2.1. Классы и объекты

Сайт: IT Академия SAMSUNG
Курс: MDev @ IT Академия Samsung
Книга: 2.1. Классы и объекты
Напечатано:: Егор Беляев
Дата: Суббота, 18 Апрель 2020, 19:13

Оглавление

2.1.1. Введение

В связи с постоянным ростом сложности программ возникла необходимость новой технологии, которая позволила бы упростить разработку. В процессе разработки программистам стало слишком сложно удерживать в памяти детали реализации и те неявные связи, при помощи которых одни компоненты программы влияют на другие.

Технология получила название — объектно-ориентированное программирование (ООП). ***Объектно-ориентированное программирование*** — это основная методология программирования 1990-х годов и начала XXI века. Она представляет собой продукт более 35 лет практики и опыта, которые восходят к использованию языка Simula 67. Идея, которую предложили разработчики этого языка оказалась судьбоносной в истории развития языков программирования. Позже на этих идеях были построены самые известные объектно-ориентированные языки, такие как Java, C++,

В рамках второго модуля мы изучим базовые принципы ООП и освоим инструменты, которые дает нам язык Java.

2.1.2. Классы и объекты

В реальном мире нас окружают объекты. Мы учимся в аудиториях, перемещаемся на транспорте, работаем на компьютерах, звоним по телефонам и т. д. Все это — объекты реального мира. В ООП важно понимать, что описание задачи нужно вести в объектах.

В рамках первого модуля мы уже встречались с ключевым словом class. Одно из многочисленных определений говорит, что класс — это абстрактный тип данных. Центральной идеей ООП является реализация понятия «абстракция данных». Смысл абстракции заключается в том, что сущность можно рассматривать, а также производить определенные действия над ней, как над единым целым, не вдаваясь в детали внутреннего построения. Абстракция означает разработку классов исходя из их интерфейсов и функциональности, не принимая во внимание реализацию деталей.

Например, с помощью типа int мы можем описать одно число целого типа, но при этом мы не можем сказать, что это будет за число: возраст человека, номер дома, страница книги. В нашей повседневной жизни существует огромный разрыв между таким примитивным понятием, как число, и сложными сущностями: «Юнит в компьютерной игре», «Персональная страничка человека в социальной сети», «Счет в банке» и т. д. На практике ООП сводится к написанию некоторого количества классов и последующему их использованию.

С точки зрения ООП каждую такую сущность можно описать как совокупность:

- полей класса — переменных для хранения данных, описывающих класс. Это те свойства, параметры, характеристики, которые описывают состояние сущности;
- методов класса — функций для работы с полями класса. Это те действия, которые можно производить с этой сущностью.

Например, класс «Юнит в компьютерной игре» можно охарактеризовать как совокупность полей: «Имя», «Здоровье», «Броня», «Манна», «Количество патронов» и т. д. и методов: «Атаковать», «Защищаться», «Использовать манну» и т. д.

В итоге, если описать такой класс, мы получаем возможность работать со всем этим набором, как с единым целым, а не отдельными примитивными типами.

Так же, как мы объявляем переменную примитивного типа, мы можем создать переменную класса — объект. Объект — это экземпляр класса. Можно привести аналогию: класс — это форма для выпечки печений, а объект — это сами печенья.

Приведем примеры полей для описанного ранее класса «Юнит в компьютерной игре»:

- Terminator, 100 единиц здоровья, 100 единиц брони, 0 единиц манны, 5 патронов;
- Wizard, 100 единиц здоровья, 100 единиц брони, 200 единиц манны, 0 патронов.

В идеологии ООП разработчик начинает написание программы с проектирования классов. В процессе создания программы — это важнейшая и действительно сложная задача, для решения которой предлагается множество подходов и методик, а также разработаны специальные инструменты для графической иллюстрации структуры классов и их связей между собой (например, диаграммы классов языка графического описания UML). Более подробно процесс разработки будет рассмотрен в следующем модуле.

В основе ООП лежат три принципа:

- ***инкапсуляция*** (от слова encapsulation или incapsulation) — объединение данных и процедур для работы с этими данными в единое целое;
- ***наследование*** (от слова inheritance) — средство получения новых классов из существующих;
- ***полиморфизм*** (от слова polymorphism) — создание общего интерфейса для группы близких по смыслу действий.

В данной теме мы дадим общее описание этих парадигм. Позже каждая из них будет разобрана более детально.

2.1.3. Инкапсуляция

Инкапсуляция (encapsulation) — это механизм, который объединяет данные и код, манипулирующий этими данными, а также защищает и то, и другое от внешнего вмешательства или неправильного использования. Инкапсуляция означает, что методы и поля класса рассматриваются в качестве единого целого. В связи с чем инкапсуляция требует соблюдения следующих принципов:

- все характеристики, которые описывают состояние объекта, должны храниться в его полях;
- все действия, которые можно осуществлять с объектом, должны описываться его методами;
- нельзя извне менять поля объекта.

Обратиться к полям объекта можно только при помощи методов. Из-за чего инкапсуляцию часто связывают с понятием «сокрытие данных». Такой подход позволяет, с одной стороны, обеспечить правильное функционирование внутренней структуры объекта. С другой стороны — при необходимости изменить режим внутренней работы объекта, не меняя способы его внешнего использования. Достаточно изменить внутреннюю структуру.

Например, объект реального мира — наручные часы. Для того чтобы ими пользоваться, нужно уметь их заводить и знать, какое время они показывают. Пользователю не нужно знать, как работает часовой механизм.

Если пользователю позволить менять местами шестеренки, то часы он, скорее всего, сломает.

С другой стороны, если часовщик изменит принцип внутренней работы наручных часов, пользователь этого не почувствует — в его распоряжении все также останется циферблат со стрелками, которые показывают время.

Наручные часы в этом примере (как и инкапсулированный объект) — это «черный ящик» для пользователя. Таким образом, ООП дает возможность пользователю работать с объектами, не задумываясь об их внутреннем устройстве.

2.1.4. Наследование

Наследование — механизм языка, позволяющий описать новый класс на основе уже существующего (родительского, базового) класса. Класс-потомок может добавить собственные методы и свойства, а также пользоваться родительскими методами и свойствами. Позволяет строить иерархии классов.

Некоторые объекты являются частным случаем более общей категории объектов. Например, вы разрабатываете компьютерную игру. В игре должны быть предусмотрены различные персонажи: люди, роботы, волшебники и т. д. С одной стороны, каждый из объектов обладает отдельными характеристиками. С другой стороны, они имеют общие свойства (например, поля «Имя», «Здоровье», «Броня»). Хотелось бы иметь возможность рассматривать их как объект общей категории «Юнит», так как для всех этих объектов применимы общие действия (например, «Атаковать» и «Защищаться»), но при этом уметь хранить для каждого в отдельности его особенные свойства и применять к ним присущие каждому специфические методы.

Используя технологию наследования, говорят, что классы «Человек», «Робот» и «Волшебник» — наследники класса «Юнит». Абстрактный (в данном случае) объект «Юнит» хранит информацию об имени, здоровье и броне, и к нему может быть применена операции «Атаковать» и «Защищаться».

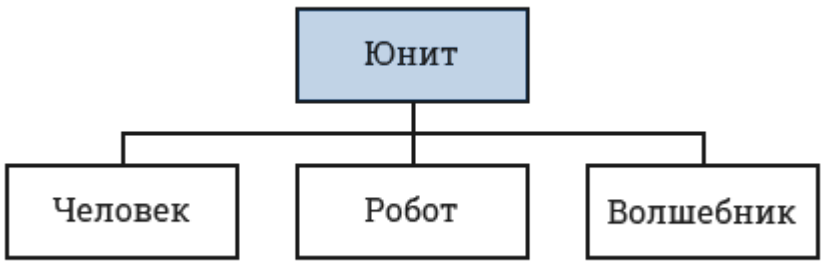


Рис. 2.1.

Еще принято говорить, что класс «Юнит» является родительским по отношению к классам «Человек», «Робот» и «Волшебник» (см. рис. 2.1). Каждый из объектов наследует от класса «Юнит» его поля и методы. И при этом для каждого из них можно добавить свои дополнительные свойства и методы.

2.1.5. Полиморфизм

Полиморфизм — это реализация одинакового по смыслу действия различным способом в зависимости от типа объекта.

Например, в предыдущем разделе мы рассмотрели метод «Атаковать», общий для всех юнитов компьютерной игры.

Несмотря на то что действие общее, каждый наследник реализует его по-своему. Например, человек использует для атаки меч, робот использует лазер, а волшебник — магическое заклинание. В этом заключается идея полиморфного поведения наследников по отношению к родительскому объекту.

При вызове метода «Атаковать» родительского класса для объектов разного типа компилятор сам понимает, какой метод нужно использовать.

2.1.6. Описание класса

Ключевое слово class как начало описания нового типа данных

Научимся создавать собственные классы в языке Java. Прежде всего, нужно написать ключевое слово class. После этого нужно написать имя нового класса. Это имя должно быть создано по общим правилам имен идентификаторов (в том числе переменных) в Java.

Описание класса содержит:

- описание данных, которые хранятся в классе (поля/переменные);
- описание методов, которые обрабатывают эти данные.

Общая форма оформления класса:

```
//ПСЕВДОКОД
class ИмяКласса {
        типПоля1 имяПоля1; // имяПеременной1
        типПоля2 имяПоля2; // имяПеременной2

        типРезультата1 имяМетода1 (параметрыМетода1) {
                //телоМетода1
        }
        типРезультата2 имяМетода2 (параметрыМетода2) {
                //телоМетода2
        }
}
```

Для соблюдения конвенции по оформлению кода принято использовать стиль написания составных слов, при котором несколько слов пишутся слитно без пробелов, при этом каждое слово пишется с заглавной буквы (CamelCase). В Java используют UpperCamelCase для названий классов и lowerCamelCase — для названий полей и методов.

Заметим, что создание нового класса означает создание нового типа данных. Например, давайте создадим класс, который описывает прямоугольник:

```
class Rect {
        double width;
        double height;
}
```

Теперь можно создавать переменные (объекты) этого типа:

```
//ПСЕВДОКОД
ИмяКласса имяОбъекта;
```

В нашем примере:

```
Rect rect; // объявили переменную класса Rect под именем rect
```

Заметим, что в этой строчке кода мы только объявили переменную, то есть сообщили компилятору, что в переменной rect мы собираемся хранить ссылку на объект — экземпляр класса Rect. Самого объекта в памяти еще нет.

Мы даже можем явно указать, что объектная переменная пока ни на что не ссылается, используя значение null:

```
rect = null;
```

Чтобы создать сам объект, необходимо использовать операцию new (мы уже встречались с этой операцией, когда создавали массивы):

```
rect = new Rect();
```

Можно объединить описание и создание объекта. В нашем случае:

```
Rect rect = new Rect();
```

При выполнении оператора new создается новый объект указанного класса, и полученная в результате ссылка на область памяти, в которой был размещен созданный объект, сохраняется в переменную rect. При этом автоматически вызывается специальный метод класса, называемый конструктор, который мы рассмотрим подробно в следующей теме модуля.

В соответствии с официальной документацией Java для переменных различных типов определены следующие значения по умолчанию:

Тип	Значение по умолчанию
byte, short, int, long, float, double	0 (в соответствующем формате представления)
char	'\u0000'
boolean	false
объекты и массивы	null

При создании переменных типа int или float мы не используем операцию new, так как в Java они относятся к примитивным типам данных, а у таких типов значения хранятся непосредственно в переменных. Это позволяет языку обеспечить высокую скорость работы.

В переменных объектного типа, о которых сейчас идет речь, хранятся ссылки на динамическую область памяти, где операцией new было выделено место под хранение объекта со всеми его полями и ссылками на методы.

На самом деле, в первом модуле мы уже создавали переменные объектного типа. Например экземпляры типа Scanner. Scanner — это тоже класс, описание которого находится в пакете java.util. Если удерживать клавишу Ctrl и навести на Scanner в программе, то появится ссылка, пройдя по которой, можно увидеть содержание класса.

Описание полей класса

Для обращения к полям (и методам) созданного объекта нужно использовать операцию «точка».

Например, для обращения к полю width объекта rect нужно написать rect.width.

Установим ширину и высоту прямоугольника rect и посчитаем его площадь:

```
rect.width = 10;
rect.height = 15;
System.out.println(rect.width * rect.height);
```

Заметим, что каждый объект имеет собственные значения полей. Данные одного объекта полностью отделены от данных другого объекта (если необходимо, то это можно поменять, используя ключевое слово static).

Рассмотрим фрагмент:

```
Rect rect1 = new Rect();
rect1.height = 10;
rect1.width = 20;

Rect rect2 = new Rect();
rect2.height = 10;
rect2.width = 20;

System.out.println(rect1 == rect2); // ?
rect2 = rect1;
System.out.println(rect1 == rect2); // ?
```

У каждого из объектов rect1 и rect2 мы можем задать свои собственные значения полей: ширина (width) и высота (height), при этом изменение параметров одного из прямоугольников никак не повлияет на другой.

Если запустить этот фрагмент кода, то будет выведено:

```
false true
```

Почему в первом случае, несмотря на то что значения полей объектов rect1 и rect2 одинаковы, результат операции сравнения == отрицательный, а во втором случае положительный? Дело в том, что здесь операция == сравнивает не содержимое объектов, а ссылки на объекты в памяти. А как мы уже разбирали, оператор new возвращает ссылку на новую область динамической памяти, следовательно, rect1 и rect2 сначала хранят ссылки на разные ячейки памяти.

Далее, после операции rect2 = rect1; переменной rect2 мы присвоили адрес объекта, на который ссылается rect1. Таким образом, обе переменные стали ссылаться на один и тот же объект и во втором случае результат сравнения — true.

Метод класса, его аргументы и возвращаемое значение. Описание метода в протоколе класса

Кроме полей, в классе можно также описать методы (действия), которые можно совершать с объектами класса. Синтаксис описания методов мы разбирали в теме 1.9.1:

```
//ПСЕВДОКОД
типДанныхКоторыеМетодВернетимяМетода(списокПараметровМетода){
        операторы тела метода
}
```

Например, добавим в определение нашего класса Rect метод getArea, который вычисляет площадь прямоугольника:

```
class Rect {
        double width;
        double height;

        double getArea() {
                return width * height;
        }
}
```

Методу getArea в данном случае не нужны параметры, так как он напрямую обращается к полям объекта.

Для вызова метода, как и для обращения к полю класса, нужно использовать операцию «точка». Только для метода необходимо после имени обязательно написать круглые скобки, в которых указываются параметры его вызова.

Даже если методу не требуются параметры, пустые скобки все равно должны быть.

Например, в данном примере:

```
System.out.println(rect1.getArea());
```

Пример метода с параметром:

```
/ Увеличение в указанное число раз размеров прямоугольника
void magnify(double koeff) {
        width *= koeff;
        height *= koeff;
}
```

Обращение к этому методу:

```
rect1.magnify(1.5);
```

Заметим, что для вызова метода, описанного в классе, нужно обязательно указать объект, к которому применяется этот метод (исключение — статические методы, которые будут рассмотрены позже). Например, рассмотренный метод magnify невозможен без указания того, чьи именно ширину и высоту нужно изменить — ведь у каждого объекта они разные. Вызов метода класса полезно рассматривать как посылку сообщения объекту. Указанный (перед точкой) объект рассматривается как адресат, которому посылают сообщение — приказ что-то выполнить. Если не указать конкретный объект — адресата нет и неясно, кому адресовать сообщение (метод).

Еще раз уточним различие между классом и объектом:

- класс — это описание типа данных, некая абстрактная логическая конструкция;
- объект — это нечто реально существующее. Он занимает место в памяти.

Пример 2.1

Спроектируем и реализуем простейший класс, описывающий точку на плоскости. Необходимо предусмотреть методы ввода, вывода точки на экран, перемещения, расчета расстояния между двумя точками.

Для начала опишем поля класса. В качестве полей в данном случае могут выступать координаты точки на плоскости.

```
public class Point {
        double x,y;
}
```

Напишем метод перемещения точки на величины dx и dy

```
void move(double dx, double dy) {
        x += dx;
        y += dy;
}
```

Далее напишем методы ввода и вывода точки на экран. Они могут выглядеть следующим образом

```
void input() {
        Scanner in = new Scanner(System.in);
        System.out.print("Enter x: ");
        double x = in.nextDouble();
        System.out.print("Enter y: ");
        double y = in.nextDouble();
        move(x,y);
}
```

Расстояние между точками вычисляется с помощью теоремы Пифагора

```
double getDistance(Point a) {
        return Math.sqrt( Math.pow(x-a.x, 2) + Math.pow(y-a.y, 2) );
}
```

Наконец, напишем функцию main, в которой продемонстрируем возможности работы с объектами класса Point.

```
public static void main(String [] args){
        //создаем два объекта класса Point
        Point p1 = new Point();
        Point p2 = new Point();

        //вводим и выводим две точки
        p1.input();
        System.out.println("Point 1:");
        p1.printPoint();

        p2.input();
        System.out.println("Point 2:");
        p2.printPoint();

        //выводим расстояние между точками
        System.out.println("Distanse:" + p1.getDistance(p2));

        //перемещаем одну из точек
        p1.move(1, 2);

        //выводим новое положение точки
        System.out.println("Point 1 after move:");
        p1.printPoint();

        //выводим новое расстояние между точками
        System.out.println("New distanse:" + p1.getDistance(p2));
}
```

Результат работы программы может быть следующим:

```
Enter x: 1
Enter y: 2
Point 1:
(1.0; 2.0)
Enter x: 3
Enter y: 4
Point 2:
(3.0; 4.0)
Distanse:2.8284271247461903
Point 1 after move:
(2.0; 4.0)
New distanse:1.0
```

Ниже приведен полный код программы.

```
package ru.samsung.itschool.book;

import java.util.Scanner;

public class Point {
	//поля класса - координаты точки
	double x,y;

	//метод перемещения
	void move(double dx, double dy) {
		x += dx;
		y += dy;
	}

	//метод вывода
	void printPoint() {
		System.out.println("("+x+"; "+y+")");
	}


	//метод ввода
	void input() {
		Scanner in = new Scanner(System.in);
		System.out.print("Enter x: ");
		double x = in.nextDouble();
		System.out.print("Enter y: ");
		double y = in.nextDouble();
		move(x,y);
	}

	//hfccnjzybt vt;le njxrfvb
	double getDistance(Point a) {
		return Math.sqrt( Math.pow(x-a.x, 2) + Math.pow(y-a.y, 2) );
	}

	public static void main(String [] args){
		//создаем два объекта класса Point
		Point p1 = new Point();
		Point p2 = new Point();

		//вводим и выводим две точки
		p1.input();
		System.out.println("Point 1:");
		p1.printPoint();

		p2.input();
		System.out.println("Point 2:");
		p2.printPoint();

		//выводим расстояние между точками
		System.out.println("Distanse:" + p1.getDistance(p2));

		//перемещаем одну из точек
```

```
                p1.move(1, 2);

                //выводим новое положение точки
                System.out.println("Point 1 after move:");
                p1.printPoint();

                //выводим новое расстояние между точками
                System.out.println("New distanse:" + p1.getDistance(p2));
        }
}
```

2.1.7. Обзор классов-оболочек примитивных типов

Мы уже хорошо знаем примитивные (базовые) типы данных языка Java, не являющиеся классами. Эти типы данных не вписываются в общую «объектно-ориентированную картину мира» Java.

Для того чтобы с примитивными типами данных можно было работать так же, как с остальными объектными, для них существуют классы-оболочки (wrapper classes). Они инкапсулируют в себе эти примитивные типы и предоставляют широкий набор методов для работы с ними. Такой прием достаточно широко применяется.

Далее рассмотрим классы-оболочки, соответствующие шести примитивным числовым типам данных. Все эти классы-оболочки являются наследниками класса Number (см. табл. 2.1).

Примитивный тип	**Класс-оболочка**
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Табл. 2.1.

Приведем некоторые примеры использования методов перечисленных классов-оболочек (см. табл. 2.2).

Примеры методов	**Класс-оболочка**	**Описание**
public static Integer **valueOf**(int i) public static Double **valueOf**(double d) …	все	Создание объектов соответствующих классов: Integer i = Integer.valueOf(50); Double db = Double.valueOf(50.5); Long ln = Long.valueOf(50); Short sh = Short.valueOf((**short**)50); Byte bt = Byte.valueOf((**byte**)50);
public int **intValue**() public double **doubleValue**() …	все	Для объекта возвращает соответствующее значение примитивного типа: Integer i = Integer.valueOf(50); int d = i.intValue(); //50

public String **toString**()	все	Результат — строковый десятичный эквивалент вызываемого объекта.
		Пример:
		int x = 125;
		Integer y = x;
		String s1 = y.toString(); //«125»
		String s2 = Double.toString(2.5); //«2.5»

public static int **parseInt**(String s)	Integer	Переводит строковое представление числа s в int **int** value = Integer.parseInt(«10»); System.out.println(value); //10

Примеры методов	Класс-оболочка	Описание
public static int **parseInt**(String s, int radix)	Integer	Переводит строковое представление числа s в системе счисления radix, в целое число: **int** value = Integer.parseInt(«110»,2); System.out.println(value); //выведет 6 (110 в 2-чной системе счисления)
public static double **parseDouble** (String s)	Double	Переводит строковое представление числа в вещественное: double value1 = Double.parseDouble(«256.01»); System.out.println(value1); //256.01

public static String **toBinaryString**(int i)	Integer	Переводит число из 10-чной системы в 2-чную и возвращает в виде строки:
		int x = 12;
		System.out.println(Integer.toBinaryString(x)); //1100

public static String **toOctalString**(int i)	Integer	Аналогичны предыдущему методу, но переводят в 8-чную и 16-чную системы соответственно:
		int x = 12;
		System.out.println(Integer.toOctalString(x)); // 14
		System.out.println(Integer.toHexString(x)); // c

Табл. 2.2.

Автоупаковка/автораспаковка

Автоупаковка (autoboxing) — автоматическая инкапсуляции данных примитивного типа в его класс-оболочку. При этом не нужно создавать объекты с помощью операции new.
Автораспаковка (***auto-unboxing***) — обратный процесс преобразования объектов в соответствующие им примитивные типы.

Подобные средства появились начиная с версии JDK 5. Добавление подобных возможностей значительно упрощает написание кода.

Ниже приведен пример современного способа создания объекта типа Integer:

```
Integer iObj = 5; //автоупаковка
```

Распаковку объекта iObj можно сделать так:

```
int i = iObj; //автораспаковка
```

Обратите внимание, что не нужно явно создавать объект типа Integer для упаковки значения 5 и нет необходимости использовать метод intValue() для распаковки этого значения.

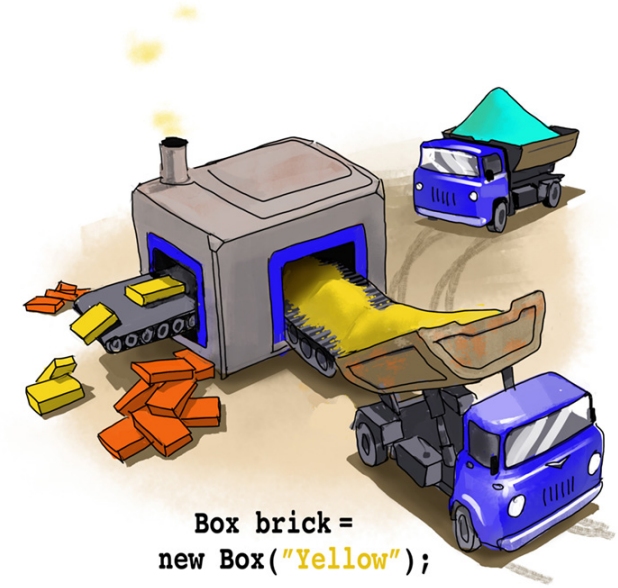
2.2. Конструкторы и их перегрузка. Статические поля и методы

Сайт: IT Академия SAMSUNG
Курс: MDev @ IT Академия Samsung
Книга: 2.2. Конструкторы и их перегрузка. Статические поля и методы
Напечатано:: Егор Беляев
Дата: Суббота, 18 Апрель 2020, 19:15

Оглавление

2.2.1. Конструкторы

Для работы с объектами недостаточно просто объявить переменную определенного класса. Ее также нужно создать. Во время создания объекту отводится память для хранения полей объектов и ссылок на методы. При этом поля объекта автоматически инициализируются значениями по умолчанию, принятыми в Java. Чтобы инициализировать объект по своим правилам используют особые, похожие на методы объявления — конструкторы. Конструктор в Java обладает следующими свойствами:



- обязательно имеет имя, совпадающее с названием класса;
- вызывается при создании объекта;
- в отличие от метода не имеет явным образом определенного типа возвращаемых данных и не наследуется.

Опишем простейший класс натуральной дроби, состоящий из двух полей: числитель и знаменатель.

```
//Класс «Натуральная дробь»
public class Fraction {
        int numerator; // Числитель
        int denominator; // Знаменатель
}
```

Как видно, в данном классе конструктор не написан. Дело в том, что даже если конструктор в классе не описан явно, он есть. Компилятор автоматически создает конструктор, не имеющий параметров. Он проинициализирует поля класса значениями по умолчанию. Такой конструктор называется «конструктором по умолчанию».

Конструктор по умолчанию задаст знаменателю значение 0 (значение по умолчанию для типа int), а нас такой вариант не устраивает, потому что мы можем получить ошибку деления на ноль. Опишем свой конструктор, который присвоит знаменателю единицу.

```
//Класс «Натуральная дробь»
public class Fraction {
        int numerator; // Числитель
        int denominator; // Знаменатель

        public Fraction() {
                denominator = 1;
        }
}
```

Такой конструктор очень примитивный и вполне закономерно замечание, что тот же функционал можно получить, если инициализацию знаменателя совместить с его же объявлением, например, так:

```
public class Fraction {
        int numerator; // Числитель
        int denominator = 1; // Знаменатель
}
```

В этом случае необходимость собственного конструктора отпадет.

