный проигрыватель;

– виджет. Некоторые приложения представлены исключительно в виде виджетов, размещаемых на домашнем экране.

Сложные приложения трудно втиснуть в какую-то одну категорию, так как они часто содержат элементы каждого из этих типов. При создании приложения вы должны определиться с тем, как именно оно будет использоваться, и только потом приступать к его проектированию.

2.1.3. Строительные блоки Android-приложения

В Android SDK определено несколько объектов, с которыми должен быть хорошо знаком каждый разработчик. Наиболее важные из них – это деятельности (activities), намерения (intents), сервисы (services) и контент-провайдеры (content providers).

Деятельность – это окно или экран пользовательского интерфейса. Приложение может определить одну или несколько деятельностей для поддержки различных стадий работы программы. Каждая деятельность ответственна за сохранение своего состояния, которое может быть восстановлено позднее как часть жизненного цикла приложения. Деятельности расширяют класс Context, поэтому можно использовать их для получения глобальной информации о приложении.

Намерение – это механизм для описания одного действия, такого как «выбрать фотографию», «позвонить домой» или «открыть двери модуля». В Android почти все работает благодаря намерениям, и существует множество возможностей по замене или повторному использованию компонентов.

Сервис – это задача, которая выполняется в фоновом режиме без прямого взаимодействия с пользователем. Сервисы похожи на демоны Unix. Для примера рассмотрим музыкальный проигрыватель. Проигрывание музыки может быть запущено по намерению, но вы хотите, чтобы она играла даже тогда, когда пользователь переместился в другую программу. Для этого код, который выполняет проигрывание музыки, должен находиться внутри сервиса. Позже другая деятельность может подключиться к этому сервису и сообщить ему, что следует переключить или остановить воспроизведение. Android поставляется с множеством встроенных сервисов вместе с удобными API для доступа к ним.

Контент-провайдер – это набор данных, «завернутый» в пользовательский интерфейс API для чтения и записи. Это лучший способ разделять глобальные данные между приложениями. Например, Google предоставляет контент-провайдер для адресной книги. Вся информация здесь – имена, адреса, номера телефонов и т. д. – может быть использована любыми приложениями, которым она нужна.

2.1.4. Настройка среды разработки

Начать разработку приложений для Android довольно просто. Не нужен даже телефон на Android – лишь компьютер, где вы можете установить Android SDK и эмулятор сотового телефона.

Установка инструментов

Набор инструментов разработчика для Android (SDK – Software Development Kit) работает под Windows, Linux и MacOS X. Приложения, которые будут созданы, естественно, можно будет установить на любое Android-устройство. Прежде чем писать текст программы, необходимо установить на свой ПК платформу Java, IDE (Integrated Development Environment – интегрированную среду разработки) и Android SDK.

Java 7

Для начала понадобится копия платформы Java. Все инструменты разработки под Android нуждаются в ней, и программы, которые вы пишете, будут использовать язык Java. Нам нужен JDK 7 или 8. Недостаточно иметь лишь среду времени выполнения (JRE – Java Runtime Environment), понадобится полный комплект для разработчика. Практика показывает, что 32-битная версия предпочтительнее. Пользователи MacOS X должны выбрать последнюю версию MacOS X и JDK с вебсайта Apple.

Последнюю версию JDK можно скачать с сайта компании Sun по адресу http://java.sun.com/javase/downloads/index.jsp.

Установка Android Studio

Существуют разные среды разработки для Android. Например, это могут быть такие среды, как NetBeans, Eclipse, Visual Studio. Рекомендуемой средой разработки является

Android Studio, поэтому ее и будем использовать. Загрузить файл установщика можно с официального сайта: http://developer.android.com/sdk/index.html. Для скачивания пакета установки для OS Windows надо нажать на кнопку «Download Android Studio for Windows».

В процессе установки на компьютер, кроме самой среды Android Studio, также будет установлен набор инструментов Android SDK, в который входит SDK Manager. В нем представлены SDK-компоненты, которые можно скачать, обновить или удалить.

Сначала идет папка Tools – в ней находятся утилиты необходимые для разработки под Android. Далее идет список версий Android. И в самом низу еще есть папка Extras, в которой обычно находятся дополнительные библиотеки.

Как минимум для разработки необходимы два компонента:

1) SDK Platform. Здесь содержатся все программные компоненты системы Android, которые будут использоваться при создании приложений, т. е. окна, кнопки и т. п.;

2) ARM EABI v7a System Image – образ Android системы. Используется для создания эмулятора Android, который нужен будет для тестирования приложений прямо на компьютере, без подключения реальных устройств.

После завершения установки необходимо создать новый проект и запустить его на эмуляторе.

2.2. Манифест и ресурсы приложения

2.2.1. Манифест

Любое приложение, создаваемое в Android, содержит файл манифеста, AndroidManifest.xml, который хранится в корневом каталоге проекта. Манифест позволяет описывать структуру и метаданные приложения, его компоненты и требования.

Манифест включает в себя узлы (теги) для каждого компонента (Активностей, Сервисов, Источников данных и Широковещательных приемников), из которых состоит приложение, и с помощью Фильтров намерений (Intent Filters) и полномочий

определяет, каким образом они взаимодействуют друг с другом и со сторонними программами.

В манифесте предусмотрены атрибуты для указания метаданных (значков и визуальных стилей). Надо отметить, что дополнительные узлы верхнего уровня можно использовать для описания настроек безопасности, модульных тестов (юнит-тестов), аппаратных и системных требований.

Манифест содержит корневой тег с атрибутом package, который ссылается на пакет проекта. Как правило, этот тег также включает в себя атрибут xmlns:android, поддерживаемый системными узлами внутри файла. Необходимо задействовать атрибут versionCode для задания текущей версии приложения в виде целого числа. Это внутреннее значение используется для сравнения версий программы. Примените атрибут versionName для указания публичной версии, которая выводится для пользователей.

Тег <*manifest*> включает в себя узлы, описывающие программные компоненты, настройки безопасности, классы для тестирования и требования, из которых состоит приложение. Укажем теги, доступные внутри узла <manifest>, а также фрагменты кода в формате XML, демонстрирующие, как этими тегами пользоваться.

uses-sdk. Позволяет задать минимальную, максимальную и целевую версии SDK, которые должны быть доступны на устройстве, чтобы приложение смогло правильно функционировать. Основываясь на версии SDK, которая поддерживается установленной платформой, и используя сочетание атрибутов *minSDKVersion, maxSDKVersion* и *targetSDKVersion,* вы можете ограничить круг устройств, способных запускать приложение.

supports-screens. После первой волны устройств с экранами HVGA в 2009 г. список аппаратов под управлением Android пополнился моделями с поддержкой WVGA и QVGA. Поскольку будущие устройства, вероятно, станут оснащаться большими дисплеями, с помощью тега *supports-screen* вы можете указать экранные размеры, которые будут поддерживаться вашим приложением.

Точные цифры будут варьироваться в зависимости от аппаратного обеспечения, но в целом соответствие размеров и разрешений экранов определяется следующим образом:

smallScreens – экраны с разрешением меньшим, чем обычное HVGA, как правило, речь идет о QVGA;

normalScreens – используется для описания экранов стандартных мобильных телефонов, как минимум HVGA;

largeScreens – экраны больших размеров, значительно больше, чем у мобильного телефона;

anyDensity – установить значение true, если приложение способно масштабироваться для отображения на экране с любым разрешением

application. В манифесте может присутствовать только один экземпляр данного тега. В нем используются атрибуты, содержащие метаданные для приложения (включая его название, значок и визуальный стиль). Во время разработки необходимо устанавливать атрибуту *debuggable* значение true, чтобы активизировать режим отладки, хотя для конечных версий его нужно отключить.

Тег <*application*> также играет роль контейнера, который включает в себя узлы для Активностей, Сервисов, Источников данных и Широковещательных приемников, описывающих компоненты приложения. Кроме того, вы можете задать собственную реализацию класса Application.

activity. Тег требуется для каждой Активности, которую отображает приложение. Используйте атрибут *android:name* для указания имени класса Активности.