Однако, если необходимо предоставить возможность совместить создание объекта и инициализацию его полей собственными значениями, то тогда используют конструктор с параметрами. Создадим такой конструктор и не забудем проверку на недопустимость нулевого знаменателя:

```
public class Fraction {
        int numerator;// Числитель
        int denominator = 1;// Знаменатель
        public Fraction(int num, int denom) {
                if (denom == 0) {
                        System.out.println("Denominator can't be zero. Choose another on
e.");
                        return;
                }
                numerator = num;
                denominator = denom;
        }
}
```

Представленный пример конструктора с параметрами носит скорее иллюстративный характер. Писать такой код в реальных проектах не рекомендуется. Во-первых, для проверки деления на ноль в Java лучше использовать специальный механизм обработки исключений (он будет рассмотрен в следующем модуле). Во-вторых, методология ООП и Java в частности подразумевает многократное использование кода на разных устройствах. Например, создав в Android-приложении объект типа Fraction с нулевым знаменателем, никакого сообщения никуда выведено не будет, так как в Android используют другие возможности языка для вывода информации.

Оператор return так же, как в методе, завершает выполнение конструктора. Как правило, его используют для прекращения инициализации объекта в нештатных ситуациях.

Чтобы не произошло конфликта имен переменных, нам пришлось назвать параметры другими именами. Но в Java есть способ избежать такой конфликт с использованием ключевого слова this. Это ключевое слово ссылается на объект текущего класса, и обращаться к полям или методам класса можно через конструкцию this.название_поля (см. подробнее в 2.2.3).

```
public class Fraction {
        int numerator;// Числитель
        int denominator = 1;// Знаменатель

        public Fraction(int numerator, int denominator){
                if (denominator == 0){
                        System.out.println("Denominator can't be zero. Choose another on
e.");
                        return;
                }
                this.numerator = numerator;
                this.denominator = denominator;
        }
}
```

В приведенном примере видим, что, используя this, можно легко различить, к чему мы обращаемся: к одноименному параметру или полю объекта. Класс может иметь несколько конструкторов, различающихся количеством и типами параметров. Это возможно благодаря приему, который в программировании называется перегрузкой.

Если в классе определен конструктор с параметрами, то конструктор не будет создаваться автоматически по умолчанию и при необходимости его нужно будет написать. Конструкторы имеют следующие особенности:

- не могут напрямую вызываться (необходимо использовать ключевое слово new);
- вызываются один раз при создании объекта в памяти;
- называются так же, как называется класс;
- им нельзя задать возвращаемое значение (всегда возвращают this).

Как вы заметили, инициализировать поля можно в месте, где они объявляются (в нашем примере denominator устанавливается в единицу), либо в конструкторе. В конструкторе, как правило, инициализируют поля, без которых существование объекта не имело бы смысла.

Вернемся к нашему классу. Конструктор дроби защищен от неправильного присвоения значения знаменателю при создании объекта, но ничто не мешает изменить его на ноль после создания объекта. Исправить это нам поможет один из принципов ООП — инкапсуляция.

2.2.2. Перегрузка методов

В Java существует возможность создавать несколько методов с одинаковыми именами, при этом они должны иметь разные параметры. Такой механизм называется перегрузкой.

Метод является перегруженным в том случае, если существует несколько его реализаций с одинаковым именем, но с различной сигнатурой. ***Сигнатура метода*** в Java — это совокупность имени метода с набором параметров. То есть тип возвращаемого значения не входит в сигнатуру, а порядок следования параметров и их типы — входят.

```
void print(double a){
        System.out.println(a);
}
// 1-я перегрузка
void print(String a){
        System.out.println(a);
}
// 2-я перегрузка
void print(int[] a){
        for(int i=0; i<a.length; i++) {
                System.out.print(a[i]+" ");
        }
        System.out.println();
}
```

Пример использования метода:

```
int a = 10;
int [] m = {1, 2, 8, 3}
String s = "Hello";
print(a) //сработает исходный метод
print(a + s); //сработает 1-я перегрузка и будет выведено:5 Hello
print(m); //сработает 2-я перегрузка и будет выведено:1 2 8 3
print(m + a); // произойдет ошибка компиляции
```

В приведенном примере при вызове print(a) компилятор сначала будет искать метод с полным совпадением сигнатуры, то есть метод print(), у которого один параметр типа int. Не найдя такой, компилятор ищет перегруженный метод с совпадающим количеством параметров и совместимыми по преобразованию типами. В нашем примере переменная a не относится к типу double, но поскольку возможно автоматическое приведение типа int в double, то запускается метод с сигнатурой void print(double a). В обратной ситуации, если бы в сигнатуре метода был определен параметр типа int, а в вызове был фактический параметр типа double, подобная конструкция не сработает, так как тип double не может быть автоматически преобразован в int.

Статические методы могут перегружаться нестатическими и наоборот — без ограничений (подробнее в теме 2.2.5). При вызове перегруженных методов следует избегать ситуаций, когда компилятор будет не в состоянии выбрать тот или иной метод.

Пример вызова перегруженных методов:

```
public class Example{
        public static void printNum(Integer i){ // 1
                System.out.println("Integer = " + i);
        }

        public static void printNum(int i){ // 2
                System.out.println("int = " + i);
        }

        public static void printNum(Float f){ // 3
                System.out.println("Float = " + f);
        }

        public static void printNum (Number n){ // 4
                System.out.println("Number = " + n);
        }

        public static void main(String[] args){
                Number[] num = {new Integer(5), 30, 3.14f, 7.5 };
                for (Number n : num)
                        printNum (n);
                printNum (new Integer(8));
                printNum(81);
                printNum(4.14f);
                printNum(8.2);
        }
}
```

В результате компиляции в консоль будет выведено:

```
Number = 5
Number = 30
Number = 3.14
Number = 7.5
Integer = 8
int = 81
Float = 4.14
Number = 8.2
```

При передаче в метод элементов массива каждый раз будет вызван четвертый метод, так как выбор метода происходит на этапе компиляции и зависит от типа массива. Таким образом, при передаче объекта в метод выбор производится в зависимости от типа объекта на этапе компиляции.

При перегрузке методов необходимо придерживаться следующих правил:

- стараться не использовать сложных вариантов перегрузки;
- заменять при возможности перегруженные методы на несколько разных методов;
- избегать произвольного изменения имен параметров в перегрузках. Если параметр в одной перегрузке представляет то же входное значение, что и параметр в другой перегрузке, параметры должны иметь одинаковые имена;
- будьте последовательны при упорядочении параметров в перегружаемых членах. Параметры с одинаковыми именами должны находиться во всех перегрузках на одном и том же месте.

2.2.3. Ключевое слово this

Ранее мы уже сталкивались с ключевым словом this. Рассмотрим возможности его использования подробнее.

Ключевое слово this является ссылкой на экземпляр класса, в котором оно указано. Наподобие объекта, через который получается доступ к полю или методу. Цель этого слова в том, чтобы не придумывать и не создавать лишние переменные.

Одним из возможных вариантов его использования является инструкция return, когда необходимо вернуть явно ссылку на текущий объект:

```
public class Example {
        int i = 0;

        Example inc() {
                i++;
                return this;
        }

        void print() {
                System.out.println("i = " + i);
        }

        public static void main(String[] args) {
                Example x = new Example();
                x.inc().inc().inc().print();
        }
}
```

В результате выполнения программы на экране появится i = 3

В приведенном примере метод inc() возвращает ссылку на текущий объект. При этом возможно многократное увеличение поля i одного и того же объекта.

Слово this часто используется в сеттерах и конструкторах, чтобы отличить поле объекта от параметра метода (если параметр имеет то же имя, что и поле):

```
public void setField(String field) {
        this.field = field;
}
```

Вызов перегруженных конструкторов через this()

При определении перегруженных конструкторов зачастую для того, чтобы не дублировать код, также используют ключевое слово this. В этом случае внутри одного конструктора можно вызвать другой. Компилятор, когда встречает вызов конструктора через this, ищет соответствующий по сигнатуре перегруженный конструктор и выполняет его. Затем управление переходит на операторы, которые следуют за вызовом this(), то есть продолжается выполнение исходного конструктора.

Пример:

```
public class Person {
	private String firstName;
	private String lastName;
	private char gender; // m-male, f- female
	// Конструктор 1
	Person() {
		this("", "",'-'); //Вызывается конструктор 4

	}
	// Конструктор 2
	Person(String lastName) {
		this(lastName, "",'-'); //Вызывается конструктор 4
	}
	// Конструктор 3
	Person(String lastName, String firstName) {
		this.firstName = firstName;
		this.lastName = lastName;
	}
	// Конструктор 4
	Person(String lastName, String firstName, char gender) {
		this(lastName, firstName);//Вызывается конструктор 3
		this.gender = gender;
	}

	public void PrintPerson() {
		System.out.print(this.lastName + " ");
		System.out.print(this.firstName + " ");
		System.out.print(this.gender);
		System.out.println();
	}
}
```

При этом вызов конструктора this() должен быть первым оператором в конструкторе.

В приведенном примере, если мы захотим задать значение полю gender до вызова this, нужно сделать следующим образом:

```
Person(String lastName, String firstName, char gender) { this.gender = gender; this(lastNam
e, firstName);
}
```

В таком случае компилятор выдаст ошибку.

2.2.4. Спецификаторы доступа

В прошлой теме мы говорили об одном из важнейших принципов ООП — инкапсуляции. Инкапсуляция является одним из механизмов сокрытия данных. Говорят, что данные инкапсулируются, то есть скрываются от прямого доступа вне класса. Инкапсуляция позволяет объекту самому управлять доступом к своим данным.

Инкапсуляция (encapsulation) — это сокрытие реализации класса и отделение его внутреннего представления от внешнего (интерфейса).

Поля и методы объекта могут обладать различной степенью открытости. Открытые члены класса являются внешним интерфейсом объекта. Это те поля и методы, которые доступны в других классах. Закрытые поля доступны только внутри класса. На практике закрытыми обычно являются все поля класса, а также вспомогательные методы, которые выступают тонкостями реализации и от которых не зависят другие части программы.

Благодаря такой технике появляется возможность менять внутреннее содержание и логику отдельного класса, при этом не меняя остальных компонентов программы. Обеспечение доступа к полям класса через его методы позволяет контролировать корректные значения полей, так как обращение к свойствам напрямую отслеживать практически невозможно, следовательно, им можно присвоить некорректные значения. Программный код, написанный с использованием принципа инкапсуляции, гораздо легче отладить. Достаточно добавить отладочный вывод в метод, посредством которого выполняется доступ к полю объекта. В противном случае такой вывод пришлось бы добавлять во все участки кода, где используется рассматриваемый объект.

В языке Java существует три модификатора разграничения доступа: public, private, protected. Если модификатор доступа не указан явно, то подразумевается возможный доступ для всех классов, которые находятся в том же пакете (package-private). Также этот модификатор называют модификатором доступа по умолчанию (default).

Ключевое слово private используется для сокрытия методов или переменных класса от других классов, то есть они доступны только внутри класса. Если вспомнить пример с микроволновой печью из предыдущего урока, то ток и напряжение, которые пользователю знать необязательно, — это как раз переменные с доступом private. Если необходимо получить доступ к этим переменным, то определяют внутри класса специальные методы: getter и setter, которые позволяют включить дополнительную логику перед тем как вернуть либо присвоить значение.

Модификатор protected и использование в целом модификаторов к классам будет рассматриваться в теме 2.6. Ключевое слово protected используется для определения видимости членов класса в самом классе и в его наследниках. Модификатор доступа public означает, что метод или поле видны и доступны любому классу. При этом в одном файле может присутствовать только один публичный класс.

В таблице 2.3. более детально показан доступ к членам класса в соответствии с модификаторами доступа.

В том же классе	**Из другого класса**		**Из потомка этого класса (через наследование)**	
	В том же пакете	**В другом пакете**	**В том же пакете**	**В другом пакете**

	В том же классе	Из другого класса		Из потомка этого класса (через наследование)	
		В том же пакете	В другом пакете	В том же пакете	В другом пакете
private	+	–	–	–	–
default	+	+	–	+	–
protected	+	+	–	+	+
public	+	+	+	+	+

Табл. 2.3.

Когда вы создаете файл исходного кода в Java, его называют модулем компиляции (иногда модулем трансляции). Каждый такой модуль имеет расширение .java. Внутри него должен быть только один публичный класс, в противном случае компилятор выдаст ошибку. Имя этого публичного класса и имя файла должны быть одинаковыми (учитывается регистр, но без расширения .java). Если в модуле присутствуют другие классы, то они не видны за пределами текущего пакета. Они являются не публичными и чаще всего выступают вспомогательными для главного публичного класса. Такое ограничение позволяет по названиям файлов легко определить, какие классы в них находятся.

Если выделенные четыре уровня доступа расставить в порядке расширения области доступа, то получится пирамида (http://www.quizful.net/post/features-of-the-application-of-modifiers-in-java) — каждый нижний слой расширяет предыдущий, добавляя свою логику.

Вернемся к нашему классу «Натуральная дробь» и скроем поля от прямого изменения. Для получения и изменения полей создадим свои методы getter и setter (в русской литературе их иногда называют геттеры и сеттеры). Эти методы получили названия от английских слов get и set, то есть получить либо установить значение.

Среда разработки Eclipse обладает рядом полезных возможностей по генерации кода. Например, можно сгенерировать конструкторы, методы getter и setter. Для этого необходимо нажать правой кнопкой по коду, навести на пункт Source и выбрать интересующий Generate. Как альтернатива — можно нажать на пункт Source главного меню и выбрать необходимую генерацию кода.

В Intellij IDEA и Android Studio нужно щелкнуть правой кнопкой по коду, нажать Generate и в появившемся окне выбрать нужный метод, или выбрать в главном меню *Code -> Generate*, также можно воспользоваться горячими клавишами Alt+Insert.

```
public class Fraction{
        private int numerator;// Числитель
        private int denominator = 1;// Знаменатель

        public Fraction(int numerator, int denominator){
                if (denominator == 0) {
                        System.out.println("Denominator can't be zero. Choose another on
e.");
                        return;
                }
                this.numerator = numerator;
                this.denominator = denominator;
        }

        public int getNumerator() {
                return numerator;
        }

        public int getDenominator() {
                return denominator;
        }

        public void setNumerator(int numerator) {
                this.numerator = numerator;
        }

        public void setDenominator(int denominator) {
                if (denominator == 0) {
                        System.out.println("Denominator can't be zero. Choose another on
e.");
                        return;
                }
        this.denominator = denominator;
        }
}
```

Остался маленький штрих: для того чтобы общий знак числа не мог появляться и в числителе, и в знаменателе, нужно хранить знак только в числителе.

```
public class Fraction {
	private int numerator;// Числитель
	private int denominator = 1;// Знаменатель

	public Fraction(int numerator, int denominator) {
		if (denominator == 0) {
			System.out.println("Denominator can't be zero. Choose another on
e.");
			return;
		}
		// знак храним в числителе
		this.numerator = numerator * (denominator < 0 ? -1 : 1);
		// знаменатель всегда положительный
		this.denominator = Math.abs(denominator);
	}

	public int getNumerator() {
		return numerator;
	}

	public int getDenominator() {
		return denominator;
	}

	public void setNumerator(int numerator) {
		this.numerator = numerator;
	}

	public void setDenominator(int denominator) {
		if (denominator == 0) {
			System.out.println("Denominator can't be zero. Choose another on
e.");
			return;
		}
		if (denominator < 0) {
			this.numerator *= -1;
		}
		this.denominator = Math.abs(denominator);
	}
}
```

Пример 2.2

Продолжим разработку класса, описывающего рациональную дробь. Добавим метод для поиска наибольшего общего делителя, например, с помощью алгоритма Евклида.

```
private int gcd(int numerator, int denominator){
        while (numerator != 0 && denominator != 0) {
                if (numerator > denominator) {
                        numerator %= denominator;
                } else {
                        denominator %= numerator;
                }
        }
        return numerator + denominator;
}
```

Далее напишем метод для сокращения дроби:

```
private void normalization(){
        int n = gcd(Math.abs(numerator), Math.abs(denominator));
        numerator /= n;
        denominator /= n;
}
```

Вызовем метод normalization() в последней строчке конструктора класса.

```
public Fraction(int numerator, int denominator){
        if(denominator == 0){
            System.out.println("Denominator can't be zero. Choose another one.");
            return;
        }
        // знак храним в числителе
        this.numerator = numerator * (denominator < 0 ? -1 : 1);
        // знаменатель всегда положительный
        this.denominator = Math.abs(denominator);
        normalization();
}
```

Теперь при вызове конструктора, например, с параметрами Fraction(2, -4), будет создаваться дробь со значением полей numerator = -1 и denominator =2.

Напишем еще несколько вспомогательных методов:

```
public boolean properFraction(){ //проверка правильности дроби
        return (Math.abs(numerator) < denominator ? true : false);
}

public int isIntegerPart(){ // выделение целой части
        return (numerator / denominator);
}

public Fraction fractionalPart(){ // выделение дробной части
        return new Fraction(numerator % denominator, denominator);
}

public double toDecimalFractions(){ // результат деления в виде десятичной дроби
        return (double) numerator / denominator;
}

public String toString(){ // вывод дроби на печать
        return new String(numerator + " / " + denominator);
}
```

Добавим функции сложения дробей

```
public Fraction sumFractionTo(Fraction obj){
        return new Fraction(this.numerator * obj.denominator + obj.numerator * this.denomin
ator, this.denominator * obj.denominator);
}

public static Fraction sumFraction(Fraction obj1, Fraction obj2){
        return new Fraction(obj1.numerator * obj2.denominator + obj2.numerator * obj1.denom
inator, obj1.denominator * obj2.denominator);
}
```

Далее самостоятельно напишите методы для вычитания, умножения и деления дробей. Протестируйте написанные методы в классе Main. Проект с классом Fraction находится здесь.

Полностью написанный нами класс будет выглядеть следующим образом:

```
public class Fraction{
	private int numerator;
	private int denominator = 1;

	public Fraction(int numerator, int denominator){
		if(denominator == 0){
			System.out.println("Denominator can't be zero. Choose another on
e.");
			return;
		}
		this.numerator = numerator * (denominator < 0 ? -1 : 1);
		this.denominator = Math.abs(denominator);
		normalization();
	}


	public int getNumerator(){
		return numerator;
	}

	public int getDenominator(){
		return denominator;
	}

	public void setNumerator(int numerator){
		this.numerator = numerator;
	}

	public void setDenominator(int denominator){
		if (denominator == 0) {
			System.out.println("Denominator can't be zero. Choose another on
e.");
			return;
		}
		if (denominator < 0){
			this.numerator *= -1;
		}
	}

	private int gcd(int numerator, int denominator){
		while (numerator != 0 && denominator != 0) {
			if (numerator > denominator) {
				numerator %= denominator;
			} else {
				denominator %= numerator;
			}
		}
		return numerator + denominator;
	}

	private void normalization(){
		int n = gcd(Math.abs(numerator), Math.abs(denominator));
		numerator /= n;
```

```
                denominator /= n;
        }

        public boolean properFraction(){ //проверка правильности дроби
                return (Math.abs(numerator) < denominator ? true : false);
        }

        public int isIntegerPart(){ // выделение целой части
                return (numerator / denominator);
        }

        public Fraction fractionalPart(){ // выделение дробной части
                return new Fraction(numerator % denominator, denominator);
        }

        public double toDecimalFractions(){ // результат деления в виде десятичной дроби
                return (double) numerator / denominator;
        }

        public String toString(){ // вывод дроби на печать
                return new String(numerator + " / " + denominator);
        }

        public Fraction sumFractionTo(Fraction obj){
                return new Fraction(this.numerator * obj.denominator + obj.numerator * thi
s.denominator, this.denominator * obj.denominator);
        }

        public static Fraction sumFraction(Fraction obj1, Fraction obj2){
                return new Fraction(obj1.numerator * obj2.denominator + obj2.numerator * ob
j1.denominator, obj1.denominator * obj2.denominator);
        }
}
```

2.2.5. Статические компоненты класса

Ключевое слово static

В языке Java модификатор static применяется к внутренним классам, методам, полям и логическим блокам. И тогда мы получаем соответственно статические классы, статические методы, статические поля и статические блоки инициализации.

Статические поля

Иногда бывает нужно, чтобы поле класса имело одинаковое значение для всех объектов данного класса и при изменении значения поля все методы видели новое значение. В таком случае это поле нужно объявить с ключевым словом static (статический) и оно станет общим для всех экземпляров класса.

В языке Java, в отличие, например, от Паскаля или C, отсутствуют глобальные переменные и константы в привычном смысле. Но так как это остается необходимым, применяют статические поля.

Наиболее часто статические поля используют как константы. Например, библиотечный класс Math имеет статическую константу PI:

```
class Math {
        ...
        public static final double PI = 3.14159265358979323846;
        ...
}
```

С одной стороны модификатор доступа public позволяет использовать константу везде, а с другой стороны, добавив слово final, мы гарантируем, что это поле невозможно изменить. Применять общедоступные статические поля без final очень опасно, так как любой объект сможет изменить их значения.

Статические члены класса создаются в памяти сразу при загрузке класса, то есть не нужно создавать экземпляр класса, чтобы их использовать. Обращение происходит через ИмяКласса.поле или ИмяКласса.метод().

В предыдущем примере в любом месте программы (стоит модификатор public) можно получить значение PI, написав Math.PI.

Необходимо с осторожностью подходить к использованию статических полей по нескольким причинам:

- они создаются в момент загрузки класса в память и живут до момента завершения работы приложения, тем самым занимая ресурсы;
- доступность публичных статических полей усложняет отслеживание изменения их содержимого, что особенно опасно в случае многопоточных программ, а также это нарушение принципа ООП — инкапсуляции.

Несколько рекомендаций по использованию статических полей со словом public:

- можно использовать в качестве констант (со словом final);
- по возможности не использовать статические поля в качестве глобальных переменных: нарушение принципа инкапсуляции, а также небезопасно в случае многопоточных

программ.

Блоки статической инициализации

Класс может содержать блоки статической инициализации. Они присваивают значения статическим полям или выполняют иную необходимую логику. Статические блоки обычно применяют, когда простой инициализации в объявлении статического поля недостаточно. Например, создание статического массива зачастую должно осуществляться одновременно с его инициализацией. Ниже приведен пример инициализации статического массива со степенями двойки:

```
public class Example{
        static int[] arr = new int[4]; //статический массив степеней 2

        static { //статический блок
                arr[0] = 1;
                for (int i = 1; i < arr.length; i++)
                        arr[i] = arr[i - 1] * 2;
        }

        public static void main(String[] args) {
                for (int i = 0; i < arr.length; i++)
                       System.out.print(arr[i] + " ");

        }
}
```

В данном примере гарантированно, что массив будет создан до выполнения статического блока. В результате выполнения программы на экране появится:

```
1 2 4 8
```

Статические методы

Как уже было сказано статический метод можно вызвать, используя тип класса, в котором эти методы описаны, то есть не нужно каждый раз создавать новый объект для доступа к таким методам. Класс java.lang.Math — замечательный пример, в котором почти все методы статичны.

Когда использовать статические методы?

1. Когда вы создаете класс-утилиту с часто используемыми простыми методами. Например, вы пишете программу, работающую с деньгами и вам часто необходимо в одном и том же формате выводить стоимость. Или вы пишете программу, работающую с файлами, и часто приходится показывать размер файла, то можно вынести такой метод в отдельный вспомогательный класс.
2. Когда пишите метод, который не зависит от полей своего класса, то есть работает только с входными параметрами. Например, в классе `String` есть метод `valueOf`, который создает строку на основе переданного аргумента.
3. Очень часто статические методы используются при логировании. Например, в Android есть класс Log, имеющий статические методы `d`, `w`, `e` и т. д. (см. тему 2.3.3). Если бы

они были нестатическими, то каждый раз при необходимости вывода сообщения в лог, пришлось бы создавать (или получать) объект класса `Log`, что было бы неудобно.

При использовании статических методов есть ограничения. Вы не можете получить доступ к нестатическим членам класса внутри статического метода. Результатом компиляции приведенного ниже кода будет ошибка:

```
public class Example{
        private int count = 0;

        public static void main(String args[]){
                System.out.println(count); //compile time error
        }
}
```

Приведенный пример — это одна из наиболее распространенных ошибок новичков в Java. Так как метод main статический, а переменная count нет, в данном примере метод println, внутри метода main выдаст «Compile time error». Еще одна распространенная ошибка, когда пытаются сделать локальную переменную статической.

2.2.6. Инициализация различных типов данных

В прошлой теме мы уже затронули вопросы инициализации данных в Java. В этом параграфе рассмотрим их подробнее. В языке Java гарантируется инициализация переменных перед использованием. Если не задать начального значения, то каждое поле примитивного типа получает инициализирующее значение.

```
class InitData{
        boolean b;
        byte B;
        short s;
        int i;
        long l;
        char c;
        float f;
        double d;

        void print(){
                System.out.println("Default values:");
                System.out.println("boolean: " + b);
                System.out.println("byte: " + B);
                System.out.println("short: " + s);
                System.out.println("int: " + i);
                System.out.println("long: " + l);
                System.out.println("char: " + (int)c);
                System.out.println("float: " + f);
                System.out.println("double: " + d);
        }
 }

public class Example {
        public static void main(String [] args){
                InitData initData = new InitData();
                initData.print();
        }
}
```

Программа выведет следующий результат:

```
Default values:
boolean: false
byte: 0
short: 0
int: 0
long: 0
char: 0
float: 0.0
double: 0.0
```

Для демонстрации тип char при выводе в программе приведен к типу int, так как в противном случае выводился бы нулевой символ таблицы кодов ASCII. Как уже говорилось ранее, возможно присвоить переменным начальные значения.

```
class InitData{
        boolean b = true;
        byte B = 10;
        short s = 50;
        int i = 1234;
        long l = 43443434;
        char c = 'A';
        float f = 12.f;
        double d = 123.3;
        //
}
```

Объектные типы по умолчанию инициализируются значением null. Для их инициализации другими значениями необходимо воспользоваться оператором new и вызвать один из определенных в классе конструкторов. Если Test — это класс, то можно вставить переменную и инициализировать ее следующим образом:

```
class InitData{
        Test o = new Test();
        // . . .
}
```

Для инициализации поля можно вызвать метод:

```
class InitData{
        int i = f();
}
```

Как уже говорилось ранее, в языке Java для создания массивов и объектов, используется оператор new. При этом тип элементов массива может быть как примитивным (int, double и т. д.), так и ссылочным.