Добавьте главную Активность, которая будет запускаться первой, а также остальные экраны и диалоговые окна, которые могут показываться (рис. 2.1). Попытка запустить Активности без соответствующего описания в манифесте приведет к выбросу исключения. Каждый тег поддерживает вложенные узлы <*intent-filter*>, указывающие, какие именно Намерения могут запустить Активность.

service. Как и в предыдущем случае, каждый класс Сервиса должен иметь тег service. Теги service поддерживают вложенные узлы <*intent-filter*>, с помощью которых происходит латентное связывание.

provider. С помощью этого тега указываются все Источники данных в приложении. Источники данных используются для управления доступом к базам данных и для обмена информацией в рамках одной или нескольких программ.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.test.myapplication">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="My Application"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Рис. 2.1. Активность в манифесте

receiver. Добавляя в манифест тег *receiver,* можно зарегистрировать Широковещательный приемник, не запуская при этом приложение. Широковещательные приемники отслеживают события на глобальном уровне: пройдя регистрацию, они начнут срабатывать при трансляции системой или приложением соответствующего Намерения. Регистрируя их в манифесте, можете сделать этот процесс полностью анонимным. При трансляции соответствующего Намерения ваше приложение стартует автоматически, запуская зарегистрированный Приемник.

uses-permission. Теги *uses-permission* как часть системы безопасности описывают полномочия, которые, по вашему мнению, нужны приложению для полноценной работы. Добавленные полномочия предоставляются пользователю до установки. Для использования многих стандартных сервисов в Android требуются полномочия (в частности, для действий, связанных с платными услугами и безопасностью, таких как телефонные звонки, прием SMS или использование геолокационных сервисов).

permission. Сторонние приложения также могут указывать полномочия, прежде чем предоставлять доступ к общим программным компонентам. Чтобы ограничить доступ к компонен-

ту приложения, вы должны описать соответствующие полномочия в манифесте. Для этого необходимо использовать тег *permission.* Компоненты текущего приложения могут требовать полномочия с помощью атрибутов *android:permission.* Другие программы должны содержать в своем манифесте теги *uses-permission,* чтобы использовать эти защищенные компоненты. Внутри тега *permission* вы можете указать уровень доступа, который обеспечивается данным полномочием *(normal, dangerous, signature, signatureOrSystem),* метку и внешний ресурс, содержащий описание и объяснение рисков, которыми сопровождается выдача этого полномочия.

Мастер создания проектов в составе ADT (New Project Wizard) автоматически добавляет файл с манифестом для каждого нового проекта.

2.2.2. Создание ресурсов

Ресурсы приложения хранятся в каталоге *res/* внутри дерева проекта. Каждый тип ресурсов представлен в виде подкаталога, содержащего соответствующие данные.

При создании нового проекта дополнение ADT автоматически добавит в него каталог *res* с подкаталогами *values, drawable-ldpi, drawable-mdpi, drawable-hdpi u layout.* В них будут храниться следующие ресурсы: разметка по умолчанию, значок приложения и определения строковых констант.

Каталоги предусмотрены для девяти главных типов ресурсов: простых значений, ресурсов Drawable, менеджеров компоновки, анимации, стилей, меню, настроек поиска, XML и «сырых» (необработанных) данных. При сборке приложения эти ресурсы скомпилируются самым эффективным образом и включатся в программный пакет.

При этом генерируется файл для класса R, содержащий ссылки на все ресурсы проекта, что позволяет ссылаться на ресурсы внутри кода программы. Это дает одно преимущество – проверку синтаксиса во время разработки. Имена файлов для ресурсов должны состоять исключительно из букв в нижнем регистре, чисел, а также символов . (точка) и _ (нижнее подчеркивание).

2.2.3. Создание простых значений

Поддерживаются простые значения – строки, цвета, размеры и массивы (строковые и целочисленные), эти данные хранятся в формате XML внутри каталога res/values.