Элементы массивов примитивных типов по умолчанию инициализируются 0 (false для boolean). Элементы ссылочного массива содержат ссылку null до тех пор, пока не будут явно проинициализированы. Поэтому при попытке обращения к не проинициализированному элементу массива произойдет выброс исключения NullPointerException. Про обработку исключительных ситуаций будем говорить в следующем модуле. В любом случае таких ситуаций необходимо избегать. Инициализацию массива можно проводить отдельно после объявления:

```
Integer[] array = new Integer[3];
array[0] = new Integer(1);
array[1] = new Integer(2);
array[2] = new Integer(3);
```

Также можно явно указать значения массива при объявлении:

```
Integer[] array = new Integer[] {new Integer(1), new Integer(2), new Integer(3)};
```

Или так:

```
Integer[] arr = {new Integer(1), new Integer(2), new Integer(3)};
```

Для числового массива явная инициализация записывается следующим образом:

```
int arr1[]={1, 2, 3};
int arr2[]={}; // это эквивалентно new int[0]
```

Для создания многомерных массивов можно использовать инициализаторы. Тогда необходимо обратить внимание на количество необходимых вложенных фигурных скобок:

```
int i[][] = {{1,2}, null, {3}, {}};
```

В приведенном примере порождается четыре объекта. Это, во-первых, массив массивов длиной 4, а во-вторых, три массива второго уровня с длинами 2, 1, 0 соответственно.

2.3. Приемы тестирования и отладки на примерах со строками

Сайт: IT Академия SAMSUNG
Курс: MDev @ IT Академия Samsung
Книга: 2.3. Приемы тестирования и отладки на примерах со строками
Напечатано:: Егор Беляев
Дата: Суббота, 18 Апрель 2020, 19:16

Оглавление

2.3.1. Введение

Программы без ошибок можно написать двумя способами, но работает только третий.
Алан Перлис, *американский ученый в области информатики*
Отладка кода вдвое сложнее, чем его написание. Так что если вы пишете код настолько умно, насколько можете, то вы по определению недостаточно сообразительны, чтобы его отлаживать.
Брайан Керниган, *создатель языка Си*
Программы всегда содержали ошибки, содержат и будут их содержать. Как правило, последняя найденная ошибка на самом деле является предпоследней.

Отладка — это процесс определения и устранения причин ошибок. Чтобы понять, где возникла ошибка, приходится:

- узнавать текущие значения переменных;
- выяснять, по какому пути выполнялась программа.

Отладка включает поиск дефекта и его исправление, причем поиск дефекта и его понимание обычно составляют 90% работы. В некоторых проектах процесс отладки занимает до 50% общего времени разработки. Существуют две взаимодополняющие технологии отладки.

1. Отладочный вывод и логирование (от англ. log — журнал событий, протокол).
2. Текущее состояние программы логируется с помощью расположенных в критических точках программы операторов вывода.
3. Использование отладчиков.

Отладчики позволяют пошагового выполнять программы оператор за оператором и отслеживать состояние переменных.

2.3.2. Строки

Прежде чем перейти к приемам отладки, рассмотрим подробнее тип данных String, который мы уже не раз использовали. String — это класс, а не примитивный тип данных.

Для создания строки можно использовать следующие конструкции:

```
String имя_строки="Hello World!"; //рекомендуемая форма
String имя_строки= new String("Hello World");
```

Например:

```
String hello1 = "Здравствуйте!";
String hello2 = new String("Здравствуйте!");
```

В результате выполнения приведенных строк кода в памяти создаются объекты класса String, каждый из которых хранит символы строки «Здравствуйте». Ссылки на объекты сохраняются в переменных hello1 и hello2.

В Java строки относятся к ***неизменяемым объектам*** *(англ. immutable)* — объектам, состояние которых не может быть изменено после создания. То есть символы строковой переменной после ее создания поменять нельзя.

При работе с объектами String, если требуется получить другую строку, используя символы имеющейся строки, нужно создать новую строку. При необходимости можно присвоить прежней переменной ссылку на новую строку в памяти, но поменять строку в уже существующем объекте String нельзя.

При необходимости менять символы именно имеющейся строки используйте класс StringBuffer.

В Java неизменяемые объекты получили широкое распространение. Они позволяют обеспечить безопасность при многопоточном выполнении программ, что характерно для приложений под Android.

Основные методы и операции, которые можно применять к строковым переменным, приведены в таблице 2.4.

Операция	Описание
+	Операция конкатенации — две строки сливаются в одну. К символам первого аргумента (того, что стоит до «плюса») приписываются справа символы второго аргумента (того, что стоит после «плюса»). Если при этом один из аргументов — не строка, а другой тип данных, он преобразуется в строковое представление.

Операция	Описание
boolean equals(Object object)	Метод сравнения двух строк — той строки, к которой применяется, и строки, указанной в качестве параметра.Например: String s1 = new String("Ваня учится в IT ШКОЛЕ SAMSUNG"); String s2 = new String("Ваня учится в IT ШКОЛЕ SAMSUNG"); System.out.println(s1.equals(s2)); System.out.println(s1 == s2); На экран будет выведено: true false Метод equals позволяет сравнивать содержимое строк, что не разрешает сделать операция ==.
int length()	Вычисляет длину строки в символах. Для предыдущего примера результат s1.length() будет равен 28.
char charAt(int index)	Выдает символ, находящийся в строке на позиции номер index. Номера символов считаются с нуля (как в массивах). Например, s1.charAt(3) будет равен 'я'.
int compareTo(String anotherString)	Сравнивает две строки по буквам с учетом регистра и языка. Возвращает целое число, показывающее, является ли эта строка больше (результат > 0), равный (результат = 0), или менее (результат < 0) аргумент.

Табл. 2.4.

Мы ранее определили две формы инициализации строк:

```
String имя_строки="содержимое строки";
String имя_строки= new String("содержимое строки");
```

Давайте перепишем предыдущий пример на метод equals(), заменив вторую форму на первую:

```
String s1 = "Ваня учится в IT ШКОЛЕ SAMSUNG";
String s2 = "Ваня учится в IT ШКОЛЕ SAMSUNG";
System.out.println(s1.equals(s2));
System.out.println(s1 == s2);
```

Удивительно, но мы получим другой результат!

```
true
true
```

В чем дело? Почему, казалось бы, эквивалентная замена привела к другому результату?

Ответ кроется в том, что в Java первый способ определяет создание строковых литералов (англ. String literal) с использованием строкового пула, а во втором случае каждый раз создается новый объект String.

Поэтому всякий раз, когда создаются строковые литералы, в строковом пуле проверяется, существует ли такой же строковый литерал. Если он существует, то новый образец для этой строки в строковом пуле не создается и переменная получает ссылку на уже существующий

строковый литерал.

В нашем примере переменным присвоены одинаковые строковые литералы и поэтому переменная s2 получила ссылку на литерал, который был создан для s1. И в результате операция s1 == s2 вернула true.

В связи со всем вышесказанным рекомендуется в программах на Java использовать первую форму создания строк через строковые литералы.

2.3.3. Отладочный вывод и логирование

Логирование основано на включении в программу дополнительных отладочных выводов в местах, где меняются значения переменных (узловые точки). Например, если вы пишите консольную программу, то для ее отладки можно выводить промежуточные значения переменных. Зачастую это помогает найти ошибки в коде и исправить их. В Android для логирования есть специальный класс android.util. Log.

Создадим новый проект. В созданном проекте открываем файл *MainActivity.java*. Чтобы добавить логирование, вставим вызовы метода Log.d() до и после setContentView(). Код теперь должен выглядеть так:

```
protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        Log.d(LOG_TAG, "Creating view..");
        setContentView(R.layout.activity_main);
        Log.d(LOG_TAG, "View created!");
}
```

Android Studio будет подсвечивать Log.d красным цветом, так как класс Log не импортирован. Для быстрого импорта наведите на Log нажмите Alt + Enter.

Android создаст нам константу LOG_TAG со значением null. Заменим null на Main_Activity.

```
public class MainActivity extends Activity {

        public static final String LOG_TAG = "Main_Activity";

        @Override
        protected void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);

                Log.d(LOG_TAG, "Creating view..");
                setContentView(R.layout.activity_main);
                Log.d(LOG_TAG, "View created!");
        }
}
```

Наш код выведет информацию о подгрузке макета для текущей Activity (понятие Activity будет рассмотрено в теме 2.4.5). Для просмотра этой информации нужно запустить проект и открыть вкладку с LogCat.

На вкладке LogCat видны наши сообщения, а также куча других сообщений, которые создаются различными модулями и программами. В данном примере по столбцу Time можно сделать вывод, сколько миллисекунд заняла подгрузка макета.

Чтобы открыть вкладку с LogCat в Android Studio в нижней части окна выберите Android и откройте вкладку.

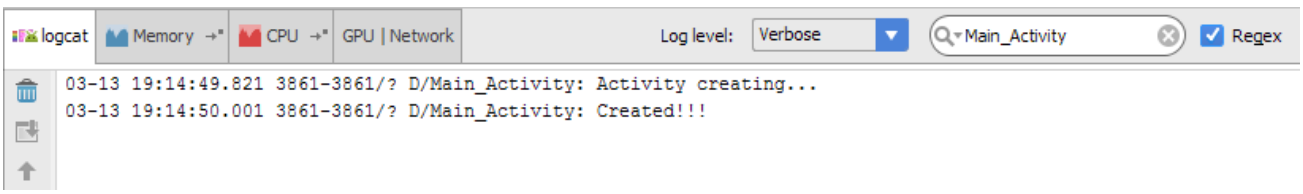
Класс Log разбивает сообщения по категориям в зависимости от важности. Для этого используются специальные методы, которые легко запомнить по первым буквам, указывающим на категорию:

- Log.e() – ошибки (error);
- Log.w() – предупреждения (warning);
- Log.i() – информация (info);
- Log.d() – отладка (debug);
- Log.v() – подробности (verbose);
- Log.wtf() – очень серьезная ошибка! (What a Terrible Failure!, работает, начиная с Android 2.2).

В первом параметре представленных методов класса Log используется строка, называемая тегом. В качестве тега обычно задают имя класса, название библиотеки или название приложения. Обычно принято объявлять глобальную статическую строковую переменную, и уже в любом месте вашей программы вы вызываете нужный метод записи в Log с этим тегом, например:

```
Log.d(LOG_TAG, "Сообщение для записи в журнал");
```

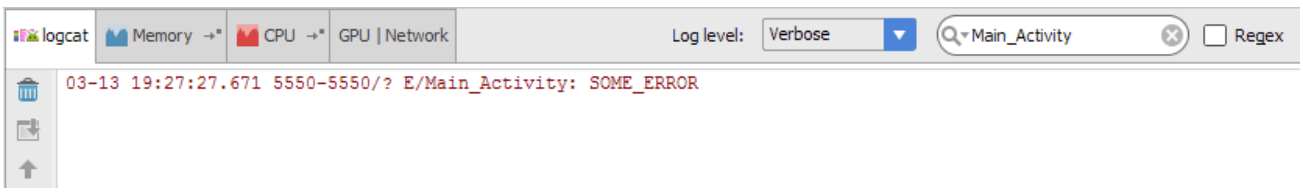
Данный тег мы можем применить для фильтрации сообщения в LogCat. Для этого нужно добавить фильтр по тегу. В Android Studio нажмем в поле поиска и введем наш тег Main_Activity.



Если в коде написано так:

```
Log.e(LOG_TAG, "SOME_ERROR");
```

То запись лога будет подсвечена красным цветом



Также мы можем отображать сообщения по уровням: VERBOSE, DEBUG, INFO, WARN, ERROR и ASSERT. Если выбрать уровень сообщений ERROR, то будут выводиться сообщения, сгенерированные с уровнем ERROR и ASSERT. Если выбрать VERBOSE, то будут выводиться все сообщения.

Как правило, в серьезных приложениях в режиме тестирования постоянно логируется информация об обращении к сторонним сервисам API (application programming interface), обращении к базе данных, при возникновении нестандартных ситуаций и т. д. При выкладывании приложения в маркеты рекомендуется отключать всю отладочную информацию.

Существенный минус данного метода отладки — для получения информации нужно заранее в тексте программы проставить вызовы логирования с соответствующими данными.
Но существует и другой метод отладки, позволяющий отлаживать программу на лету, — использование отладчика.

2.3.4. Использование отладчика

С помощью встроенного отладчика Android Studio мы можем отлаживать программы во время их выполнения.

Предположим мы написали программу, которая генерирует двумерный массив и переводит его в строку. Этот проект выложен в материалах и называется DebugExample.2.3.2.

```
public class MainActivity extends Activity {
        private static final String LOG_TAG = "MainActivity";

        @Override
        public void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                setContentView(R.layout.activity_main);
                Log.v(LOG_TAG, "Matrix: \n" + matrixToString(createMatrix(5, 5)));
        }

        public int[][] createMatrix(int n, int m) {
                int[][] array = new int[n][m];
                for (int i = 0; i < array.length; i++) {
                        for (int j = 0; j < array[i].length; j++) {
                                array[i][j] = (int) (Math.random() * 10);
                        }
                }
                return array;
        }

        public String matrixToString(int[][] array) {
                String result = "";
                for (int i = 0; i < array.length; i++) {
                        for (int j = 0; j < array[i].length; j++) {
                                result += array[i][j] + " ";
                        }
                        result += "\n";
                }
                return result;
        }
}
```

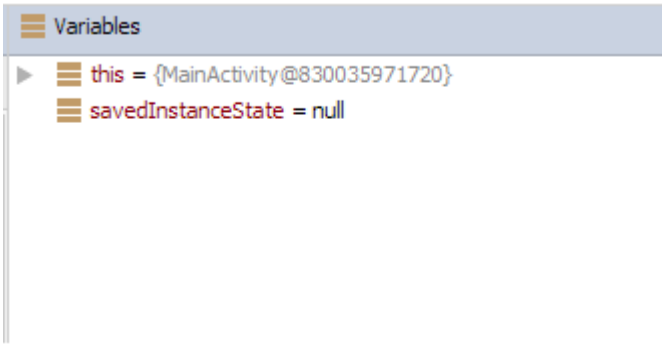
В методе onCreate установим точку останова (breakpoint). Для этого нужно поместить курсор на нужную строчку метода и щелкнуть левой кнопкой мыши слева от номера строки.

На строке кода, где установлен breakpoint, появится кружок. Теперь запустим выполнение в режиме отладки. Для этого нужно нажать на кнопку с жучком, либо выбрать из меню *Run -> Debug ‘app’,* либо нажать на клавиатуре Shift + F9. После запуска приложения Android Studio автоматически откроет вид отладки. Если этого не произошло, то выберите в меню *View -> Tool -> Window -> Debug.*

Перед вами откроется много разных окон, но мы остановимся на самых важных из них: окно с кодом программы и окно *Variables.*

В окне кода одна из строчек подсвечена синим цветом. Так выделена текущая строчка, на которой приостановилось выполнение программы.

В окне *Variables* показаны переменные текущей области видимости:

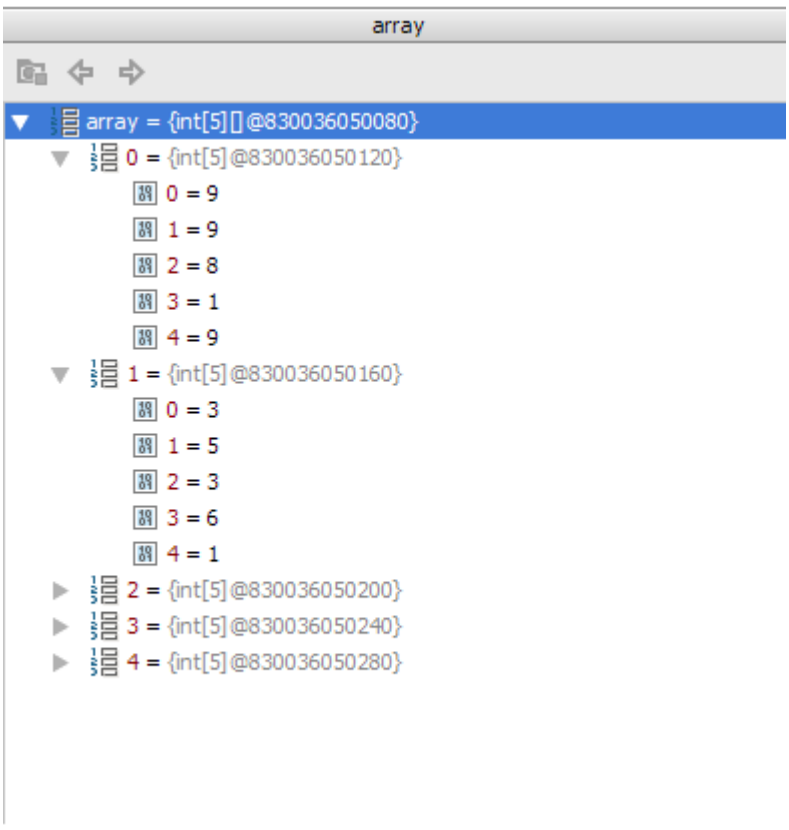


Для работы в режиме отладки удобно пользоваться панелью управления отладкой. На ней расположены кнопки:

Android	Eclipse	Описание
Resume Program (F9)	Resume (F8)	Продолжить выполнение программы
Pause Program	Suspend	Приостановка
Stop (Ctrl + F2)	Terminate (Ctrl + F2)	Прекращение режима отладки
Step Into (F7)	Step Into (F5)	Зайти внутрь метода
Step Over (F8)	Step Over (F6)	Переход к следующей строке
Step Out (Shift + F8)	Step Return (F7)	Выход из текущего метода

Добавим точки останова в начало методов createMatrix и matrixToString и запустим отладку (Resume Program). Выполнение программы остановится на строке с красной точкой в методе createMatrix. В окне *Variables* можно увидеть значения параметров m и n.

Нажимая на кнопку Step Over, выполняем программу шаг за шагом. Заметим, что в окне *Variables* появилась переменная array. По мере выполнения массив наполняется элементами. Если навести курсор на массив, мы увидим значения его элементов.



Если нажать на Step Out, выполнение программы остановится после выхода из текущего метода, то есть к тому методу, который вызвал данный метод. В нашем случае выполнение программы вернулось в метод onCreate. Нажимаем на *Resume Program*, и выполнение переходит в метод matrixToString. Здесь также можно пошагово выполнять команды, чтобы посмотреть, как наполняется строковая переменная result.

Данный инструмент отладки отлично подходит для просмотра значений переменных и просмотра пути выполнения программы. Можно на ходу добавлять, убирать точки останова, задавать условия останова. Но данный метод не может полностью заменить метод вывода отладочной информации. Чаще всего в обычных ситуациях разработчики используют логи для важной информации, а при возникновении необходимости отладки используют инструменты отладки.

2.3.5. Использование утверждений (assertions)

Утверждения (англ. *assertion*, они же ассерты) — это код, используемый, как правило, только во время разработки, с помощью него программа проверяет правильность своего выполнения.

Проверка утверждений во время выполнения программы предполагает выполнение произвольных (и, возможно, длительных) вычислений, которые могут серьезно повлиять на производительность приложения. Одна из самых привлекательных особенностей механизма утверждений — это возможность отключения проверки утверждений в промышленной версии приложения. Таким образом, утверждения — это простой и удобный механизм для поиска ошибок во время разработки, который не оказывает никакого влияния на работу готового продукта. Если утверждение ложное, то программа прекращает выполнение с выводом стека вызова, по которому можно определить, в каком месте программы произошла ошибка.

По умолчанию в Java ассерты (утверждения) отключены. Чтобы их включить, нужно запускать JVM со специальным параметром.

Для включения ассертов в Eclipse нужно зайти в *RunConfigurations -> Arguments -> VM Arguments* и прописать там «-ea». В IntelliJ Idea так же необходимо изменить конфигурацию приложения по умолчанию. В верхнем меню необходимо нажать *Run -> Edit Configurations -> Defaults -> Application -> VM options* и также прописать там «-ea».

Приведем пример утверждения, проверяющего параметры функции, использующей функцию корня числа. Например, нам нужно вычислить выражение:

```
double y = x * Math.sqrt(x);
```

Допустим, вы убеждены, что x является неотрицательным. Однако вы все же хотите проверить свои предположения, чтобы в ходе выполнения программы не возникло ошибки. Одним из возможных вариантов такой проверки является механизм диагностических утверждений (ассертов). Он позволяет вставлять в код программы проверки для тестирования, а затем удалять их из конечной программы. Для этого в Java (начиная с версии Java SE 1.4) существует ключевое слово assert.

```
assert условие; // или assert условие : выражение;
```

Оператор проверяет условие и генерируют исключение AssertionError (про генерацию исключений подробнее будет рассмотрено в теме 3.2), если оно не выполняется. Например, в нашем случае проверка может выглядеть следующим образом:

```
Scanner in = new Scanner(System.in);
int x = in.nextInt();
assert x>=0;
double y = x * Math.sqrt(x);
System.out.println(y);
```

При вводе отрицательного числа на экране появится

```
Exception in thread «main» java.lang. AssertionError at Example.main(Example.java:18)
```

Еще один пример утверждений (проект называется CrossLines.2.3.3, импортировать как Java-проект):

```
import java.util.Scanner;

public class CrossLines {

	public static Scanner in = new Scanner(System.in);
	public static PrintStream out = System.out;

	public static void main(String[] args) {
		double k1, b1, k2, b2;
		k1 = in.nextDouble();
		b1 = in.nextDouble();
		k2 = in.nextDouble();
		b2 = in.nextDouble();
		double[] cross = crossLines(k1, b1, k2, b2);
		if (cross != null) {
			out.println("Пересекаются в (" + cross[0] + ", " + cross[1] + ")");
		}
		else {
			    out.println("Не пересекаются");
		}
	}

	// Функция пересекает две прямые на плоскости,
	// заданных уравнением y = kx + b.
	// Если прямые не пересекаются или совпадают,
	// то возвращается null.
	public static double[] crossLines(double k1, double b1, double k2, double b2) {
		Double dk = k1 - k2;
		// если они параллельны, то возвращаем null
		if (dk == 0.0) {
			return null;
		}
		// формула получена из системы уравнений
		double x = (b2 - b1) / dk;
		double y = k1 * x + b1;
		// Проверка того, что полученная точка находится
		// на обоих прямых (проверка инвариантов).
		// Чтобы проверка работала, необходимо добавить в
		// Run Configurations ⇒ Arguments ⇒ VM Arguments
		// строку -ea.
		assert onLine(k1, b1, x, y);
		assert onLine(k2, b2, x, y);
		return new double[]{x, y};
	}

	// Функция проверяет, что точка лежит на прямой,
	// заданной уравнением y = kx + b.
	private static boolean onLine(double k, double b, double x, double y) {
		return y == k * x + b;
	}
}
```

Если в данном коде поменять формулу вычисления точки пересечения на ложную, то получим:

```
Exception in thread "main" java.lang.AssertionError at CrossLines.crossLines(CrossLines.jav
a:37) at CrossLines.main(CrossLines.java:16)
```

Хочется подчеркнуть, что, так как ассерты могут быть удалены на этапе компиляции, либо во время исполнения программы, они не должны менять поведение программы. Если в результате удаления утверждения поведение программы может измениться, то это явный признак неправильного его использования. Таким образом, внутри ассерта нельзя вызывать функции, изменяющие состояние программы либо внешнего окружения программы. Иначе поведение программы при разработке с включенными ассертами и поведение готового продукта будут различаться.

Более подробно об утверждениях можно почитать в статье «Assert. Что это такое и с чем его едят?»: http://dev.by/blogs/main/assert-chto-eto-takoe-i-s-chem-ego-edyat
О том, каким образом отключаются утверждения в готовых приложениях, можно почитать в блоге компании Oracle: https://blogs.oracle.com/vmrobot/entry/проверкаутверждений_assertionsи_условная

Резюмируя, хочется подчеркнуть, что ассерты нужны программистам для более раннего нахождения грубых ошибок в коде, для того чтобы обратить внимание других программистов на обязательные предусловия/постусловия. Для ожидаемых ошибок (например: отсутствие файла, отсутствие сети, ошибка при установлении соединения, отсутствие определенного датчика на устройстве и т. д.) используется механизм исключений, который будет рассмотрен в следующих разделах курса.

2.3.6. Модульное тестирование

С усложнением программного проекта растет и потенциальное количество ошибок в нем. При этом увеличивается не только количество строк кода проекта, но и, к примеру, появляются новые разработчики, которые заменяют старых или работают совместно (каждый разработчик разрабатывает свою часть проекта). Для добавления нового функционала зачастую приходится переделывать старый код. Ошибки неизбежны.

В связи со всем вышесказанным в промышленной разработке программ используют модульное тестирование. Идея модульного тестирования заключается в написании тестов для всех нетривиальных методов, что позволяет оперативно проверять корректность уже протестированного кода после очередной его модификации. Цель модульного тестирования — исключить из поиска ошибок отдельные части программы путем их автоматической проверки на заранее написанных тестах.

Загрузим проект Testing.2.3.4 и создадим Unit-тест для него. Для этого в Android Studio добавляем через gradle библиотеки junit 4.12. В *Project* выбираем *app -> Gradle Scripts -> build.gradle.* В dependencies добавляем *testCompile* 'junit:junit:4.12'

```
dependencies {
        testCompile 'junit:junit:4.12'
}
```

Далее нажимаем Sync Now. Открываем «Build variants» слева и выбираем в колонке Test Artifacts Unit Tests. Далее создадим новую директорию и собираем там класс MainActivityTest. Напишем тесты для методов MainActivity. Любой метод с аннотацией «@Test» в JUnit считается отдельным тестом. Рекомендуется называть такие методы со слова «test». Кроме этого, все методы тестирования обязательно являются public void. Если внутри теста хотя бы один ассерт вызовет ошибку, то тест будет считаться заваленным. Если все гладко, то тест пройден.

```
public class MainActivityTest {

        int testArray[][] = {{1,2},{1,2}};

        @Test
        public void testCreateMatrix(){
                assertArrayEquals( MainActivity.createMatrix(1, 1), new int[1][1]);
        }

        @Test
        public void testMatrixToString(){
                assertNotEquals( MainActivity.matrixToString(testArray),"1 2 1 2");
        }
}
```

Запустим проект.

Широко известна оценка распределения трудозатрат: на отладку разработчик в среднем тратит в 4 раза больше времени, чем на само написание кода. Поэтому вполне объяснимо, почему в современном мире программирования Unit-тесты стали очень популярны. Даже появилась новая методика разработки программ — разработка через тестирование (англ. *test-driven development, TDD*).

При таком подходе программист еще до написания кода пишет тесты, отражающие требования к модулю.

Несмотря на то что при TDD необходимо написать больше кода, как правило, общее время, затраченное на проект, меньше. Тесты предназначены для защиты от ошибок, поэтому время отладки кода значительно снижается.