Каждый тип ресурсов принято хранить в отдельном файле, например файл res/values/strings.xml включает только строковые константы. Далее будут перечислены некоторые виды наиболее часто используемых ресурсов.

Строки

Внешние строковые ресурсы помогают поддерживать совместимость внутри вашего приложения, упрощая процесс создания локализованных версий.

Строковые ресурсы обозначаются тегом *<string>*.

Android поддерживает простую текстовую разметку, поэтому можно использовать теги <b>, <i> и <u>, а также из языка HTML, чтобы выделять части текста полужирным, наклонным или подчеркнутым стилем соответственно:

```
<string name="app_name"><b>Первое
приложение</b></string>
```

Цвета

Для описания цветовых ресурсов применяется тег <color>. Указывайте цвет с помощью символа #, за которым следуют альфа-канал (необязательно) и значения для красного, зеленого и синего цветов в виде одного или двух шестнадцатеричных чисел. Поддерживаются следующие форматы записи: #RGB; #RRGGBB; #ARGB; #AARRGGBB.

Пример:

```
<color name="Gold">#FFD700</color>
```

Размеры

Ссылки на размеры чаще всего встречаются внутри ресурсов со стилями и разметкой. Они пригодятся при создании констант – толщины рамки или высоты шрифта.

Чтобы описать ресурс, используйте тег *<dimen>*, указывая масштаб и один из видов размерности:

– px (экранные пикселы);

– in (физические дюймы);

– pt (физические точки);

– mm (физические миллиметры);

– dp (аппаратно-независимые пикселы, которые вычисляются относительно экрана с плотностью 160 dpi);

– sp (пикселы, не зависящие от масштаба).

В итоге можно описывать размеры не только в абсолютных, но и в относительных значениях, которые зависят от разрешения и плотности экрана, упрощая тем самым масштабирование интерфейса на разных устройствах.

Визуальные стили и темы

Ресурсы со стилями позволяют поддерживать единство внешнего вида приложения с помощью атрибутов, используемых Представлениями. Чаще всего визуальные стили и темы используются для хранения цветовых значений и шрифтов для программы. Вы можете легко менять внешний вид приложения, указывая различные стили в качестве темы в манифесте своего проекта. Чтобы создать стиль, используйте тег <style>, включающий атрибут name, а также один или несколько вложенных узлов item. Каждый тег item, в свою очередь, также должен иметь атрибут name, содержащий тип описываемого значения (например, размер шрифта или цвет). Внутри тега должно храниться само значение.

Ресурсы Drawable

Ресурсы Drawable содержат растровые и растягиваемые (NinePatch) изображения. В эту категорию также входят сложные композитные ресурсы, такие как LevelListDrawables и StateListDrawables, которые могут быть описаны в формате XML.

Все ресурсы Drawable хранятся в виде отдельных файлов в каталоге res/drawable. Идентификаторами для них служат имена файлов в нижнем регистре без расширения.

Разметка

Ресурсы с разметкой (или менеджеры компоновки) позволяют отделять уровень представления от бизнес-логики. С помощью разметки вы можете проектировать пользовательские интерфейсы в формате XML, вместо того чтобы описывать их в коде программы. Чаще всего разметка применяется при описании пользовательского интерфейса для Активности. Создав

разметку в формате XML, можно загрузить ее в Активность с помощью метода setContentView (как правило, внутри обработчика onCreate). Вы также можете получать ссылки на экземпляры разметки, содержащиеся в других ресурсах (например, разметка для каждой строки в элементе ListView).

Использование менеджеров компоновки – рекомендуемый подход при проектировании пользовательских интерфейсов в Android. Отделяя разметку от кода программы, вы получаете возможность оптимизировать пользовательский интерфейс для различных аппаратных конфигураций, учитывая размеры экрана, ориентацию, наличие клавиатуры и сенсорного экрана. Каждый ресурс, описывающий разметку, хранится в отдельном файле в каталоге res/layout. Имя файла выступает как идентификатор ресурса.

2.3. Основы проектирования пользовательского интерфейса

2.3.1. Основные термины

Пользовательский интерфейс (user interface, UI), впечатления от использования (user experience, UX), взаимодействие человека с компьютером (human computer interaction, HCI), юзабилити (usability) – весьма обширные темы.

С появлением Android введено несколько новых терминов для обозначения уже известных программных абстракций.

Представления. Это базовый класс для всех визуальных элементов интерфейса (более известных как элементы управления или виджеты). Все элементы UI, включая разметку, производные от класса View.

Группы представлений. Это потомок класса View, который может содержать внутри себя несколько дочерних Представлений. Наследуйте класс ViewGroup, чтобы создавать сложные Представления, состоящие из взаимосвязанных элементов. Класс ViewGroup также стал основой для менеджеров компоновки, которые помогают размещать элементы управления внутри Активностей.

Активности. Это отображаемые окна (или экраны). Активность – эквивалент Формы для Android. Чтобы вывести на экран пользовательский интерфейс, необходимо назначить для Активности хотя бы одно Представление (как правило, разметку).

2.3.2. Виджеты доступные в Android

Android – это набор стандартных элементов управления, с помощью которых можно создавать простые интерфейсы. Используя эти Представления (а также изменяя и расширяя их при необходимости), можно упростить процесс разработки и обеспечить преемственность между разными приложениями.

Отметим некоторые из наиболее используемых элементов.

TextView. Стандартная метка, предназначенная для вывода текста. Она поддерживает многострочное отображение, форматирование и автоматический перенос слов и символов.

EditText. Редактируемое поле для ввода текста. Поддерживает многострочный ввод, перенос слов на новую строку и текст подсказки.

ListView. Группа представлений, которая формирует вертикальный список элементов, отображая их в виде строк внутри списка. Простейший объект ListView использует TextView для вывода на экран значений toString, принадлежащих элементам массива.

Spinner. Составной элемент, отображающий TextView в сочетании с соответствующим Представлением ListView, которое позволяет выбрать элемент списка для отображения в текстовой строке. Сама строка состоит из объекта TextView, показывающего текущий выбор, и кнопки, при нажатии которой всплывает диалог выбора.

Button. Стандартная кнопка, которую можно нажимать.

CheckBox. Кнопка, поддерживающая два состояния. Представлена в виде отмеченного или неотмеченного флажка.

RadioButton. Переключатель, поддерживающий два состояния и группировку. Группы таких переключателей пользователь видит как набор двоичных вариантов, из которых в определенный момент времени может быть выбран только один.

Это только некоторые из доступных виджетов. Android также поддерживает несколько более продвинутых реализаций Представлений, включая элементы для выбора даты и времени, поля ввода с автодополнением, карты, галереи и вкладки.

2.3.3. Менеджеры компоновки

Менеджер компоновки (более известный как разметка) – это расширение класса ViewGroup, которое используется для позиционирования дочерних элементов внутри пользовательского интерфейса. Экземпляры разметки могут быть вложенными. Комбинируя их, можно создавать сколь угодно сложные интерфейсы.

Android SDK включает некоторые простые виды разметки, которые могут помочь конструировать пользовательские интерфейсы. Только от разработчика зависит выбор правильного сочетания менеджеров компоновки, которые помогут сделать интерфейсы понятными и простыми в использовании.

Перечислим некоторые наиболее универсальные доступные классы разметки.

FrameLayout. Самый простой из менеджеров компоновки, прикрепляет каждое дочернее Представление к верхнему левому углу экрана. Каждый новый элемент накладывается на предыдущий, заслоняя его.

LinearLayout. Помещает дочерние Представления в ряд (горизонтальный или вертикальный). Вертикальная разметка представляет собой колонку с элементами, горизонтальная вытягивает их в строку. LinearLayout позволяет задавать «ширину» каждого дочернего Представления, благодаря чему можно контролировать их размеры в пределах доступного пространства.

RelativeLayout. Наиболее гибкий среди стандартных видов разметки, позволяет задавать позицию каждого дочернего Представления относительного других элементов и границ экрана.