При модификации кода, который хорошо «покрыт тестами», риск возникновения новых ошибок значительно снижается. Тесты должны сразу показать, какие ошибки присутствуют в новой функциональности программы. При работе с кодом без тестов обнаружение ошибки может произойти спустя существенное время. Тогда исправить и доработать программу будет значительно сложнее и дороже.

2.3.7. Другие виды тестирования

Разработка программных проектов не ограничивается тестированием отдельных компонентов системы. После модульного тестирования следует интеграционное. ***Интеграционное тестирование*** — это тестирование не отдельных компонентов системы, а результата их взаимодействия между собой в какой-либо среде.

Упор делается именно на тестировании взаимодействия. Так как интеграционное тестирование производится после модульного, то все проблемы, обнаруженные в процессе объединения модулей, скорее всего, связаны с особенностями их взаимодействия.

Вначале описывается план тестирования, подготавливаются тестовые данные, создаются и исполняются тест-кейсы (пошаговые действия для тестирования определенного функционала системы). Найденные ошибки исправляют и снова запускают тестирование. Цикл повторяется до тех пор, пока взаимодействие всех компонентов не будет работать без ошибок.

Для того чтобы автоматизировать интеграционное тестирование, используются системы непрерывной интеграции (англ. *continuous Integration System, CIS*). CIS проводит мониторинг исходных кодов. Как только разработчики выкладывают обновление кода, выполняются различные проверки и модульные тесты. Далее проект компилируется и проходит интеграционное тестирование. Выявленные ошибки включаются в отчет тестирования.

Автоматические интеграционные тесты выполняются сразу после внесения изменений. Это существенно сокращает время поиска и устранения ошибок.

Следующий этап тестирования — системное тестирование, которое разделяется на 2 этапа.

- Альфа-тестирование — имитация реальной работы с системой разработчика, либо реальная работа с системой, ограниченной кругом потенциальных пользователей.
- Бета-тестирование — в некоторых случаях выполняется распространение предварительной версии для некоторой группы лиц с тем, чтобы убедиться, что продукт содержит достаточно мало ошибок. Иногда бета-тестирование выполняется для того, чтобы получить обратную связь о продукте от его будущих пользователей.

Например, среда разработки Android Studio, созданная компанией Google и быстро набирающая популярность, с мая 2013 года находилась в альфа-тестировании, а в июне 2014 года перешла в стадию бета-тестирования.

Кроме тестирования по этапам выделяют также классификацию по объектам тестирования:

1. Юзабилити-тестирование.
2. Тестирование удобства пользования продуктом.
3. Тестирование безопасности.
4. Оценка уязвимости программного обеспечения к различным атакам.
5. Тестирование локализации.
6. Проверка перевода пользовательского интерфейса, документации и сопутствующих файлов программного обеспечения на различных языках.
7. Тестирование производительности.
8. Проверка скорости работы системы под определенной нагрузкой. В тестировании производительности выделяют следующие направления:
    - Нагрузочное тестирование. С помощью нагрузочного тестирования обычно оценивают поведение программы под заданной ожидаемой нагрузкой. Такой

нагрузкой зачастую выступает, например, ожидаемое число одновременно работающих пользователей, которые совершают определенное количество действий за заданный интервал времени.
    - Стресс-тестирование. С его помощью обычно исследуют пределы пропускной способности программы. Такое тестирование проводят для определения надежности системы во время значительных нагрузок. Стресс-тестирование позволяет ответить на вопрос о производительности системы в случае значительного превышения ожидаемого максимума нагрузки. Стресс-тестирование позволяет определить «узкие места» системы, которые вероятнее всего приведут к сбоям системы в пиковой ситуации. Проверка правильности работы программы осуществляется на большом количестве случайно сгенерированных данных. Мотивация данного тестирования — предпочтительнее быть готовым к обработке экстремальных условий системы, чем ожидать отказа системы, тем более когда стоимость отказа системы в экстремальных условиях может быть очень велика.

Для лучшей иллюстрации рассмотрим виды тестирования по аналогии с производством техники, например, телефонов.

Завод закупает множество комплектующих — от винтиков до печатных плат. Очевидно, что качество готовой продукции напрямую зависит от любого из компонентов телефона. Поэтому контроль входного качества компонентов очень важен.

Все начинается с простейших тестов — веса и размера компонентов. Из партии деталей выбирают часть для тестов. Если это новый поставщик или деталь ранее не тестировалась, то проверка проходит самым тщательным образом — устойчивость к агрессивной среде, влажность, прочностные характеристики, рентген-снимки на наличие скрытых дефектов и так далее. В результате тестирования фабрика решает, будет ли она работать с данным поставщиком. Это все модульное тестирование.

После сборки телефона вставляется сим-карта и начинается проверка его работоспособности в целом: по заданным сценариям проверяются функции телефона. Это — интеграционное тестирование.

Аппараты могут проходить и более тщательную проверку:

- время работы телефона в различных режимах;
- деградация аккумулятора с количеством циклов разряда/заряда;
- работа процессора при максимальной нагрузке.

В тестовой лаборатории могут присутствовать помещения для проверки в условиях различных температур и влажности — имитация климатических зон. Кроме этого, есть специальные камеры «старения» для получения реального эффекта старения за максимально короткий срок.

Механическая прочность экрана испытывается металлическими шариками, которые падают на него с определенной высоты. На других автоматах телефон поднимают на определенную высоту и роняют на металлическую поверхность.

Все описанное специальное оборудование применяется для нагрузочного и стресс-тестирования. Зачастую эти понятия считают синонимами, но это не совсем верно. Нагрузочное тестирование — это когда для проверки воссоздают предполагаемые нормальные условия эксплуатации. Когда же производится проверка в условиях сверхвысоких (выходящих за пределы обычного использования) нагрузок, то это уже стресс-тестирование.

Задание

В приложении LogCat.2.3.1 добавьте логирование в другие методы класса, в частности onCreateOptionsMenu и onOptionsItemSelected, и посмотрите, когда происходят вызовы этих методов.

2.4. Знакомство с Android-разработкой

Сайт: IT Академия SAMSUNG
Курс: MDev @ IT Академия Samsung
Книга: 2.4. Знакомство с Android-разработкой
Напечатано:: Егор Беляев
Дата: Суббота, 18 Апрель 2020, 19:16

Оглавление

2.4.1. Платформа Android

Android — свободно распространяемая, активно развивающаяся операционная система (ОС) для мобильных устройств. Эта ОС основана на ядре Linux 2.6, включая прикладное программное обеспечение. Первая устойчивая версия Android 1.0 была выпущена 23 сентября 2008 года. На сегодняшний день Android — самая распространенная система для мобильных устройств. Изначально ОС разрабатывалась компанией Android Inc., которую затем купила Google. Google в альянсе Open Handset Alliance (OHA) занимается поддержкой и дальнейшим развитием платформы Android. Инструментарий программной разработки Android SDK находится в свободном доступе и включает в себя интерфейсы прикладного программирования (API) на языке Java. В IT ШКОЛЕ SAMSUNG используют IDE Eclipse с плагином ADT (Android Developer Tools) или Android Studio, в состав этих сред разработки включены все инструменты, необходимые для программирования под Android. С декабря 2014 года компания Google официально объявила о прекращении поддержки плагина ADT и переходе на Android Studio. Скачать Android Studio можно по адресу http://developer.android.com/sdk/index.html. IDE Eclipse с плагином ADT размещен в учебном курсе. Архитектура системы включает в себя четыре уровня (см. рис. 2.2):

- приложения;
- система приложений;
- библиотеки;
- ядро Linux.

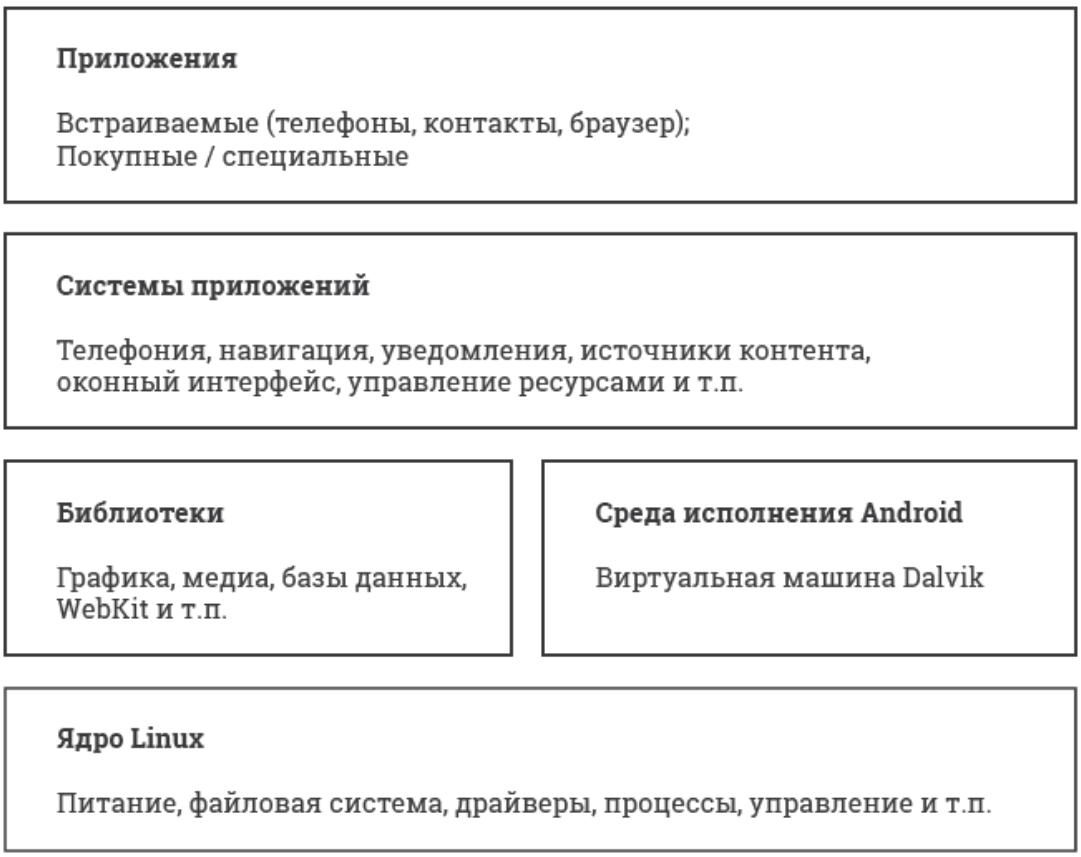


Рис. 2.2. Структура ОС Android

Система приложений. Включает различные службы, курирующие работу составляющих системы:

- ***Activity Manager*** — диспетчер активности, который отвечает за функционирование приложения и его жизненный цикл;

- ***Resource Manager*** — диспетчер ресурсов, необходим для доступа к используемым внутренним ресурсам (строковым, графическим и т. п.);
- ***Package Manager*** — диспетчер пакетов, отвечающий за установку и функционирование пакетов прикладных программ;
- ***Window Manager*** — диспетчер окон, распределяющий активность окон приложений и порядок их отображения;
- ***Telephony Manager*** — менеджер телефонии, следит за типом доступа и параметрами сети;
- ***Location Manager*** — менеджер местоположения — навигационные службы, передающие приложениям информацию о местоположении устройства;
- ***Notification Manager*** — диспетчер уведомлений, позволяет приложению публиковать сообщения в строке состояния;
- ***Content Providers*** — менеджер внешних ресурсов, открывающий доступ к другим приложениям;
- ***View System*** — система представлений, используемая для создания внешнего оформления приложения. Имеет расширяемую функциональность.

Библиотеки. Кроме стандартных SLD (2D-графика), OpenGL 3D-графика), Media Framework (мультимедиа), WebKit (встроенный браузер), FreeType (поддержка шрифтов), SQLite (работа с базой данных), SSL (зашифрованные соединения), разработчики Android создали собственную версию стандартной библиотеки C/ C++ — библиотеку Bionic (не поддерживаются исключения C++ и несовместима с GNU libs и POSIX).

Среда исполнения. Dalvik Virtual Machine — виртуальная машина Java, выполняющая Java программы также, как их выполняет JVM на PC. Начиная с версии Android 4.4 KitKat, в системе появилась альтернатива DVM — ART (Android Runtime) — среда исполнителя приложений без предварительной программной настройки устройства.

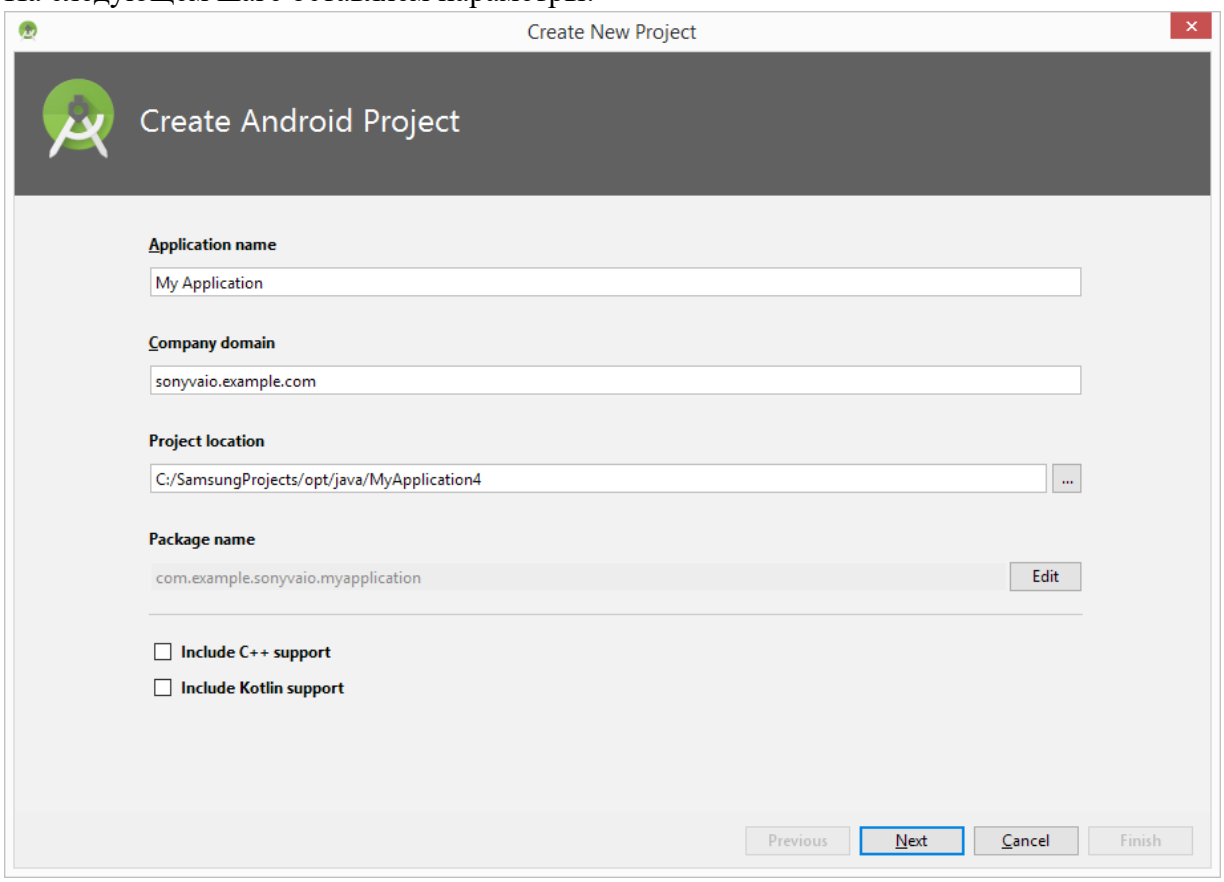
Ядро Linux. На этом уровне контролируется аппаратное обеспечение устройства, в том числе работают драйверы межпроцессорного взаимодействия (IPC) и управления питанием. Хотя система и построена на ядре Linux, однако, она имеет некоторые специфические расширения ядра, свойственные Android, а значит, не является Linux-системой.

2.4.2. Создаем Android-проект

Важно, чтобы разные программисты имели возможность свободно использовать любые имена классов в своих проектах, в том числе и одинаковые. Имя домена сайта разработчика уникальны по определению, поэтому размещение файлов в пакете с таким именем гарантирует отсутствие конфликтов. При этом для удобства имена пишутся в обратном порядке. При этом естественным образом соблюдается иерархичность. Например, создавая проект Hello Application в пакете hello.itschool.samsung.ru ваши классы автоматически расположится в папке *ru/samsung/itschool/hello/helloapplication*.

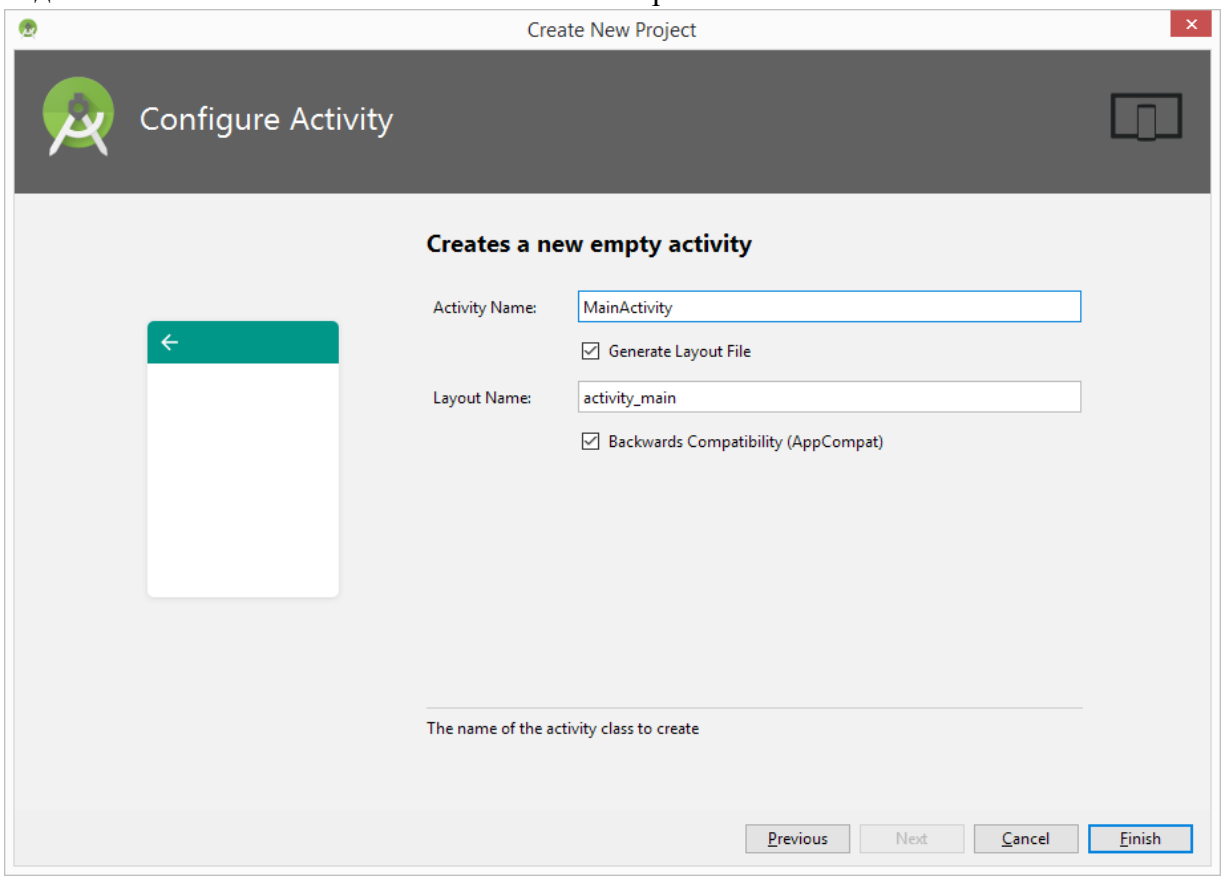
Как создать проект в Android Studio

1. Открываем мастер создания проекта: File-> New -> Project.
2. Задаем имя, домен и папку для сохранения проекта.
3. На следующем шаге оставляем параметры:



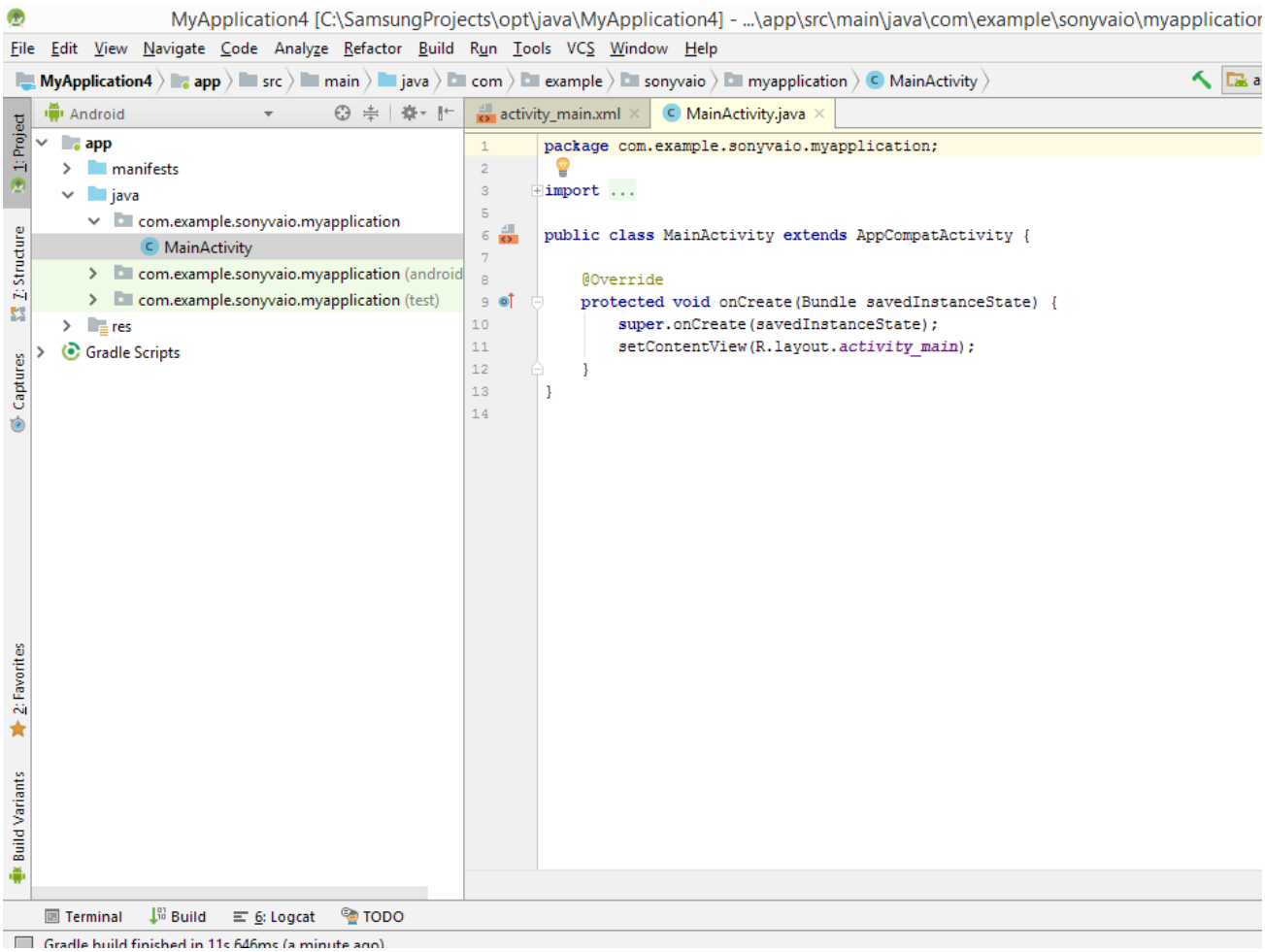
4. На следующем шаге выбираем проект Blank Activity.

5. Задаем наименования основных компонентов приложения:

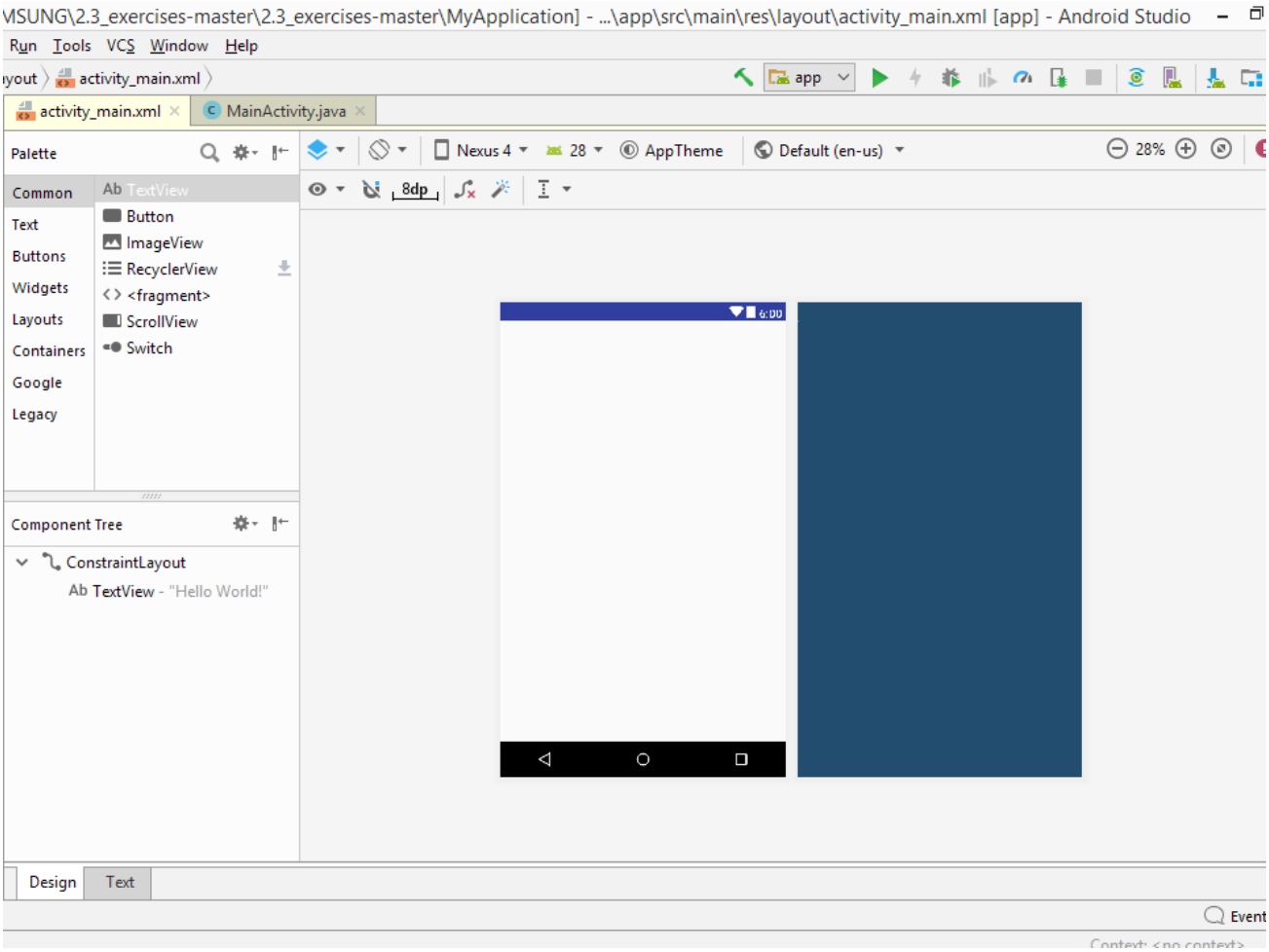


6. Завершаем создание пустого кнопкой Finish.

В результате открывается окно проекта:

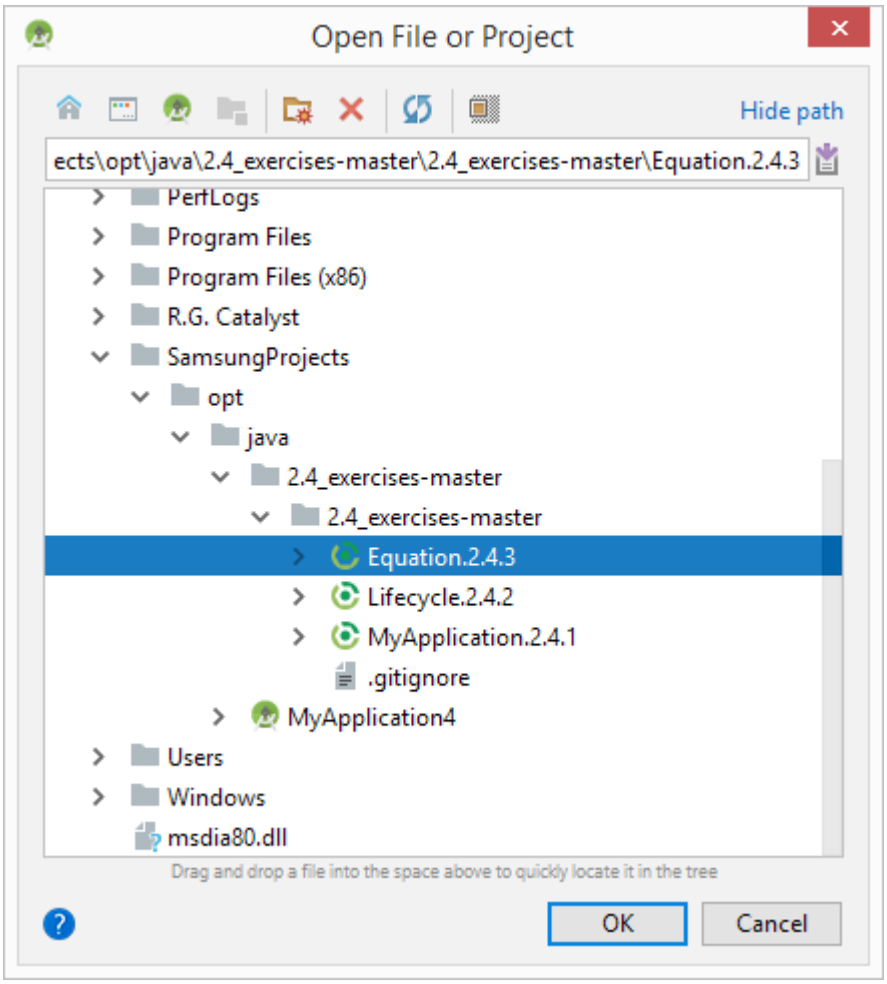


Если будут проблемы с отображением окна, то перещелкните на другой API для отображения:



Как загрузить существующий проект под Android.

Для загрузки существующего проекта скопируйте проект из указанной преподавателем папки на сервере или системы обучения и импортируйте его. В Android Studio для импорта необходимо в меню выбрать File->New->Import Project.



Android-проекты современных сред разработки (Idea, Android Studio, Eclipse ADT) обычно совместимы и могут быть загружены в Android Studio. Однако при импорте проекта, даже в рамках одной IDE, могут возникнуть следующие типовые проблемы:

- кодировки — русский текст в проекте, созданном на Linux, MacOS на Windows системе может стать нечитаемым (и наоборот);
- разница в версиях SDK — если загружаемый проект ориентирован на иную версию API, чем установленные версии API в текущей SDK;
- дополнительные пакеты — если в загружаемом проекте используются дополнительные библиотеки, которые по каким-то причинам не загружаются;
- ключи — если в проекте используются ключи от внешних API (например, Google API), сгенерированные с использованием keystore SDK машины источника.

2.4.3. Запуск приложения

Для запуска приложения можно использовать физическое Android-устройство либо его эмулятор. При этом надо отметить, что эмулятор очень требователен к производительности компьютера, на котором работает IDE, и не всегда может заменить реальное устройство. Поэтому мы рекомендуем использовать первый способ.

Важное замечание!
При работе с Android-проектами IDE заметно притормаживает. Нужно к этому очень терпеливо относиться. Android Studio, например, особенно долго грузится в первый раз, потом это будет происходить быстрее. А Eclipse сразу после загрузки может показать много ошибок в проекте, но через полминуты все станет нормально. Если при этом немедленно пытаться исправить ситуацию, щелкая мышкой по управляющим элементам среды и фактически запуская новые команды, скорее всего, Eclipse зависнет совсем и его придется перезагружать. Будьте терпеливы!

Как подключить Android-устройство

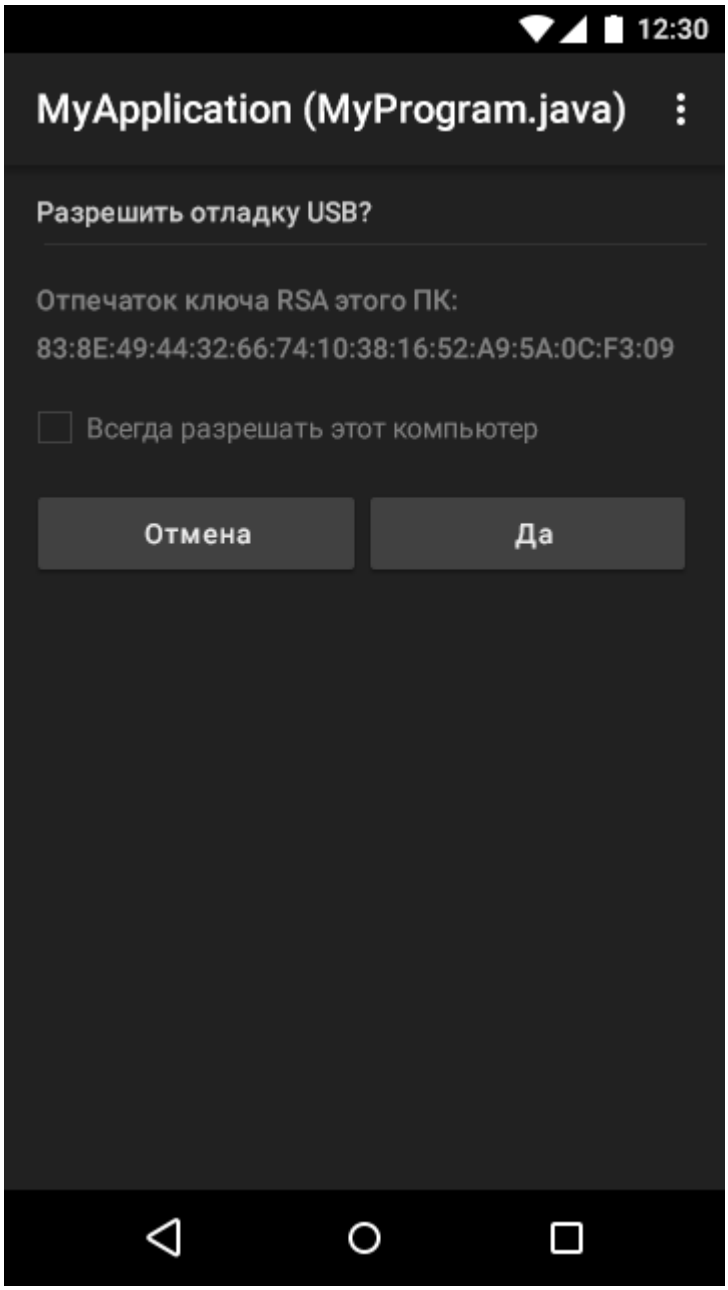
Прежде всего, нужно подготовить мобильное устройство к загрузке программ. Для этого проверьте, что:

- на компьютере разработчика установлены ADB драйвера для этого устройства;
- на самом устройстве включена отладка по USB (см. «Параметры разработчика» в настройках). На некоторых устройствах с ОС Android 4.2 и выше в настройках отсутствует этот пункт. Чтобы его открыть, найдите пункт «Об устройстве» (или «О телефоне»), в нем нажмите на «Номер сборки» 7 раз подряд, после чего откроется меню «Параметры разработчика»;
- устройство подключено к компьютеру проводом USB.

Эмулятор Android-устройства
В составе Android SDK имеется утилита ***Android Virtual Device (AVD)*** — эмулятор мобильного устройства, запускаемый на компьютере. Эмулятор нужен для отладки и тестирования приложения для разных устройств непосредственно в среде разработки. Можно создать несколько устройств с разными параметрами конфигурации и даже с разными версиями системы Android. Создать эмулятор можно в командной строке при помощи утилиты *android.bat,* находящейся в каталоге tools или с использованием Менеджера виртуальных устройств (AVD Manager), входящего в состав любой среды разработки для Android. В среде Eclipse менеджер виртуальных устройств AVD Manager вызывается командой меню *Window | Android Virtual Device Manager* либо кнопкой в панели инструментов. В среде Android Studio AVD Manager вызывается командой меню *Tools -> Android -> AVD Manager* или кнопкой в панели инструментов.

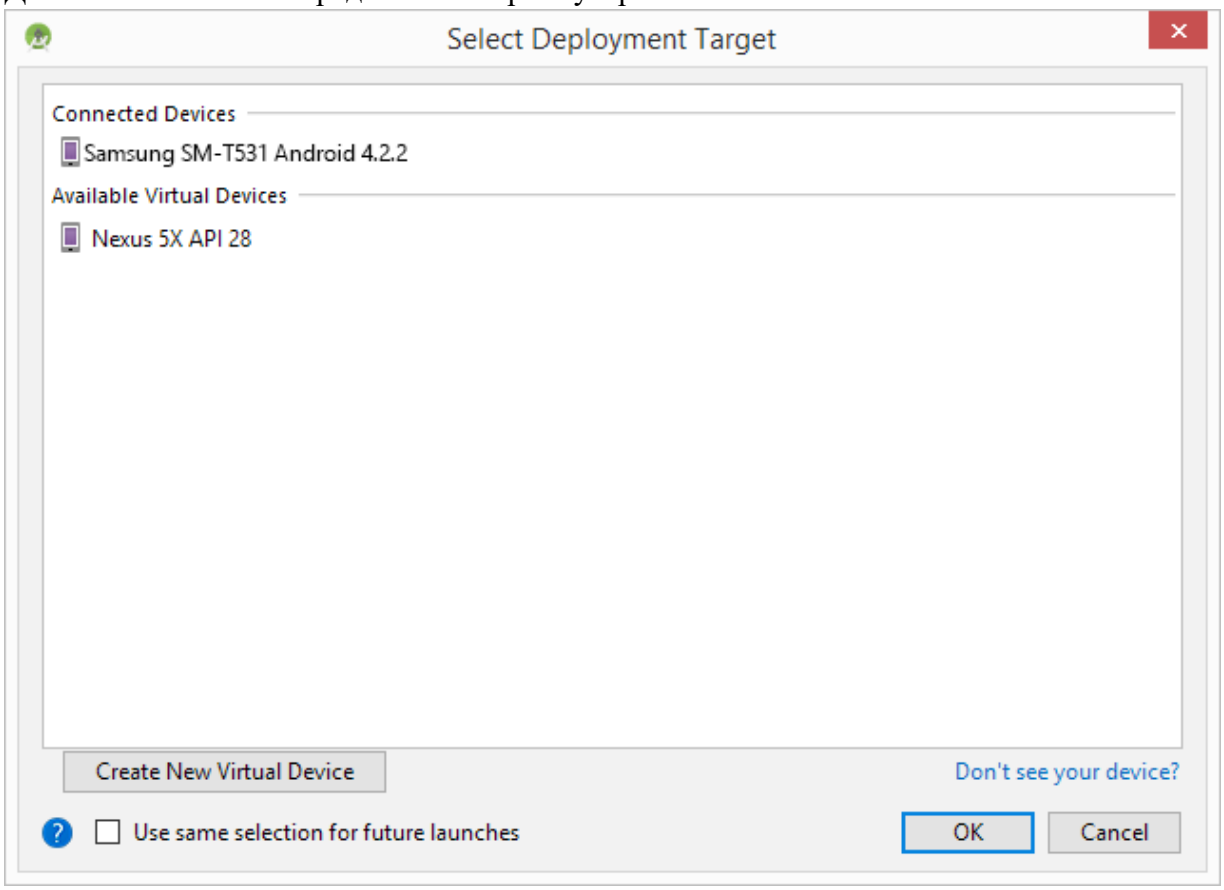
Запуск приложения на Android-устройстве

1. После подготовки Android-устройства при подключении через USB на устройстве появится сообщение вида:



2. Запустите приложение в Android Studio, нажав SHIFT+F10.

3. Далее Android Studio предложит выбрать устройство.

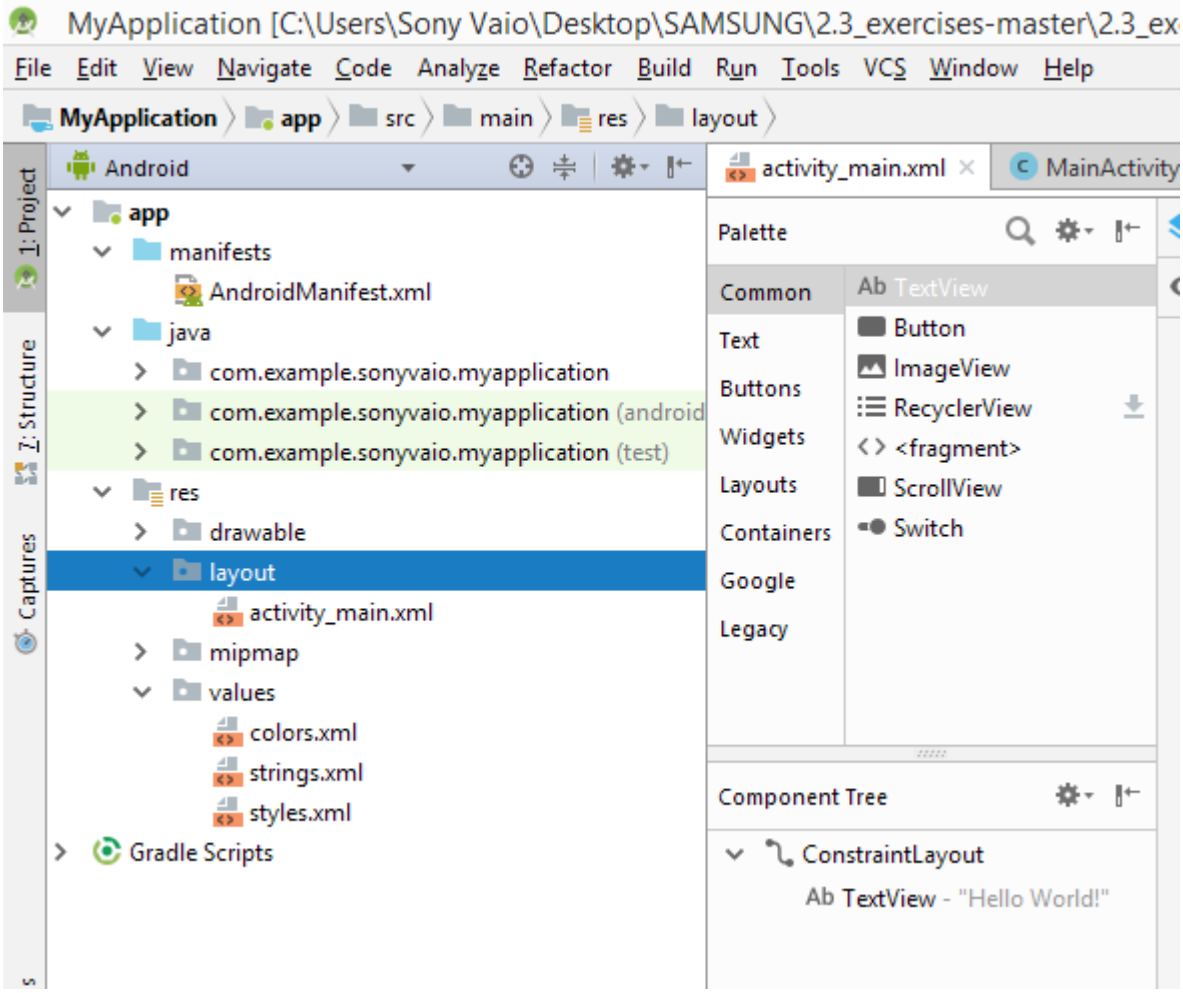


4. А на экране устройства запустится приложение с сообщением «Hello world!».

2.4.4. Структура проекта

Итак, что же получилось?

Как мы уже знаем, для отображения структуры проекта в Android Studio предусмотрено окно Project. Структура проекта меняется в зависимости от версии платформы, для которой ведется разработка (то есть от уровня API), но основные составляющие для всех проектов одинаковы.



Для IDE свойственно разделять составляющие проекта на:

- файлы кодов (программные java-файлы);
- файл разметки;
- файлы ресурсов (гипертекстовые xml-файлы).

Как уже говорилось, имена файла разметки и файла кодов можно задать при создании проекта. В нашем проекте это имена *MainActivity.java* и *activity_main.xml,* которые Android Studio (если быть точным, то Android SDK) сгенерировал по умолчанию.

Файл java-кода *MainActivity.java* располагается в пакете проекта, содержащемся в папке */app/src/main/java/com/example/myapplication* (у нас это пакет *com.example.myapplication*). Именно в этом файле создаются классы объектов и описываются методы работы с ними.

Файлы разметки и ресурсов проекта хранятся в папке */app/src/main/res/.* Ресурсных файлов в проекте много: это поддержка стилей, размеров и строковых констант для разных языковых настроек (папки *values-**), графики для разных параметров экрана (папки *drawable-**), разметки меню (папка *menu*) и непосредственно разметки элементов пользовательского интерфейса (папка *layout*), упакованного содержимого БД.

Отдельно расположенный файл *AndroidManifest.xml* содержит информацию о компонентах приложения, предназначенную для операционной системы. Файл манифеста позволяет определить, что за компонент и каковы условия, при которых он может быть запущен,

поэтому, если какой-либо компонент не описан в этом файле, то система не сможет с ним работать. Android Studio создает и редактирует файл манифеста автоматически, то есть нам не нужно заботиться о его заполнении.

Код приложения являет собой «активную» часть приложения. Ресурсы отделены от кода, что дает следующие преимущества:

- упрощается командная работа над проектом;
- изменение внешнего вида приложения зачастую вообще не требует модификации кода;
- локализация приложения для разных стран требует, как правило, всего лишь перевода строк из UI (хранящихся в строковых ресурсах) на язык нужной страны;
- упрощается адаптация внешнего вида приложения к разнообразным экранам.

Пример 2.3

Редактирование файлов ресурсов.

1. В созданном проекте MyApplication измените текстовые константы в файле *res/values/string.xml*:
   - «Hello, World» на строку «Hello, Name», где вместо Name поставьте свое имя;
   - «app_name» на строку «PROJECT ANDROID».
2. Сохраните проект и запустите приложение на планшете.
3. Скопируйте папку *res/values* и назовите копию *res/values-ru.* Переведите значения строковых констант в файле *string.xml* на русский язык и сохраните проект. Переключите на планшете язык на русский и запустите приложение.

2.4.5. Активности (Activity)

Каждая активность — это экран (по аналогии с web-формой), который приложение может показывать пользователям. Чем сложнее создаваемое приложение, тем больше экранов (активностей) потребуется. При создании приложения потребуется, как минимум, начальный (главный) экран, который обеспечивает основу пользовательского интерфейса приложения. При необходимости этот интерфейс дополняется второстепенными активностями, предназначенными для ввода информации, ее вывода и предоставления дополнительных возможностей. Запуск (или возврат из) новой активности приводит к «перемещению» между экранами UI.

Большинство активностей проектируются таким образом, чтобы использовать все экранное пространство, но можно также создавать полупрозрачные или плавающие диалоговые окна.

Создание активности

Для создания новой активности наследуется класс Activity или его подкласс (ListActivity, FragmentActivity и т. п.). Внутри реализации класса необходимо определить пользовательский интерфейс и реализовать требуемый функционал. В настоящий момент при создании активности без Фрагментов Android SDK автоматически генерирует следующий код:

```
import android.app.Activity;
import android.os.Bundle;
public class MainActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}
```

Базовый класс Activity представляет собой пустой экран, который не особенно полезен, поэтому первое, что вам нужно сделать, это создать пользовательский интерфейс с помощью Представлений (View) и разметки (Layout).

Представления (View) — это элементы UI, которые отображают информацию и обеспечивают взаимодействие с пользователем. Android предоставляет несколько классов разметки (Layout), называемых также View Groups, которые могут содержать внутри себя несколько Представлений, для создания пользовательского интерфейса приложения.

Чтобы назначить пользовательский интерфейс для активности, внутри обработчика событий onCreate используется метод setContentView().

Жизненный цикл активности

Приложения Android не могут контролировать свой жизненный цикл, ОС сама управляет всеми процессами и, как следствие, активностями внутри них. При этом, состояние активности помогает ОС определить приоритет родительского для этой активности Приложения (Application). А приоритет приложения влияет на то, с какой вероятности его работа (и работа дочерних активностей) будет прервана системой.

Стеки активностей

Состояние каждой активности определяется ее позицией в стеке (LIFO) активностей, запущенных в данный момент. При запуске новой активности представляемый ею экран помещается на вершину стека. Если пользователь нажимает кнопку «назад» или эта

активность закрывается каким-то другим образом, на вершину стека перемещается (и становится активной) нижележащая активность.

На приоритет приложения влияет его самая приоритетная активность. Когда диспетчер памяти ОС решает, какую программу закрыть для освобождения ресурсов, он учитывает информацию о положении активности в стеке для определения приоритета приложения.

Состояния активностей

Активности могут находиться в одном из четырех возможных состояний.

Активное (Active). Активность находится на переднем плане (на вершине стека) и имеет возможность взаимодействовать с пользователем. Android будет пытаться сохранить ее работоспособность любой ценой, при необходимости прерывая работу других активностей, находящихся на более низких позициях в стеке для предоставления необходимых ресурсов. При выходе на передний план другой активности работа данной активности будет приостановлена или остановлена.

Приостановленное (Paused). Активность может быть видна на экране, но не может взаимодействовать с пользователем: в этот момент она приостановлена. Это случается, когда на переднем плане находятся полупрозрачные или плавающие (например, диалоговые) окна. Работа приостановленной активности может быть прекращена, если ОС необходимо выделить ресурсы активности переднего плана. Если активность полностью исчезает с экрана, она останавливается.

Остановленное (Stopped). Активность невидима, она находится в памяти, сохраняя информацию о своем состоянии. Такая активность становится кандидатом на преждевременное закрытие, если системе потребуется память для чего-то другого. При остановке активности разработчику важно сохранить данные и текущее состояние пользовательского интерфейса (состояние полей ввода, позицию курсора и т. д.). Если активность завершает свою работу или закрывается, он становится неактивным.

Неактивное (Inactive). Когда работа активности завершена, и перед тем, как она будет запущена, данная активности находится в неактивном состоянии. Такие активности удаляются из стека и должны быть (пере)запущены, чтобы их можно было использовать.

Изменение состояния приложения — недетерминированный процесс и управляется исключительно менеджером памяти Android. При необходимости Android вначале закрывает приложения, содержащие неактивные активности, затем остановленные и, в крайнем случае, приостановленные.

Для обеспечения полноценного интерфейса приложения, изменения его состояния должны быть незаметными для пользователя. Меняя свое состояние с приостановленного на остановленное или с неактивного на активное, активность не должна внешне меняться. При остановке или приостановке работы активности разработчик приложения должен обеспечить сохранение состояния активности, чтобы его можно было восстановить при выходе активности на передний план. Для этого в классе Activity имеются обработчики событий, переопределение которых позволяет разработчику отслеживать изменение состояний активности (см. рис. 2.3).