TableLayout. Позволяет размещать Представления с помощью сетки, состоящей из строк и столбцов. При этом столбцы могут либо автоматически растягиваться, либо оставаться постоянной ширины.

Gallery. Отображает элементы в виде однострочного горизонтального списка, который можно прокручивать.

2.4. Публикация приложений

2.4.1. Подготовка приложения к публикации

Первый шаг публикации приложения на Market, конечно, написание программы. Но недостаточно просто написать код. Программа должна быть высококачественной, лишенной ошибок и совместимой с максимальным количеством устройств.

2.4.2. Подписывание

Android требует, чтобы все приложения были упакованы в .apk-файл и подписаны цифровым сертификатом, прежде чем он сможет запустить их. Это не только справедливо для эмулятора и для вашего персонального тестового устройства, но и просто необходимо для программ, которые вы хотите публиковать на Android Market.

2.4.3. Публикация

Android Market – это сервис, поддерживаемый Google, который предназначен для распространения программ. Для того чтобы начать публиковаться, вы должны подписаться как зарегистрированный разработчик на веб-сайте для публикации (также известном как Developer Console). Здесь предусмотрен небольшой регистрационный сбор.

В качестве дополнительного шага, если вы захотите продавать свою программу, подпишите ее с помощью процессора платежей. Веб-сайт для публикаций проинструктирует вас, как это сделать.

Итак, ваш проект готов к выгрузке на сайт. Щелкните по ссылке Upload Application и заполните форму.

Щелкните на кнопке Publish, и ваше приложение появится на Android Market для всех подходящих устройств.

2.4.4. Монетизация приложения

Стратегии заработка на мобильных приложениях, которые можно применить на практике.

1. Платное скачивание

Работает за счет оплаты пользователем всей стоимости вперед. Такой подход отлично работает с играми, развлекательными, навигационными и новостными приложениями, для повышения продуктивности. Однако чем больше пользователям нужно будет уплатить с самого начала, тем меньше их будет устраивать реклама или необходимость дополнительно докупать определенные функции. Вы можете попробовать предложить как бесплатную, так и платную версию приложения, в котором будет представлен дополнительный контент и сведена к минимуму или вообще исключена реклама.

2. Реклама внутри приложения

Один из наиболее популярных методов среди разработчиков мобильных приложений. Внутренняя реклама работает за счет ее отображения на предварительно выделенном пространстве в интерфейсе, что дает прибыль исходя из определенного количества просмотров и(или) переходов. Этот метод действительно эффективно работает в играх, новостных и развлекательных приложениях, мессенджерах. Однако небольшой минус здесь заключается в том, что вам нужно постараться создать такое приложение, которым будут пользоваться часто и долго.

3. Встроенные покупки или подписки

Покупки внутри приложений работают за счет предоставления возможности пользователю покупать дополнительные функции или бонусы. Эта стратегия также отлично работает в играх для слежения за образом жизни и новостных приложениях. Однако такой тип требует действительно преданных последователей вашего продукта, которые захотели бы отдавать деньги за дополнительные функции.

4. Спонсорство

Спонсорство может эффективно сработать для отдельных разработчиков или компаний. Суть спонсорства подразумевает под собой выпуск приложения от имени другого издателя в обмен на популярность и признание. Кроме того, может быть размещен логотип известной компании. Это отлично работает в приложениях, привязанных к определенному географическому положению или событию. Однако недостаток здесь заключается в том, что это одноразовый источник дохода. Обычно такая

стратегия монетизации не позволяет получать длительный или возрастающий доход. Тем не менее для первого приложения это может быть единственным шансом.

5. Реклама во всплывающих окнах

При использовании данного метода, рекламные объявления отображаются в виде диалогового окна внутри приложения, представляя пользователю небольшой призыв к действию и кнопки ОК и Отмена. К примеру, оно может предлагать пользователю «Скачать новую бесплатную 3D-игру», после чего будут представлены соответствующие кнопки.