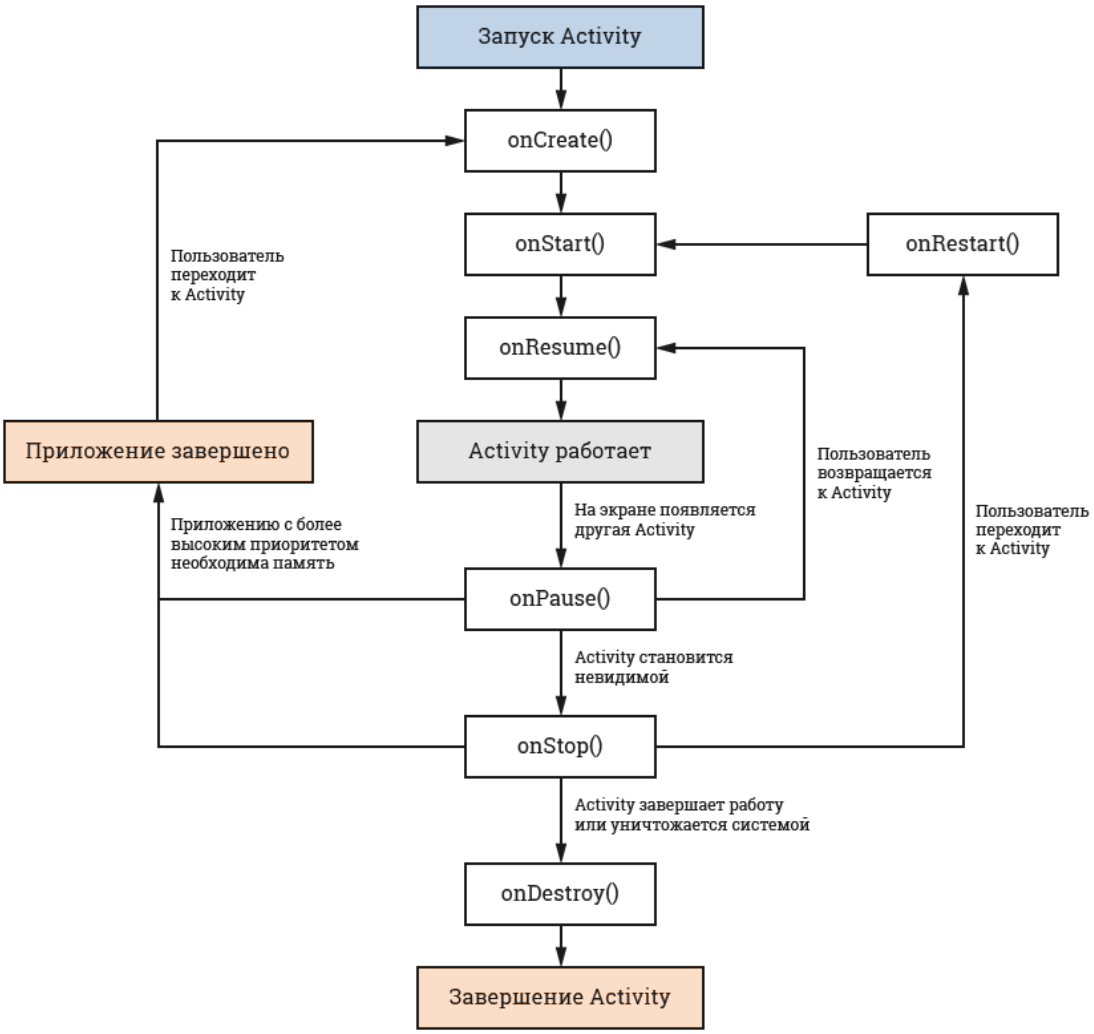


Рис. 2.3.

Отслеживание изменений

Состояния активности обработчики событий класса Activity позволяют отслеживать изменения состояний соответствующего объекта Activity во время всего жизненного цикла. Ниже показан пример с заглушками для таких методов — обработчиков событий:

```
public class ExampleActivity extends Activity {
// Вызывается при создании активности
@Override
public void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
// Инициализирует активность.
}
// Вызывается после завершения метода onCreate
// Используется для восстановления состояния UI
@Override
protected void onRestoreInstanceState(Bundle savedInstanceState) {
super.onRestoreInstanceState(savedInstanceState);
}
// Вызывается, когда активность стала видимой
@Override
protected void onStart() {
super.onStart();
// Проделать необходимые действия для активности, видимой на экране
}
// Должен вызываться в начале видимого состояния.
// На самом деле Android вызывает данный обработчик только
// для активностей, восстановленных из неактивного состояния
@Override
protected void onResume() {
super.onResume();
// Восстановить приостановленные обновления UI,
// потоки и процессы, приостановленные, когда
// активность была в неактивном состоянии
}
// Вызывается перед выходом из активного состояния,
@Override
protected void onSaveInstanceState(Bundle savedInstanceState) {
super.onSaveInstanceState(savedInstanceState);
}
// Вызывается перед выходом из активного состояния
@Override
protected void onPause() {
super.onPause();
// Приостановить обновления UI, потоки или трудоемкие процессы,
// ненужные, когда активность не на переднем плане
}
// Вызывается перед выходом из видимого состояния
@Override
protected void onStop() {
super.onStop();
// Приостановить обновления UI, потоки ненужные, когда активность не на переднем плане.
// Сохранить все данные и изменения в UI.
}
// Вызывается перед уничтожением активности
@Override
protected void onDestroy() {
super.onDestroy();
// Освободить все ресурсы, включая работающие потоки, соединения с БД.
```

```
}
}
```

Пример 2.4

Отслеживание состояния активности

1. Создайте проект Lifecycle и в файле кода MainActivity.java и измените метод onCreate().

```
@Override
protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Toast.makeText(this, "create", Toast.LENGTH_SHORT).show();
        }
```

Здесь super.onCreate() — это вызов конструктора класса-родителя Activity, а класс android.widget. Toast (класс всплывающих окон) — используется для вывода сообщения «create» на экран.

2. Переопределите остальные методы жизненного цикла Android-приложения. Сохраните и запустите проект на планшете. Понаблюдайте за всплывающими окнами в процессе работы приложения. Изучите другие объекты класса Toast и посмотрите их работу в вашем приложении.

```
@Override
protected void onStart(){
        super.onStart();
        Toast.makeText(this,"start",Toast.LENGTH_SHORT).show();
        }
```

Пример 2.5

Рассмотрим пример приложения Equation, решающего линейное уравнение ax+b=c:

```
public class MainActivity extends Activity {
    // Вызывается при создании активности
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // Инициализирует активность.
        setContentView(R.layout.activity_main);
    }

    /** Вызывается при нажатии пользователем на кнопку Решить */
    public void solveEquation(View view) {
        // ax+b=c
        double a = Double.parseDouble( ((EditText)
                findViewById(R.id.coefficient_a)).getText().toString());
        double b = Double.parseDouble( ((EditText)
                findViewById(R.id.coefficient_b)).getText().toString());
        double c = Double.parseDouble( ((EditText)
                findViewById(R.id.coefficient_c)).getText().toString());
        TextView result = (TextView) findViewById(R.id.result);
        result.setText("" + String.valueOf((c - b) / a));
    }
}
```

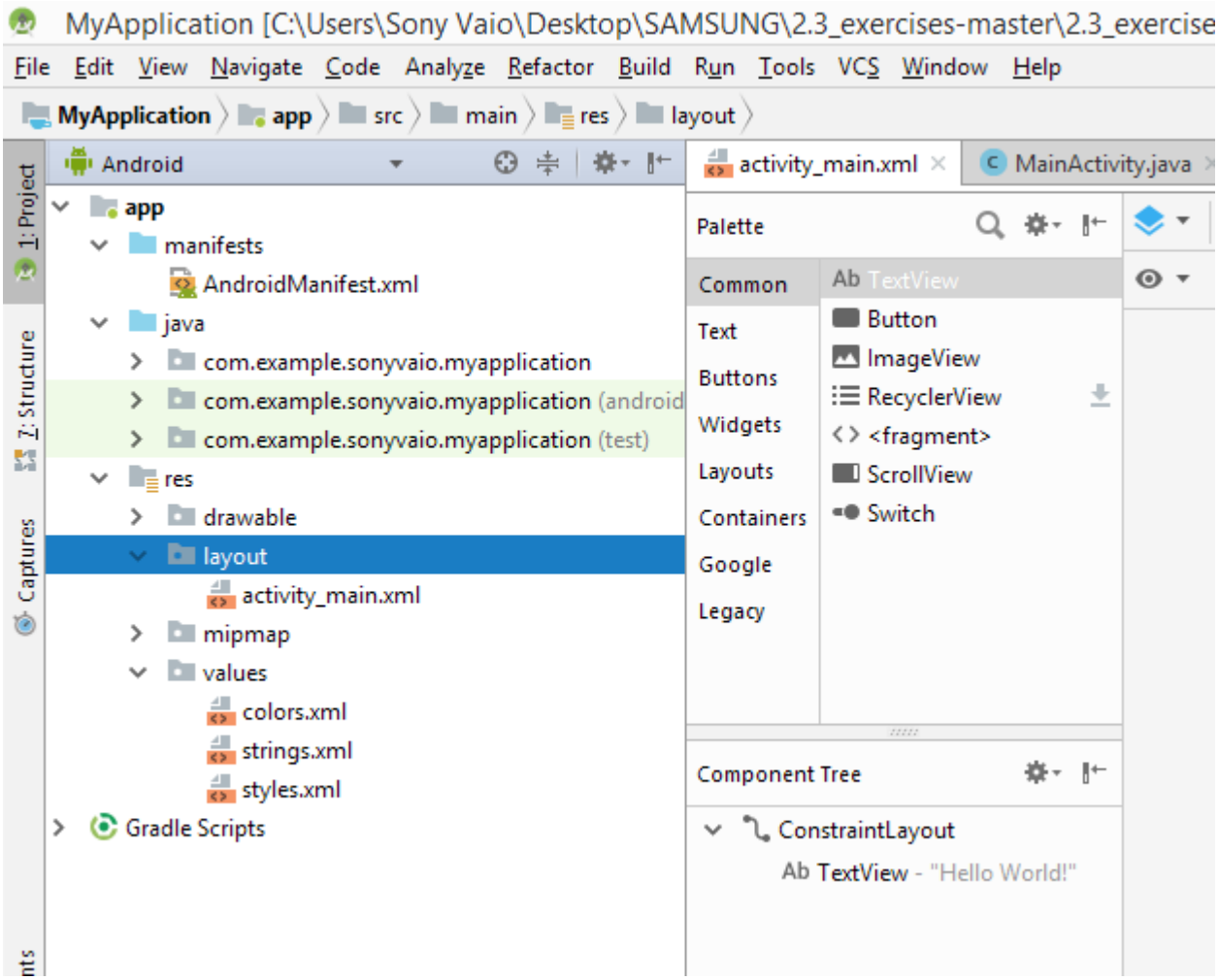
2.5. Интерфейс Android-приложения

Сайт: IT Академия SAMSUNG
Курс: MDev @ IT Академия Samsung
Книга: 2.5. Интерфейс Android-приложения
Напечатано:: Егор Беляев
Дата: Суббота, 18 Апрель 2020, 19:16

Оглавление

2.5.1. Структура проекта

На предыдущем уроке было создано наше первое Android-приложение. Давайте еще раз посмотрим его структуру. Итак, проект содержит папки:



- *src* — папка, в которой хранятся java-файлы, называемые файлами кода. Как правило, для каждого класса создается свой файл кода, но иногда разработчики располагают все классы в одном файле (не рекомендуем так делать, поскольку код становится неудобочитаемым);
- *res* — папка, в которой хранятся все ресурсы приложения и разметки экрана. Ресурсами являются текстовые константы, картинки, конфигурации экрана устройства (включая управляющие элементы), меню приложения, анимации, описание стилей объектов и подобные элементы. Все файлы ресурсов представляют собой гипертекстовые xml-файлы.
- прочие папки и файлы, содержащие, например, файлы системы автоматической сборки Gradle; автоматически созданные файлы юнит-тестов; промежуточные файлы компиляции R.java, файлы *.class и полностью собранную APK. Многие из этих файлов, кстати, могут блокироваться почтовыми системами как исполняемые, если вы захотите отправить проект по почте, даже в заархивированном виде.

Структура и настройки приложения описываются в файле *AndroidManifest.xml,* так же, как и файлы ресурсов, написанных на языке гипертекстовой xml-разметки.

Заметим, что IDE создает только «заготовки» файлов проекта, заполнять же их предстоит разработчику самостоятельно. Для этого необходимо знать язык программирования Java для кодирования приложения и язык разметки гипертекста XML для описания ресурсов и разметки экрана.

2.5.2. Язык разметки XML

Как уже говорилось ранее для описания ресурсов Android-приложений используется язык разметки XML (Extensible Markup Language) — расширяемый язык разметки, описывающий класс объектов, называемых XML-документами. Если открыть любой такой файл в текстовом редакторе, то мы увидим структуру текста, очень похожую на HTML.

Несмотря на то, что XML-файл — это просто текстовый документ, и его можно открывать любым текстовым редактором, во всех средах разработки есть возможность создать/открыть такой документ в специальном XML-редакторе. Этот редактор оказывает программисту дополнительную помощь в проверке (validation) документа, подсвечивает различными цветами теги и содержимое (coloring), автоматически закрывает открытые теги и т. д.

Например:

```
<Page>
    <TextView >Это текст</TextView>
    <br/>
    <Button text="click my"/>
</Page>
```

Несмотря на внешнюю схожесть, не следует путать XML и HTML. Язык HTML (Hyper Text Markup Language) был предложен Тедом Нельсоном в 1963 году для представления нелинейных текстовых ресурсов. HTML был также языком разметки, то есть с его помощью можно было структурировать текст для его удобной визуализации. HTML и XML используют концепцию тега для разметки текста и поэтому внешне похожи. Однако предназначение их в корне разное. В отличие от HTML, который размечает текст для удобного представления на экране пользователя, XML размечает документ для структурирования представленной в нем информации.

XML (Extensible Markup Language, расширяемый язык разметки) — это универсальный язык разметки, предназначенный для описания структурированных данных. Такая форма структурированных данных часто используется для хранения информации и обмена информацией между программами.

Начнем с того, что XML — это язык свободного описания структур документов. То есть, если необходимо, чтобы в документе присутствовал какой-либо элемент, то мы для него определяем некоторый тег (маркер в тексте). Например, для описания элемента «текстовая строка» можно условиться использовать тег <string>, где первая метка указывает начало описания элемента, а вторая (со знаком /) — конец описания. Между парой тегов помещается текстовое представления содержимого элемента. Для каждого элемента применяется своя пара тегов, при этом однотипные элементы описываются одинаковой парой тегов. Таким образом, для описания двух строк нужны две пары тегов:

```
<string>это первая строка</string>
<string>это вторая строка</string>
```

В открывающем теге можно поместить атрибуты описываемого элемента, такие как цвет, размер, начертание, выравнивание и т. п., то есть описать особенности формируемого элемента. ***Атрибут*** — это свойство описываемого элемента. При этом у однотипных элементов полный набор атрибутов будет совпадать, но в описании можно использовать не все свойства. Каждому имени атрибута присваивается значение, записанное в виде текстовой строки, то есть заключенное в двойные кавычки. Разделяются свойства пробелом либо переносом строки. Вернемся к рассматриваемому примеру: разметка

```
<string color = “red” align = “center”>это первая строка</string>
```

описывает текстовую строку, написанную красным шрифтом (начертание и размер установлены по умолчанию, поскольку эти свойства не указаны при описании) с выравниванием в центре страницы

```
это первая строка
```

Правила построения XML-документа

Каким бы свободным не был стиль XML-документа, все-таки существуют правила его формирования.

- В языке XML все теги парные. Это значит, что у каждого открывающего тега обязательно должен присутствовать закрывающий тег. Это правило позволяет описывать вложенные элементы, то есть помещать внутри одного элемента другие. Если тело тега пусто, то два тега записываются в один, который завершается косой чертой:

```
<string color = “red” align = “center”/>.
```

- Документ может содержать декларацию — строку заголовка, в которой указывается версия языка и используемая текстовая кодировка:

```
<?xml version="1.0" encoding="utf8"?>
```

- Имена тегов могут содержать буквы, цифры и специальные знаки, такие как знак подчеркивания (_), но должны начинаться с буквы. Теги записываются с соблюдением регистра, поскольку XML регистрозависим.

```
<Message>верная запись</Message>
<message>верная запись</message>
<Message>неверная запись</message>
```

- Если возникает необходимость использования одинаковых имен элементов для разного типа структур документа, применяют понятие пространства имен. Чтобы различать такие элементы, необходимо задать соответствие — специальный уникальный идентификатор ресурса или URI с конкретным именем элемента. В качестве идентификатора чаще всего используется адрес своего (необязательно реально существующий) ресурса. Пространство имен определяется благодаря атрибуту xmlns в начальном теге элемента:

```
<string xmlns:string="http://my_strings/styles/new" />
```

- В XML-тексте комментарии выделяются тегами:

```
<!-- текст, не читаемый анализатором документа -->
```

Конструкции языка

Содержимое XML-документа представляет собой декларации, набор элементов, секций CDATA, комментариев, специальных символов и текстовых данных.

В языке XML нет предопределенных тегов. Теги в приводимых примерах (например, <string> или <message>) не определены ни в одном стандарте XML. Эти теги «изобретаются» автором XML-документа. С помощью XML автор должен сам определить как теги, так и структуру документа.

Элементы данных

Элемент — это структурная единица XML-документа. Границы элементов маркируются одинаковыми начальным и конечным тегами. Внутри этой границы может быть текстовая строка значения элемента. Элемент может быть также представлен пустым тегом, то есть не включающим в себя другие элементы и/или символьные данные. Например, заключая строку 2017 в пару тегов <year> и </year>, мы определяем непустой элемент, называемый year, содержимым которого является значение 2017. В общем случае в качестве содержимого элементов могут выступать как просто какой-то текст, так и другие, вложенные, элементы документа, секции CDATA, комментарии — практически любые части XML- документа.

Например, ниже приведено три элемента:

```
<string/>
<school>167</school>
<city>Samara</city>
```

А вот эти элементами не являются:

```
<school>
<city>
Samara
```

В случае подобного ошибочного написания элементов любая программа не сможет прочитать ваш XML-документ и выдаст сообщение об ошибочности документа: что-то вроде «XML document is invalid».

CDATA

Секции CDATA выделяют части документа, внутри которых текст не должен восприниматься как разметка. CDATA означает буквально «character data» — символьные данные. Разделы CDATA представляют собой способ включения в документ текста, который иначе бы интерпретировался анализатором как разметка. Например, такая потребность возникает при включения в документ примеров, содержащих разметку, или кода функций на каком-нибудь языке программирования. Например:

```
<![CDATA[
B Character Data block я могу
я могу использовать знаки двойных кавычек, больше, меньше и любые другие символы (along wit
h <, &, ', and ")
]]>
```

Вложение элементов

Помимо текстового значения элемент может включать другие элементы. Такие элементы называются дочерними (child) элементами. Дочерних элементов может быть несколько. Элемент, который окружает дочерний элемент, называется родительским (parent).
По естественным причинам у дочернего элемента может быть только один родительский. Важно, чтобы любой дочерний элемент располагался целиком внутри родительского. То есть пары открывающих и закрывающих тегов всех дочерних элементов должны быть заключены (окружены) парой открывающего и закрывающего тегов родительского элемента. В случае нарушения этого правила любая программа не сможет прочитать ваш документ и выдаст сообщение об ошибочности. Автор документа, вкладывая одни элементы в другие, задает иерархическую структуру внутри документа.

Специальные символы

Для того чтобы использовать в документе символы, которые могут интерпретироваться синтаксическими анализаторами как элементы конструкций языка (например, символ угловой скобки) и не вызвать при этом ошибок разбора документа, нужно использовать специальные идентификаторы в символьной либо числовой форме (см. табл. 2.5).

Идентификатор	Символ	Описание
<	<	меньше
>	>	больше
&amp;	&	амперсанд
&apos;	'	апостроф
&quot;	»	двойная кавычка

Табл. 2.5.

Или, например, десятичная форма записи: & #34; — апостроф. Строковые обозначения спецсимволов можно определять в XML-документе при помощи компонентов (entity). Сущности (entity) представляют собой фрагменты текста или специальные символы. Таким образом автор может создавать свои текстовые идентификаторы спецсимволов. Например, описав как сущность author имя автора компании, везде по тексту можно использовать &author;. Например, описать сущность можно в прологе:

```
<!DOCTYPE MyDocs SYSTEM "filename.dtd" [ <!ENTITY author "Linus" ]>
```

Пример 2.6

Создадим XML-файл universities.xml, описывающий университеты города Самары. Указываем, что самарские университеты расположены в городе Самара, который, в свою очередь, находится в России, используя для этого вложенность элементов XML:

```
<?xml version="1.0" encoding="utf-8"?>
<country>
    <name>Russia</name>
    <city>
        <!-- Рассматриваемый город -->
        <name>Samara</name>
        <universities>
            <!-- Университет 1 -->
            <university>
                <name>Самарский государственный университет</name>
            </university>
            <!-- Университет 2 -->
            <university>
                <name>Самарский государственный медицинский университет</name>
            </university>
        </universities>
    </city>
</country>
```

Когда такой файл будет использоваться какой-либо программой (или даже человеком), то для нахождения конкретной информации необязательно прочитывать всю информацию от начала до конца. Опираясь на знания об иерархии документа (то есть о том, какие элементы к каким могут быть дочерними), можно для нахождения конкретного университета найти нужную страну <country>, в ней нужный город <city> и в нем нужный университет <university>. После нахождения элемента искомого университета уже считывать его и все его дочерние элементы, если таковые имеются. Таким образом, поиск информации становится гораздо более эффективным, чем просто сплошной массив информации.

Как видно, в этом примере присутствует один элемент <country>, в который вложены все остальные. Такой элемент называют корневым.

Пример 2.7

Создадим XML-файл reaction.xml, описывающий химическую реакцию:

```
<?xml version="1.0" encoding="utf-8"?>
<reaction substance="аммиак">
    <title>аммиак</title>
    <formula>2NH3</formula>
    <formulaReaction> 2NH4Cl + Ca(OH)2 → CaCl2 + 2NH3 + 2H2O
    </formulaReaction>
    <composition>
        <ingredient>хлорид аммония</ingredient>
        <ingredient>гашеная известь</ingredient>
    </composition>
    <conditions>
        <heating />
        <catalyzer optional="true" />
    </conditions>
    <cautions>
        <gas>Выделяется едкий летучий газ. Работать под вытяжкой!</gas>
    </cautions>
</reaction>
```

2.5.3. XML-документ

Структура XML-документа

Получив сведения о различных синтаксических конструкциях, можно рассмотреть общую логическую структуру документа. Любой XML-документ оформляется согласно следующей структуре:

То есть состоит из трех разделов:

- пролог;
- тело документа;
- эпилог (на схеме выше не представлен).

Пролог

Любой XML-документ начинается с пролога. В пролог помещается описательная информация для всего документа в целом, получить которую может потребоваться программе-анализатору еще до обработки документа. К ней относятся:

- команды обработки;
- декларация типа документа;
- комментарий.

Пролог не является обязательной частью XML-документа, и его отсутствие не вызовет ошибок при считывании документа программой-анализатором. Однако считается хорошим стилем всегда помещать пролог в XML-документ.

Тело

Тело документа должно состоять ровно из одного элемента, называемого корневым. Все другие данные должны быть дочерними элементами этого одного единственного элемента документа. Этим обеспечивается древовидная структура данных в XML-документе. Корневой элемент является точкой входа для синтаксического анализатора при работе с данными, представленными в XML-документе.

Эпилог

Эпилог может состоять только из комментариев. Эпилог завершает XML-документ и не является обязательным.

Пример 2.8

Рассмотрим пример иерархии плоских геометрических фигур. Как подобную иерархию можно описать с помощью документа на языке XML?

Первым шагом должен стать выбор родительского элемента, и в данном примере — это элемент «фигура». Для этого элемента придумываем название соответствующего ему тега, например, <figure>. Тег с таким названием будет корневым элементом документа:

```
<?xml version="1.0" encoding="utf-8"?>
<figure>
...
</figure>
```

По отношению к элементу «фигура» дочерними являются «открытая» и «закрытая». С точки зрения XML-документа это означает вложенность тегов. Пусть тег для элемента «открытая» называется <opened>, а для «закрытая» — <closed>. Тогда XML-документ выглядит следующим образом:

```
<?xml version="1.0" encoding="utf-8"?>
<figure>
    <opened>
        ...
    </opened>
    <closed>
    ...
    </closed>
</figure>
```

Стоит обратить внимание, что несмотря на то что элементов <opened></opened> и <closed></closed> с точки зрения самого XML-документа может быть неограниченное количество, с точки зрения логики таких элементов может быть только по одному. И действительно, вряд ли в реальном мире можно отыскать открытые фигуры, чья «открытость» отличалась бы.

Рассмотрим теперь отдельно фигуры «открытые» и «закрытые». В «открытые» фигуры попало две — точка (будем обозначать <point>) и ломанная (<polyline>). Про каждую из этих фигур можно сказать, что у них могут быть следующие свойства: толщина и цвет. Точке могут соответствовать координаты, а ломанной — длина.

Как точек, так и ломанных может быть сколь угодно много, и на первый взгляд их все можно просто разместить внутри элемента <opened>:

```
<?xml version="1.0" encoding="utf-8"?>
<figure>
    <opened>
        <point color=”red” thick=”2”>(0;0)</point>
        <point color=”blue” thick=”1”>(1;5,4)</point>
        <polyline color=”green” thick=”1”>23,1</polyline>
        <polyline color=”green” thick=”10”>1,09</polyline>
    </opened>
    <closed>
        ...
    </closed>
</figure>
```

Однако, элементы, принадлежащие одному классу, принято группировать. Например:

```
<pointList>
    <point color=”red” thick=”2”>(0;0)</point>
    <point color=”blue” thick=”1”>(1;5,4)</point>
</pointList>
```

Тогда XML-документ приобретет вид:

```
<?xml version="1.0" encoding="utf-8"?>
<figure>
    <opened>
        <pointList>
            <point color=”red” thick=”2”>(0;0)</point>
            <point color=”blue” thick=”1”>(1;5,4)</point>
        </pointList>
        <polylineList>

            <polyline color=”green” thick=”1”>23,1</polyline>
            <polyline color=”green” thick=”10”>1,09</polyline>
        </polylineList>
    </opened>
    <closed>
        ...
    </closed>
        </figure>
```

Подобные рассуждения верны и для классов многоугольник и эллипс, и для их подклассов. Причем каждый конкретный представитель, например, класса эллипс, не обязан быть кругом. Это означает, что у элемента эллипс может не быть дочерних элементов.

```
<ellipseList>
    <ellipse filling=”red” border=”2” />
    <ellipse filling=”blue” border=”1”>5</ellipse>
    <ellipse filling=”green” border=”1”>
        <circleList>
            <circle>12,55</circle>
            <circle>0,1525</circle>
        </circleList>
    </ellipse>
        </ellipseList>
```

В результате XML-документ может иметь следующий вид:

```
<?xml version="1.0" encoding="utf-8"?>
<figure>
    <opened>
        <pointList>
            <point color=”red” thick=”2”>(0;0)</point>
            <point color=”blue” thick=”1”>(1;5,4)</point>
        </pointList>
        <polylineList>
            <polyline color=”green” thick=”1”>23,1</polyline>
            <polyline color=”green” thick=”10”>1,09</polyline>
        </polylineList>
    </opened>
    <closed>
        <ellipseList>
            <ellipse filling=”red” border=”2” />
            <ellipse filling=”blue”border=”1”>
                Фокальное расстояние = 5
            </ellipse>
            <ellipse filling=”none” border=”1”>
                <circleList>
                    <circle>радиус = 12,5</circle>
                    <circle>радиус = 0,152</circle>
                </circleList>
            </ellipse>
        </ellipseList>
        <polygonList>
            <polygon angle="4">
                <rectangleList type="square">
                    <rectangle measurement="sm">сторона =
                        20
                    </rectangle>
                    <rectangle measurement="m">сторона =
                        0,45
                    </rectangle>
                </rectangleList>
                <rectangleList type="rectangle">
                    <rectangle measurement="sm">площадь =
                        17
                    </rectangle>
                    <rectangle measurement="sm">площадь =
                        5,7
                    </rectangle>
                </rectangleList>
            </polygon>
            <polygon angle="5" />
        </polygonList>
    </closed>
</figure>
```

2.5.4. Описание ресурсов Android с помощью XML

Создавая приложение для Android, помимо написания программ на языке Java необходимо также работать с ресурсами. В экосистеме Android принято отделять такие файлы, как изображения, музыка, анимации, стили, макеты окон, строковые константы — в общем все части оформления GUI (Graphical User Interface — графический интерфейс пользователя) от программного кода. Большая часть ресурсов (за исключением мультимедийных) хранятся во внешних XML-файлах. При создании и развитии программного проекта внешние ресурсы легче поддерживать, обновлять и редактировать.

Как уже было показано, каждое приложение на Android содержит каталог для ресурсов res/. Доступ к информации в каталоге ресурсов из приложения осуществляется через класс R, который автоматически генерируется средой разработки.

Для удобства разработки системой создается общий реестр всех ресурсов в проекте. Им является статический класс R.java, который содержит статические подклассы для всех типов ресурсов, для которых был описан хотя бы один экземпляр. Сами же ресурсы представлены в них как публичные статические целочисленные константы с именем, соответствующим имени ресурса. Поскольку файл *R.java* генерируется автоматически при любых изменениях в каталоге */res*, то нет смысла его редактировать вручную, так как все изменения будут утеряны при следующей генерации.

В общем случае ресурсы представляют собой файл (например, изображение) или значение (например, заголовок программы), связанные с создаваемым приложением по имени ресурса. Удобство использования ресурсов заключается в том, что их можно заменять/изменять без изменения программного кода приложения или компиляции. Поскольку имена файлов для ресурсов фактически будут использованы как имена констант в R, то они должны удовлетворять правилам написания имен переменных в Java. Так как разработка ведется на различных ОС (Windows, Mac, Linux), то также есть еще ограничения. В итоге имена файлов должны состоять исключительно из букв в нижнем регистре, чисел и символов подчеркивания.

Чаще других используют следующие ресурсы: разметка (layout), строки (string), цвета (color) и графические рисунки (bitmap, drawable).

В Android используются два подхода к процессу создания ресурсов — первый подход заключается в том, что ресурсы задаются внутри файла и тогда его имя задается в месте его описания. Второй подход — ресурс задается в виде самого файла, и тогда имя файла уже и есть имя ресурса. Общая структура каталогов, содержащих ресурсы, выглядит следующим образом:

```
/res
    /drawable
        /*.png
        /*.jpg
        /*.gif
    /layout
        /*.xml
    /values
        /strings.xml
        /colors.xml
```

Чтобы создать XML-файл ресурсов в Android Studio, нужно в главном меню выбрать File > New > Values resource file.

2.5.5. Строковые ресурсы

Любой используемый в программе текст — это отдельный ресурс, такой же, как изображение или звук.

При создании нового приложения среда разработки (например, Android Studio) создает файл *strings.xml***,** в котором хранятся строки для заголовка приложения и выводимого им (приложением) сообщения. Можно редактировать данный файл, добавляя новые строковые ресурсы. Также можно создавать новые файлы с любым именем, которые будут содержать строковые ресурсы. Все эти файлы должны находиться в подкаталоге */res/values*. Число файлов может быть любым. Строковые ресурсы обозначаются тегом. Типичный файл выглядит следующим образом:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name"> Hello world!</string>
    <string name="hello_world">Hello world!</string>
</resources>
```

Где name — имя строкового ресурса, с помощью которого можно отличить один ресурс от другого, а также обратиться к данному ресурсу в программе или в другом файле xml.

Подход, при котором строковые ресурсы хранятся в отдельном файле, удобен по двум причинам.

1. Если одна и та же строка используется в нескольких частях программы, то благодаря использованию ссылки на текстовый ресурс в дальнейшем правку (если, например, захотели изменить текст какого-то заголовка) нужно будет делать только в одном исходном файле ресурса.
2. Создание отдельных файлов с текстовыми значениями удобно и для локализации приложения на несколько языков. Все, что нужно будет сделать, — это создать отдельный файл с переводом, например, на французский язык, и создать для него отдельную директорию *res/values-fr* (*fr* — сокращение для French).

При именовании строковых ресурсов необходимо придерживаться нескольких правил:

- название строкового ресурса должно состоять из строчных букв;
- в случае, если название состоит из двух или трех слов, то их рекомендуется разделять нижним подчеркиванием;
- в названиях рекомендуется использовать только латиницу.

Во многих случаях можно задействовать системные строковые ресурсы — это строки типа OK, Cancel и др. В таких ситуациях используется схожий формат (добавляется ключевое слово android):

```
android:text="@android:string/cancel"
```

Здесь часть строки вида @android:string — обращение к системному строковому ресурсу Android.

Вот список некоторых системных строковых ресурсов Android:

```
@android:string/cancel
@android:string/no
@android:string/ok
@android:string/yes
```

Теперь строковые ресурсы можно использовать в xml-файле разметки формы. Например, если открыть файл *activity_main.xml*, расположенный в *res/layout/,* то можно увидеть:

```
<TextView
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="@string/hello_world" />
```

Среда разработки Eclipse автоматически создает код для Activity, в котором размещается текстовый элемент GUI — TextView. В качестве значения для текста помещена ссылка на текстовый ресурс:

```
android:text="@string/hello_world"
```

Сам же ресурс находится в файле *res/values/strings.xml*:

```
<string name="hello_world">Hello world!</string>
```

В приложении в описании элемента GUI (например TextView) можно строку текста записать прямо в описании элемента. Это называется ***Hard Coded*** — то есть жестко заданная строка. Но такой подход не рекомендуется — вместо жестко написанных строк следует использовать соответствующие идентификаторы, что позволяет изменять текст строкового ресурса, не изменяя исходного кода. Пример жестко написанной строки:

```
android:text="нажми на меня"
```

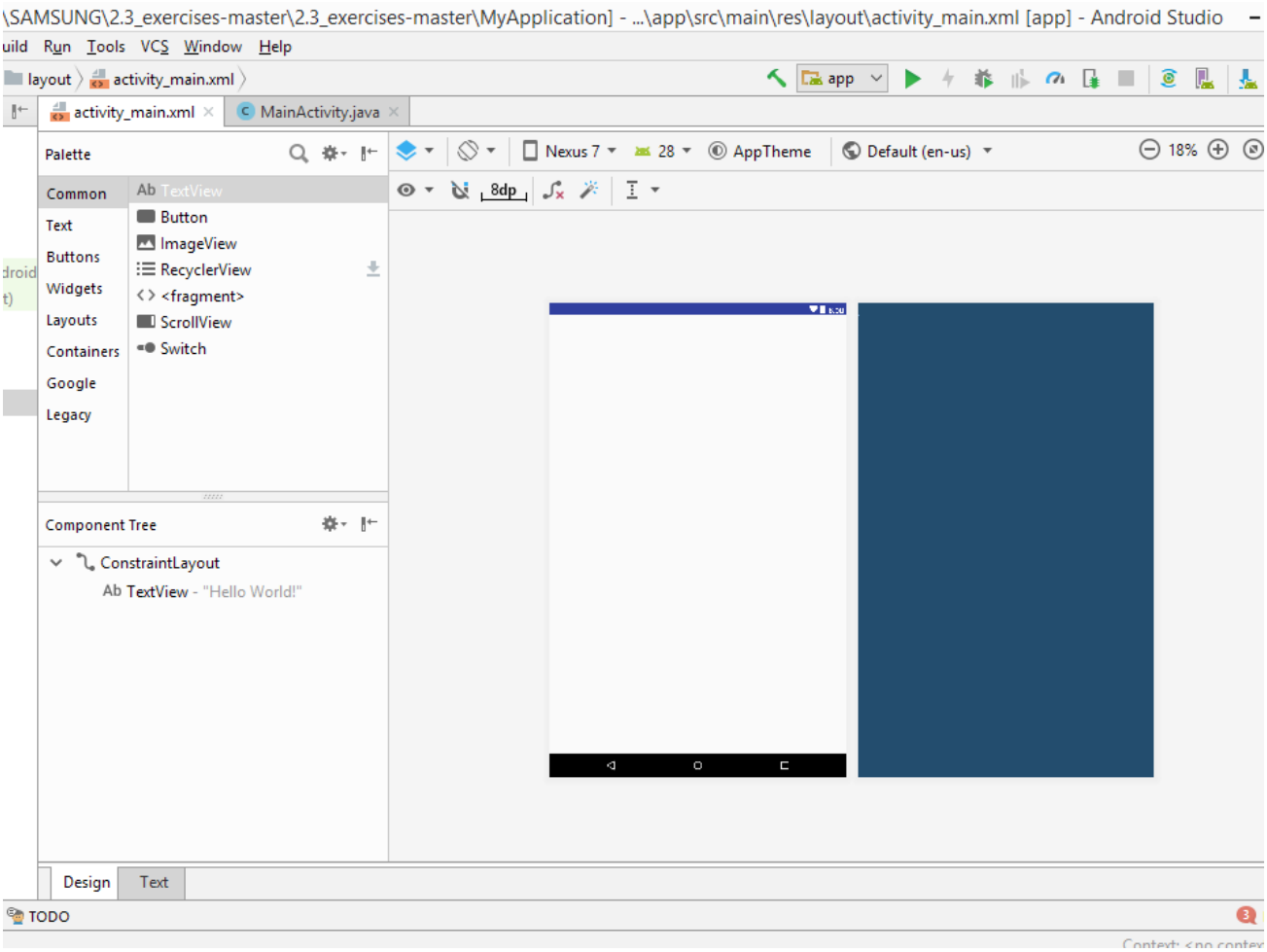
Пример строки с использованием строкового ресурса:

```
android:text="@string/close"
```

Применение ресурсов позволяет использовать механизм динамического выбора нужного ресурса в программе. Можно задать определенную структуру каталогов в проекте, чтобы создавать ресурсы для конкретных языков, регионов и аппаратных конфигураций. Во время выполнения программы Android выберет нужные значения для конкретного телефона пользователя.

2.5.6. Интерфейс пользователя. Разметка (layout)

Графический интерфейс создается с помощью представлений (View) и групп представлений (ViewGroup). Эти элементы размещаются на активности, их описания помещаются в файл манифеста, а действия с объектами прописываются программно в файле кода MainActivity.java в виде методов классов, наследуемых от классов View и ViewGroup или атрибутивно в файле разметки *layout/activity_main.xml*. У файла разметки также имеется графический вид Graphical layout — системная имитация мобильного устройства.



В окне Palette можно выбрать вид представления объектов: значки, названия, названия и значки. Выберите удобный для себя вид для работы с графическими представлениями.

Среда разработки при создании нашего проекта разместила в активности единственный объект — относительную разметку RelativeLayout с дочерним элементом — текстовой строкой TextView, значение которой находится в файле ресурсов *res/value/strings.xml* в текстовой константе string.

Мы уже знаем, как изменять значения таких констант. А как же изменить параметры самого представления? Во-первых, это можно сделать в графическом представлении, во-вторых, редактируя xml-файл разметки. Графический редактор очень прост в использовании, поэтому мы не будем останавливаться на его рассмотрении, а рассмотрим содержимое текстового файла.

Переключимся на вкладку activity_main.xml и посмотрим свойства объектов, которые присутствуют в нашей активности.

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:tools="http://schemas.android.com/tools"
android:layout_width="match_parent"
android:layout_height="match_parent"
android:paddingBottom="@dimen/activity_vertical_margin"
android:paddingLeft="@dimen/activity_horizontal_margin"
android:paddingRight="@dimen/activity_horizontal_margin"
android:paddingTop="@dimen/activity_vertical_margin"
tools:context="com.example.my_aap.MainActivity" >
    <TextView
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/hello" />
</RelativeLayout>
```

Layout — это разметка окна. Разметка может быть нескольких типов. Наиболее часто используемые:

- ***LinearLayout*** — линейная разметка, при которой элементы GUI (Views) размещаются вертикально или горизонтально. Расположение объектов в данной разметке осуществляется строго друг под другом в один столбец, если ориентация разметки вертикальная (android:orientation=«vertical»), или в одну строку рядом, если ориентация горизонтальная (android:orientation=«horizontal»). Линейная разметка позволяет объектам пропорционально заполнять пространство (android:layout_weight).
- ***FrameLayout*** — разметка в виде фреймов, когда все объекты выравниваются по левому верхнему углу экрана. Очень неудобна в качестве корневой, но можно применять как вложенную для размещения в ней одного объекта и резервирования свободного пространства рядом с ним.
- ***TableLayout*** — табулированная разметка, позволяющая располагать представления по строкам и столбцам. Каждый столбец таблицы в xml-файле ресурсов выделяется парным тегом <TableRow></TableRow>. В данной разметке во всех строках выделяется одинаковое количество столбцов. Если в строках было описано разное количество объектов, то длина строки измеряется по максимальному количеству элементов в TableRow. В строках меньшей длины крайние правые ячейки остаются свободными. Для того чтобы оставить пустой не крайнюю правую ячейку, в нее помещают пустой объект, например, TextView без текста и форматирования.
- ***RelativeLayout*** — относительная разметка, при которой позиции объектов можно определять относительно родительского элемента, а также относительно друг друга. Данный вид разметки используется по умолчанию.

С Android 4.0 в проектах появился новый вид разметки — GridLayout — табличная разметка, в которой элементы фиксируются в определенных ячейках таблицы, независимо от заполненности других ячеек. Все дочерние элементы в этой разметке имеют размеры по размеру данных, поэтому один объект может занимать несколько ячеек. Это свойство регулируется атрибутами android:layout_columnSpan и android:layout_rowSpan.

Для всех видов разметок обязательно имеются атрибуты высоты и ширины, при этом каждая разметка имеет свой набор атрибутов.

Разметка является корневым элементом активности. У окна может быть ровно один корневой элемент, в котором размещаются дочерние объекты. Потомком может быть представление любого типа, включая и тип родительского объекта.

Свойства описываются внутри парных тегов представления. Атрибут описывается пространством имен «android:» и названием свойства с присвоением ему значения: android:property = «property_value».

Описание свойств можно прочитать, выделив соответствующую строку в ниспадающем списке, который появляется при нажатии комбинации клавиш <Ctrl> + <Пробел> в процессе написания имени атрибута.

В составе Android SDK имеется утилита Hierarchy Viewer, позволяющая отследить разметку интерфейса пользователя. Запустить утилиту можно после запуска приложения на устройстве. Запустите файл *tool/hierarchyviewer.bat*, находящийся в папке установки *Android SDK*, и после его запуска выберите устройство, затем <Focused Window>, и нажмите кнопку Load View Hierarchy.

Часто используемые атрибуты объектов приведены в таблице 2.6.

Атрибут	Описание	Возможное назначение
android:id	Маркировка объекта по имени	“@+id/name” Знак + говорит о добавлении нового id объекта
android:orientation	Выравнивание разметки экране	vertical, horizontal
android:layout_width	Ширина объекта	fill_parent (match_parent) — родительский размер; wrap_content — по размеру содержимого; или фиксированный размер с указанием единиц измерения
android: layout_height	Высота объекта	fill_parent (match_parent) — родительский размер; wrap_content — по размеру содержимого; или фиксированный размер с указанием единиц измерения
android:layout_weight	Весовой коэффициент объекта	По умолчанию 0, или принимает натуральные значения — доля объекта в линии разметки
android:gravity android:layout_gravity	Выравнивание содержимого элемента или выравнивание самого элемента внутри родительского	center, left, right

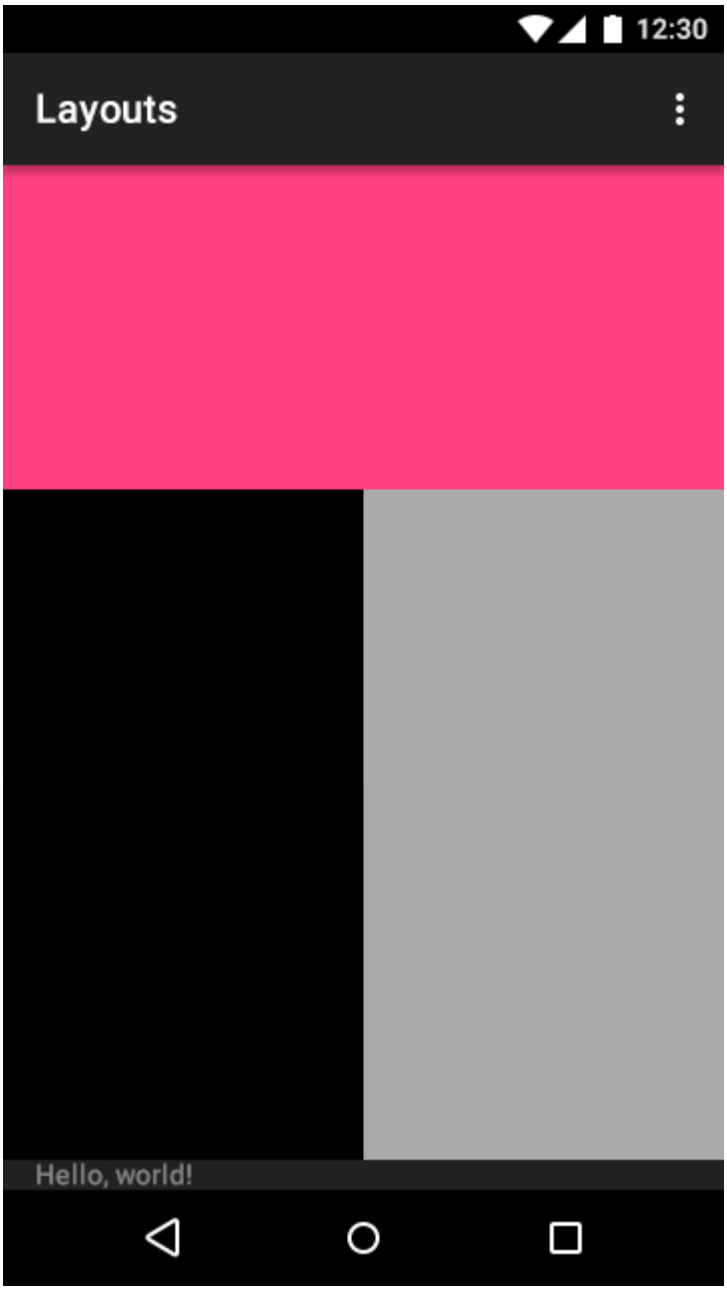
Табл. 2.6.

О практической стороне работы с разметками можно почитать, например, в заметках разработчиков:

- Android. Все о LinearLayout - 1
- Android. Все о LinearLayout - 2
- Android. Все о LinearLayout - 3

- Android. Обзор RelativeLayout
- Android. Обзор TableLayout
- Android. Обзор FrameLayout

Пример 2.9



Этот пример демонстрирует работу с атрибутами разметки Layout. Создайте проект Layouts. Измените относительную разметку на линейную (вертикальную), внутри нее создайте еще две разметки по ширине родительской разметки и по высоте, относящиеся 1:2 сверху вниз. Нижнюю разметку разделите на две равные вертикальные. Самым нижним элементом корневой разметки должно остаться текстовое поле. Установите для всех объектов различные значения цветового атрибута. Для наглядности установим фон различными цветами для различных лайоутов.

Получившийся файл *activity_main.xml*:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context="ru.samsung.itschool.book.layouts.MainActivity">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:background="@color/colorAccent"
        android:orientation="horizontal" />


    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_weight="2"
        android:background="@color/colorPrimary"
        android:orientation="horizontal">

        <LinearLayout
            android:layout_width="wrap_content"
            android:layout_height="match_parent"
            android:layout_weight="1"
            android:background="#FF000000"
            android:orientation="vertical" />

        <LinearLayout
            android:layout_width="wrap_content"
            android:layout_height="match_parent"
            android:layout_weight="1"
            android:background="#FFAAAAAA"
            android:orientation="vertical" />
    </LinearLayout>

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</LinearLayout>
```

2.5.7. Компоненты (View)

Разметка является основой для расположения визуальных компонентов (представлений, виджетов, View) приложения. В xml-файле разметки описание объекта помещается в парные теги <Тип_объекта> </Тип_объекта>. Для взаимодействия с объявленной на макете компонентой в java-файле необходимо импортировать стандартный класс из пакета android.widget. Тип_объекта, а затем создать константу или переменную этого типа и правильно инициализировать ее.

В программном коде доступ к компоненту осуществляется посредством его id, установленного атрибутом android:id, поэтому данный атрибут необходимо устанавливать всем создаваемым представлениям, к которым планируется обращение. Конечно, у компонентов значительно больше атрибутов. Например, у EditText имеется возможность установить подсказки (hint) о вводимой информации. Для этого необходимо добавить атрибут android:hint =«текст подсказки».

В файле разметки:

```
android:id = “@+id/name”
android:hint = “@string/text_hint”
```

В java-файле:

```
Type_object var_name = (Type_object)findViewById(R.id.name);
```

Здесь инструкция findViewById() находит компонент по его id, считывая адрес из класса R.

Класс R.java
Находится */app/build/generated/ source/r/debug/<packageName>/R.java* и по сути является реестром всех ресурсов, в том числе и компонентов, декларированных на макете (Layout) . В этом классе создаются в том числе константы для каждого вновь созданного (в разметке или в программном коде) компонента. При каждом запуске проекта данный класс создается снова. Для очистки класса R достаточно выполнить команду меню *Project|Clean.* При следующей компиляции файл *R.java* снова будет заполнен константами.

Текстовые объекты

Различают два вида текстовых полей: текстовая константа (TextView) и редактируемый текст (EditText).

(TextView) предназначается для отображения на экране стационарного фрагмента текста. Если содержимое не помещается в размеры объекта, то автоматически появляется полоса прокрутки. После создания в программе экземпляра текстового класса здесь же можно установить его атрибуты. Например, программным аналогом атрибута файла разметки android:text = “значение” является вызов метода setText(string):

```
android:text = “Hello!” <=> txt.setText (“Hello!”)
```

Мы видим, что для отображения текста в Textview в файле разметки используется атрибут android:text, например:

```
android:text="Результат:"
```

Как уже упоминалось, такой подход является нежелательным. Вместо этого лучше создать строковый ресурс в файле *strings.xml*:

```
<string name="res">Результат:</string>
```

А затем уже использовать этот строковый ресурс:

```
android:text="@string/res"
```

У компонента Textview есть многочисленные методы и XML-атрибуты для работы с текстом. Например, основные XML-атрибуты, отображающие свойства Textview:

1. android:textSize — размер текста. При установке размера текста используются несколько единиц измерения:
   - px — пикселы;
   - dp — независимые от плотности пикселы. Это абстрактная единица измерения, основанная на физической плотности экрана;
   - sp — независимые от масштабирования пикселы, то есть зависимы от пользовательских настроек размеров шрифтов;
   - in — дюймы, базируются на физических размерах экрана;
   - pt — 1/72 дюйма, базируются на физических размерах экрана;
   - mm — миллиметры, также базируются на физических размерах экрана.

   Обычно при установке размера текста используются единицы измерения sp, которые наиболее корректно отображают шрифты.

   ```
   android:textSize="48sp"
   ```

2. android: textStyle — стиль текста. Используются константы:
   - normal;
   - bold;
   - italic.

   Пример установки стиля через атрибуты:

   ```
   android:textStyle="bold"
   ```

3. android:textColor — цвет текста. Используются четыре формата в шестнадцатеричной кодировке:
   - #rgb;
   - #argb;
   - #rrggbb;
   - #aarrggbb.

   Где *r, g, b* — соответствующий цвет, *a* — прозрачность (alpha channel). Значение a, установленное в 0, означает прозрачность 100%. Значение по умолчанию, без указания значения alpha, — непрозрачно. Например, можно создать цветовой ресурс в файле colors.xml:

   ```
   <color name="text">#87CEFA</color>
   ```

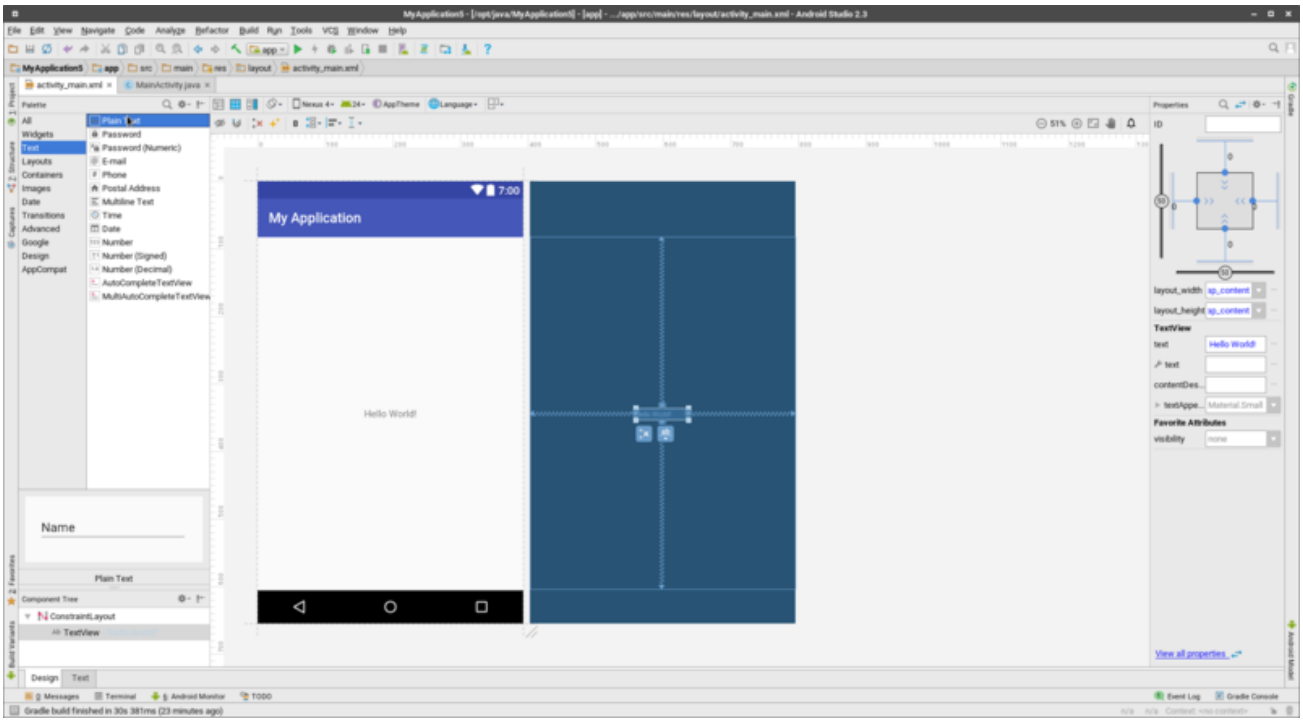
И тогда в атрибуте указываем:

```
android:textColor="@color/text"
```

Если же мы хотим реализовать ввод и редактирование текста, нам придется использовать объект класса EditText, основным методом которого является получение вводимого значения — метод getText(). Данный метод возвращает значение типа ***Editable*** — надстройка над ***String,*** обладающая возможностью динамического изменения в процессе выполнения программы. Приведем описание текстовых полей: простой текст с надписью “Hello!” и текстовое поле для ввода с подсказкой (hint) “Input your text”.