6. Видеозаставки

Качественные видеоролики показываются прямо внутри приложения. Обычно сервер демонстрирует подобные объявления только тем пользователям, кто применяет wi-fi или 4g-подключение к Интернету, следовательно, сможет просмотреть ролик без остановки от начала и до конца.

Контрольные вопросы

1. Перечислите особенности ОС Android.
2. Какие виды приложений существуют в ОС Android?
3. Что такое manifest? Перечислите основные узлы файла manifest.
4. Перечислите наиболее популярные виды простых ресурсов. Приведите примеры их записи.
5. Перечислите основные виджеты, доступные в Android.
6. Перечислите наиболее популярные менеджеры компоновки. В чем их предназначение?
7. Перечислите основные стратегии заработка с помощью мобильных приложений.

Задания для самостоятельного выполнения

В качестве самостоятельной работы студентам предлагается разработать проект мобильного приложения по одной из нижеперечисленных тем.

1. Будильник.
2. Музыкальный плеер.
3. Психологические тесты.
4. Журнал.
5. Аренда квартир.
6. Заказ такси.
7. Комиксы.
8. Справочник.
9. Игра «Угадай число».
10. Игра Black Jack.
11. Универсальный конвертер единиц измерения.
12. Правила дорожного движения.
13. Судоку.
14. Кредитный калькулятор и калькулятор вкладов.
15. Фитнес-менеджер.
16. Анекдоты.
17. Смс-бокс.
18. Записная книжка.
19. Домашняя бухгалтерия.
20. Новостная лента.

Библиографический список

1. *Бурнет Э.* Привет, Android! Разработка мобильных приложений. СПб.: Питер, 2012.

2. *Буткевич Е. Л.* Пишем программы и игры для сотовых телефонов. СПб.: Питер, 2006.

3. *Васильев А. Н.* Java. Объектно-ориентированное программирование: учеб. пособие. СПб.: Питер, 2011.

4. *Герман О. В., Герман Ю. О.* Программирование на Java и C# для студента. СПб.: БХВ-Петербург, 2005.

5. *Дейтел Х. М., Дейтел П. Дж., Сантри С. И.* Технологии программирования на Java 2. Книга 1. Графика, JavaBeans, интерфейс пользователя: пер. с англ. М.: Бином-Пресс, 2003.

6. *Кислицын Е. В., Першин В. К.* Компьютерное имитационное моделирование: системная динамика и агенты: учеб. пособие. Екатеринбург: Изд-во УрГЭУ, 2016.

7. *Лесневский А. С.* Объектно-ориентированное программирование для начинающих. М.: Бином. Лаборатория знаний, 2005.

8. *Хорстманн К. С., Корнелл Г.* Библиотека профессионала. Java 2: в 2 т. 9-е изд.: пер. с англ. М.: Вильямс, 2015.

9. *Шилдт Г.* Java. Полное руководство. 8-е изд.: пер. с англ. М.: Вильямс, 2012.

10. *Шилдт Г.* SWING: руководство для начинающих: пер. с англ. М.: Вильямс, 2007.

Оглавление

Учебное издание

Кислицын Евгений Витальевич
Шишков Евгений Иванович

РАЗРАБОТКА ПРИЛОЖЕНИЙ НА ЯЗЫКЕ JAVA

Учебное пособие

Редактор и корректор *Л. В. Матвеева*
Компьютерная верстка *И. В. Засухиной*

Поз. 27. Подписано в печать 20.06.2017.
Формат 60 × 84 1/16. Бумага офсетная. Печать плоская.
Уч.-изд. л. 3,3. Усл. печ. л. 5,1. Печ. л. 5,5. Заказ 370. Тираж 38 экз.
Издательство Уральского государственного экономического университета
620144, г. Екатеринбург, ул. 8 Марта/Народной Воли, 62/45

Отпечатано с готового оригинал-макета в подразделении оперативной полиграфии
Уральского государственного экономического университета

УРАЛЬСКИЙ ГОСУДАРСТВЕННЫЙ
ЭКОНОМИЧЕСКИЙ УНИВЕРСИТЕТ