```
<TextView
android:id="@+id/textView1"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_alignParentLeft="true"
android:layout_alignParentTop="true"
android:layout_marginTop="31dp"
android:text="Hello!" />
<EditText
android:id="@+id/etext1"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_alignLeft="@+id/textView1"
android:layout_below="@+id/textView1"
android:hint="Input your text" >
</EditText>
```

Элемент интерфейса EditText — это текстовое поле для ввода пользовательского текста, которое используется, если необходимо редактирование текста. EditText является наследником TextView.



Как и для файлов ресурсов (например, *strings.xml*), так и для файлов разметки (*activity_main.xml*), когда открываем его в Android Studio, внизу можно заметить две вкладки: Design и Text. Открыв первую, на панели инструментов слева в папке *TextFields* можно найти текстовые поля под разными именами. Например, PlainText, Password, E-mail и т. д.

Для быстрой и удобной разработки текстовые поля снабдили различными свойствами и дали разные имена. Рассмотрим некоторые из них.

PlainText

PlainText — самый простой вариант текстового поля. При добавлении в разметку его XML-представление будет следующим:

```
<EditText
android:id="@+id/editText1"
android:layout_width="match_parent"
android:layout_height="wrap_content" />
```

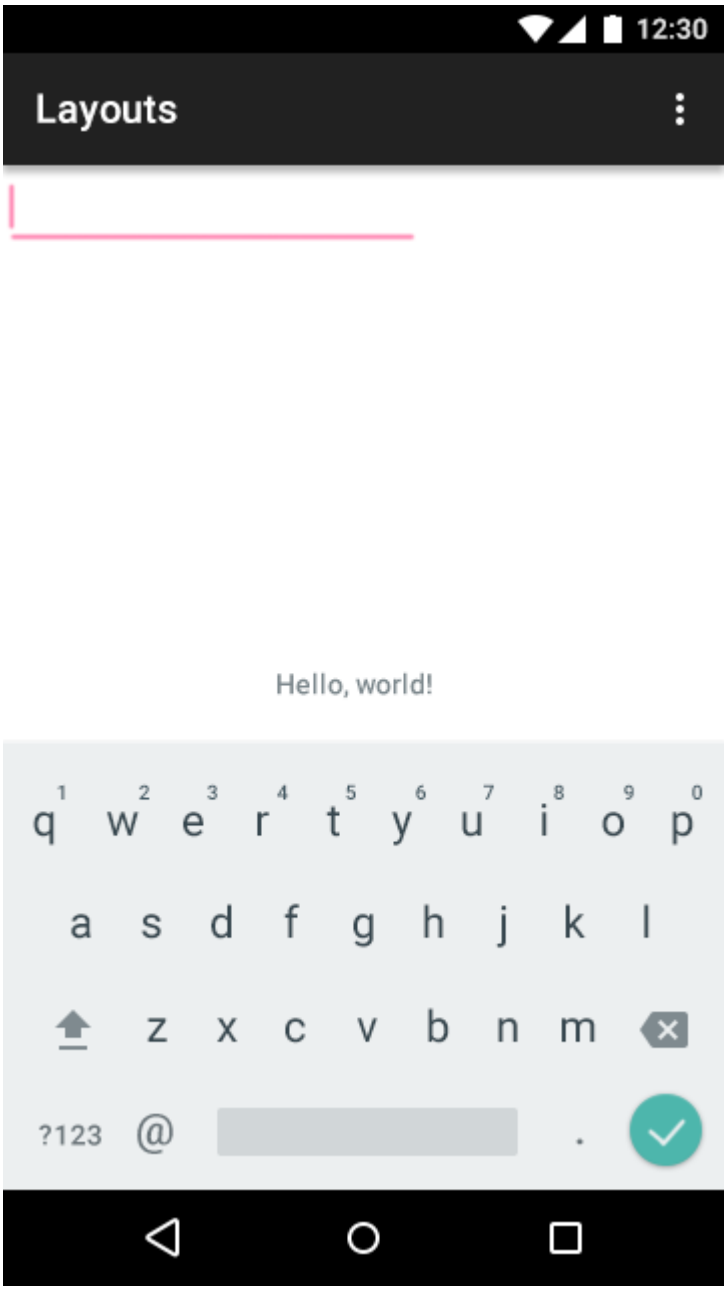
Password

При использовании типа Password в свойстве inputType используется значение textPassword. При вводе текста сначала показывается символ, который заменяется на звездочку или точку.

```
<EditText
android:id="@+id/editText2"
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:inputType="textPassword" />
```

E-mail

У элемента E-mail используется атрибут android:inputType="textEmailAddress". В этом случае на клавиатуре появляется дополнительная клавиша с символом @, который обязательно используется в любом электронном адресе.



Текст-подсказка

Подсказка видна, если текстовый элемент не содержит пользовательского текста. Как только пользователь начинает вводить текст, то подсказка исчезает. Соответственно, если удалить пользовательский текст, то подсказка появляется снова. Это очень удобное решение во многих случаях, когда на экране мало места для элементов.

В Android у многих элементов есть свойство (атрибут) Hint, который работает аналогичным образом. Установите у данного свойства нужный текст и у вас появится текстовое поле с подсказкой.

```
android:hint="подсказка"
```

Кнопки

Кнопка (Button) — это управляющий элемент. Кнопки позволяют управлять процессом выполнения приложения в ходе его работы. Кнопка описывается тегами <Button></Button> и является наследником класса android.widget.Button. Сам по себе объект Button ничего не делает, то есть при нажатии на кнопку ничего не происходит. Чтобы действия выполнялись, необходим обработчик события нажатия кнопки. Задать метод обработчик события можно двумя способами:

- декларативно, при помощи атрибута кнопки onClick;

- программно, в коде вашего приложения используя метод, устанавливающий обработчик событий для кнопки setOnClickListener().

Полный пример использования кнопок см. в примере 2.10.

Если же при описании кнопки мы применим теги <ImageButton></ImageButton>, то на кнопке вместо текста будет изображение, находящаяся в папке ресурсов drawable-* (в зависимости от параметров экрана будет выбрана одна из папок). В файле разметки установка изображения на кнопку осуществляется командой:

```
android:src = “@drawable/имя_файла_с_картинкой”
```

В программе — методом setImageResource(R.drawable.ic_launcher.png), где *ic_launcher.png* — имя файла с картинкой из папки drawable-*.

```
<ImageButton
android:id="@+id/button1"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:src="@drawable/abc_menu_dropdown_panel_holo_dark" />
```

Переключатели/включатели

Представления этого RadioButton позволяют выбирать одну из предложенных позиций и относятся к java-классу RadioGroup. Группа переключателей в файле разметки выделяется тегами <RadioGroup></RadioGroup>, каждый из переключателей описывается тегами <RadioButton></RadioButton>. В группе активным может быть только один переключатель. Если теги <RadioGroup> не ставить, то кнопки будут независимы друг от друга. Так делать не стоит, поскольку для независимых переключателей предусмотрен класс CheckBox. Атрибуты переключателей ничем не отличаются от атрибутов остальных виджетов. Определение группы переключателей из двух кнопок выглядит так:

```
<RadioGroup
android:id="@+id/radioGroup1"
android:layout_width="wrap_content"
android:layout_height="wrap_content">
<RadioButton
android:id="@+id/radio1"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:checked="true"
android:text="RadioButton 1"/>
<RadioButton
android:id="@+id/radio2"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="RadioButton 2"/>
</RadioGroup>
```

Для программирования поведения приложения при выборе определенной кнопки применяется метод isChecked(). При создании в приложении текстового поля с id равным textView1 и кнопки button1 можно задать событие на вывод текстовой строки в поле textView1 при выборе переключателя и нажатия кнопки button1.

```
public class MainActivity extends ActionBarActivity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout. activity_main);
        final RadioButton r1 = (RadioButton)findViewById(R.id.radio1);
        final RadioButton r2 = (RadioButton)findViewById(R.id.radio2);
        final Button b = (Button)findViewById(R.id.button1);
        final TextView t = (TextView)findViewById(R.id.textView1);
        b.setOnClickListener( new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                if (r1.isChecked()) t.setText("Case 1");
                if (r2.isChecked()) t.setText("Case 2"); }
        });
    }
}
```

CheckBox

Об этом классе объектов уже упоминалось ранее. Класс CheckBox не привязывает свои объекты к какой-либо группе, то есть представления работают независимо. Для описания в разметке используются теги <CheckBox></CheckBox>. Проверка включения кнопки проверяется тем же методом isChecked(), что и у RadioButton.

Другие представления

В активность приложения можно добавлять шкалу прогресса (ProgressBar), часы (AnalioClock, DigitalClock), календарь (Calendar), секундомер (Hronometer) и другие объекты. Такие объекты проще добавлять в графическом представлении разметки. Ознакомиться с полным списком виджетов можно на сайте разработчика developer.android.com.

Пример 2.10

Рассмотрим пример работы с виджетом кнопки Buttons. В этом примере необходимо разместить текстовое поле, поле ввода и три кнопки. При нажатии на кнопку «ввести» текст из поля ввода должен быть помещен в текстовое поле. При нажатии кнопки «отменить» в текстовое поле нужно поместить фразу «ничего не напишу». А при нажатии кнопки выход — покинуть программу. Для выполнения задачи на макете разместим три кнопки. На первые две зададим обработчик нажатия программно, а на последнюю декларативно. Итак, создадим следующую разметку экрана приложения и поместим в нее следующие текстовые поля и кнопки:

```
<TextView
        android:id="@+id/txt1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Текст" />

    <EditText
        android:id="@+id/etxt1"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Вводите текст сюда" />

    <Button
        android:id="@+id/bt1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Ввести" />

    <Button
        android:id="@+id/bt2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Отменить" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:onClick="quit"
        android:text="Выход" />
```

Откроем файл java-кода (если вы его не переименовали, то по умолчанию это MainActivity.java) и создадим объекты для виджетов:

```
public class MainActivity extends AppCompatActivity {
    //объявляем переменные для компонентов (виджетов)
    TextView txt;
    EditText etxt;
    Button bt1,bt2;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        // инициализируем переменные объектами связанными с ID виджетов
        txt = (TextView)findViewById(R.id.txt1);
        etxt = (EditText)findViewById(R.id.etxt1);
        bt1 = (Button)findViewById(R.id.bt1);
        bt2 = (Button)findViewById(R.id.bt2);
```

А теперь напишем обработчик событий нажатия на кнопку для первых двух кнопок (продолжаем работать в методе onCreate()):

```
//создаем обработчик
        View.OnClickListener listener=new View.OnClickListener() {
            @Override
            public void onClick(View v){
                switch(v.getId()){
                    case R.id.bt1: txt.setText(etxt.getText().toString()); break;
                    case R.id.bt2: txt.setText("ничего не напишу!"); break;
                }
            }
        };
```

Выполняемое действие зависит от того, какой id у нажатой кнопки. Для идентификации кнопки применяется метод getId(), возвращающий id виджета, на котором произошло событие клика. Далее установим созданный обработчик первым двум кнопкам:

```
bt1.setOnClickListener(listener);
        bt2.setOnClickListener(listener);
```

Конечно, можно создавать и устанавливать для каждой кнопки отдельный обработчик. В общем случае можно создавать любое необходимое количество обработчиков и устанавливать их для любого количества активных виджетов (необязательно кнопки).

Ну и самый простой способ вызова обработчика — это вызов его из атрибутов кнопки в xml-файле. Для этого нам не нужно создавать объект-обработчик, а достаточно просто определить метод, указанный в атрибуте onClick. В данном примере метод quit():

```
// декларативно заданный обработчик (атрибут onClick)
    public void quit(View view){
        finish();
    }
```

Обратите внимание на то, что в данном случае для обработчиков никаких объектов не создается, функция обработки задается на самом виджете. Такой же экономный способ написания обработчиков событий нажатия кнопок получается, если в качестве обработчика использовать Activity. Эту возможность будет рассмотрена далее при знакомстве с интерфейсами.

2.6. Наследование и полиморфизм в Java

Сайт: IT Академия SAMSUNG
Курс: MDev @ IT Академия Samsung
Книга: 2.6. Наследование и полиморфизм в Java
Напечатано:: Егор Беляев
Дата: Суббота, 18 Апрель 2020, 19:17

Оглавление

2.6.1. Понятие наследования

Одно из основных понятий ООП — это наследование. ***Наследование*** — это естественная абстракция для большинства людей, потому что с наследованием мы сталкиваемся в жизни, и возможность перенести принципы наследования в разработку программного обеспечения позволяет значительно упростить декомпозицию сложных систем и создавать такой уровень абстракции, который будет доступен человеку.

Наследование — это отношение между классами, при котором класс использует структуру или поведение другого класса.

Известно, что при разработке программы программист может держать в голове примерно 5–7 элементов, которыми он сможет одновременно оперировать.

Отсюда появляется необходимость в синтезе из нескольких мелких объектов одного более общего и крупного, или анализ массы объектов с целью выделить некоторые общие признаки.

Именно анализ группы объектов и выделение их общих свойств позволяет разработать некоторый обобщающий базовый класс, частные реализации которого уже дадут необходимую функциональность. Рассмотрим пример. Пусть вы разрабатываете компьютерную игру. В ней существуют различные персонажи: роботы, люди, волшебники и т. д. Объединим их по функции, для которой они предназначены (для участия в компьютерной игре), и выделим их в единый класс Unit.

Unit — это класс, который имеет общие поля и методы для всех персонажей в игре. Например, каждый персонаж имеет имя, индикатор здоровья и метод вывода информации о персонаже на экран. Описание класса может выглядеть следующим образом:

```
class Unit{
        private String name;
        private int health;

        public Unit(String name, int health) {
                this.name = name;
                this.health = health;
        }

        public void printInfo(){
                System.out.println("Name:" + name);
                System.out.println("Health:" + health);
        }
}
```

Теперь перечислим те классы, которые могут наследоваться от класса Unit: Robot (робот) и Wizard (волшебник), Warrior (воин). Функционально эти объекты предназначены для участия в игре, следовательно, они являются наследниками класса Unit. Помимо общих свойств, которые перешли из класса Unit, в наследниках можно добавлять поля и методы, характерные только для классов наследников. Например, для роботов — индикатор брони, волшебников — индикатор манны, воинов — метод атаки и т. д.

Таким образом, мы рассмотрели понятия, которые можно описать следующим определением. ***Наследование*** — принцип объектно-ориентированного программирования, в рамках которого возможно описание новых классов на основе уже существующих таким образом, что свойства (в Java — поля) и методы родителя будут присутствовать и в наследниках. Родительский класс в данном случае принято называть суперклассом.

Для описания наследования на языке Java используется ключевое слово extends, которое на русский переводится как «расширяет».

Например, наследование может выгладить так:

```
public class Robot extends Unit{
}
```

В данном примере класс Robot наследуется от класса Unit. Обратите внимание, что фраза "class Robot extends Unit" означает "Класс Robot расширяет Unit".

Ранее мы уже сталкивались с ключевым словом extends при создании Android-проекта

```
public class MainActivity extends Activity {
}
```

По умолчанию все классы в Java являются потомками класса Object. Поэтому все описанные нами классы уже обладают атрибутами и методами класса Object. Определения класса, приведенные ниже, являются идентичными:

```
public class MyClass { … }
public class MyClass extends Object { … }
```

В Java существует возможность не только расширять функциональность класса путем наследования. У классов-наследников методы родительского класса можно также переопределять. Ниже показан пример переопределения метода printInfo() для класса Robot.

```
class Robot extends Unit{
        @Override
        public void printInfo(){
                System.out.println("I am Robot");
        }
        }
```

Ранее нам встречалась перед методами строка @Override. Это аннотация, которая появилась в Java SE5. Если вы собираетесь переопределить метод, тогда рекомендуется ее использование. Компилятор выдаст сообщение об ошибке, если вместо переопределения будет выполнена перегрузка метода.

2.6.2. Графическое описание структуры классов в UML

Текстовое восприятие описания классов крайне тяжело воспринимать и поэтому на практике для описания их структуры (в частности, такого отношения, как наследование) используется специальный графический язык описания ***UML (Unified Modelling Language)***. В рамках данного учебника нет задачи научить полноценно использовать диаграммы классов из стандарта UML, но базовые понятия будут полезны для дальнейшего разбора примеров и постановки задач. Основным элементом диаграммы классов является блок, представляющий собой класс. Ниже представлен блок класса с двумя свойствами и тремя методами (см. рис. 2.4).

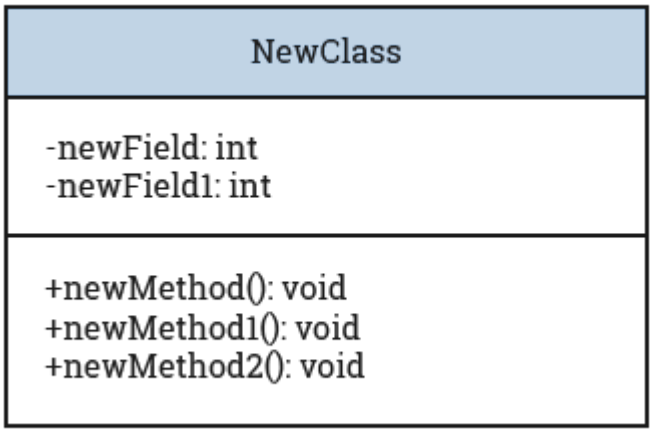


Рис. 2.4.

Блок класса представляет из себя прямоугольник, разбитый на три части: название (в примере выше — NewClass), поля класса (newField, newField1) и методы (newMethod(), newMethod1(), newMethod2()).

Как видите на картинке, каждое свойство класса имеет имя и тип (указывается после двоеточия). Также каждый метод и свойство имеют знак области видимости:

- + – public (публичный);
- – private (приватный);
- # – protected (защищенный).

Публичные и приватные методы и поля классов мы разобрали ранее, а защищенные рассмотрим чуть ниже в этой главе.

Классы не единственное, что можно отобразить на диаграмме классов. Другим важным элементом данной диаграммы является возможность показать на ней отношения между этими классами. Отношение наследования на диаграмме будет отображаться следующим образом (см. рис. 2.5).

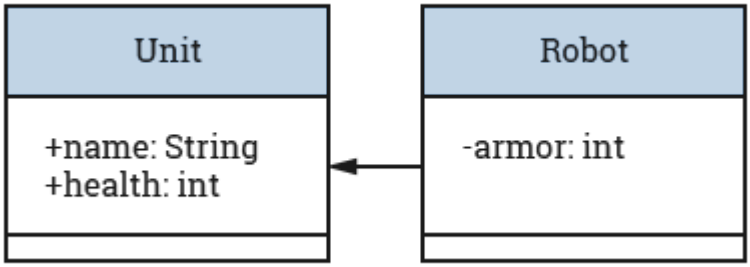


Рис. 2.5.

На данном рисунке есть суперкласс Unit и его наследник Robot. Таким образом, чтобы указать отношение наследования необходимо провести линию от наследника к суперклассу и нарисовать белый треугольник в конце.

Разберем пример с персонажами компьютерной игры и нарисуем диаграмму классов (см. рис. 2.6).

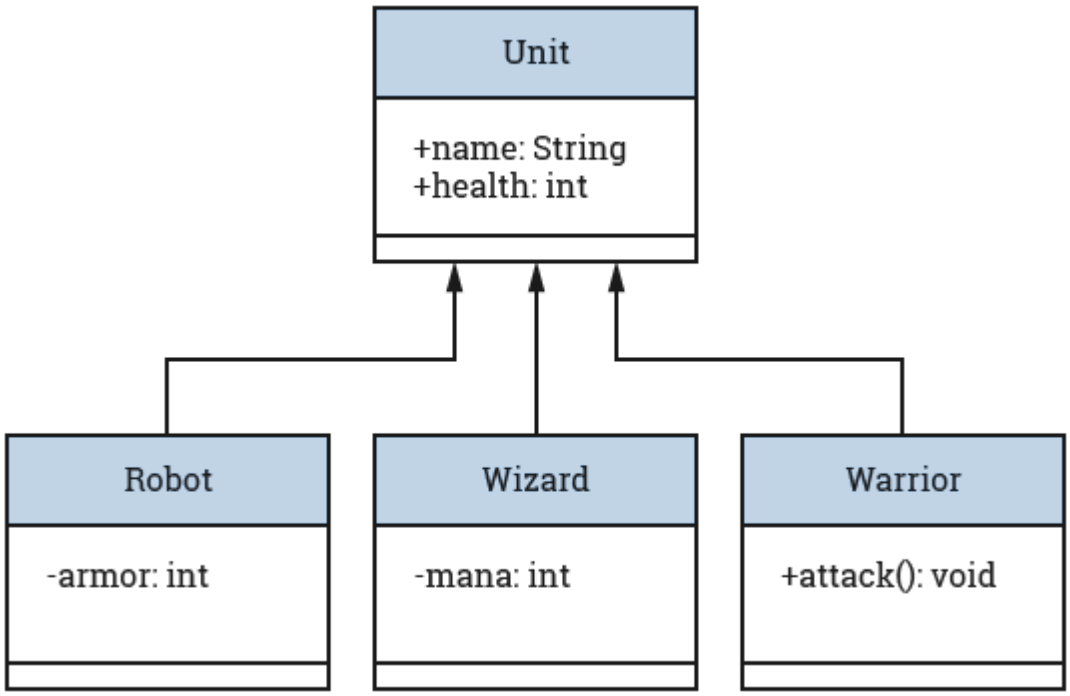


Рис. 2.6.

Классы Robot, Wizard и Warrior являются наследниками класса Unit. Это означает, что у них присутствует все свойства, которые были в Unit, хоть они в них явно не указаны.

Пример 2.11

Реализуйте набор классов с диаграммой UML, который был рассмотрен в разделе 2.6.2. Код примера находится здесь.

```
class Unit {
        public String name;
        public int health;
}

class Robot extends Unit {
        private int armor;
}

class Wizard extends Unit {
        private int mana;
}

class Warrior extends Unit {
        public void attack() {}
}
```

2.6.3. Защищенные члены класса

До этого мы изучили, что ключевое слово private означает, что свойства и методы класса будут недоступны вне класса (являются приватными), а public означает, что свойства и методы класса доступны извне (являются публичными). Использование слова private бывает полезно, когда разработчик класса хочет скрыть некоторую реализацию класса от его пользователей.

Но пользователями класса могут быть как другие не связанные напрямую классы, так и классы-наследники. И слово private точно также скрывает методы и свойства от них. Поскольку класс-наследник орудует, можно сказать, во внутренностях суперкласса, то логично было бы дать ему больше прав чем обычным пользователям класса.

Ключевое слово protected делает свойства и методы доступными как классам-наследникам, так и всем другим классам внутри пакета. То есть в рамках пакета public и protected идентичны.
Рассмотрим следующий пример:

```
public class MyProgram {

        public static class A {

                public int a = 1;

                private int b = 2;

                protected int c = 3;

        }

        public static class B extends A {

                public B() {

                    a = 11; // допустимо, так как a - public

                    b = 22; // недопустимо, так как b - private

                    c = 33; // допустимо, так как c - protected

                }

        }

        public static void main(String args[]) {

                B bObj = new B();

                bObj.a = 111; // допустимо, так как a - public

                bObj.b = 222; // недопустимо, так как b - private

                bObj.c = 333; // допустимо, так как c - protected и классы находятся в одно
м пакете

        }

}
```

2.6.4. Ключевое слово super

Ключевое слово super представляет из себя ссылку на базовый класс, которую можно использовать в классах-наследниках. В основном super используют для вызова методов родительского класса. К примеру, можно использовать его для вызова конструктора базового класса:

```
class Robot extends Unit{
        int armor;

        public Robot(String name, int health, int armor) {
                super(name, health);
                this.armor = armor;
    }
}
```

Обычно метод super() всегда является первым оператором внутри конструктора класса наследника. В примере мы вызываем конструктор класса Unit, который инициализирует переменные name и health. Остается определить добавленное поле armor.

Второй вариант использования ключевого слова super — обращение к полям и методам родительского класса. В общем виде оператор имеет следующий вид:

```
super.членКласса;
```

Рассмотрим пример:

```
class Robot extends Unit{
    int armor;

    public Robot(String name, int health, int armor) {
            super(name, health);
            this.armor = armor;
    }

    public void printInfo(){
            super.printInfo();
            System.out.println("Armor:" + armor);
    }
}
```

В приведенном примере ключевое слово super используется для вызова метода printInfo() родительского класса Unit. При этом сначала выполнится весь код, написанный в методе класса Unit, затем выполнится оператор класса-наследника System.out.println(«Armor:" + armor).

Таким образом, уже знакомый нам код Android-приложения super.onCreate(savedInstanceState) обращается к методу onCreate() из класса-родителя.

2.6.5. Понятие полиморфизма

Полиморфизм, наряду с инкапсуляцией и наследованием, является одним из важнейших инструментов ООП.

Полиморфизм — это способность предоставлять один и тот же интерфейс для различных типов данных.

Полиморфизм позволяет работать с несколькими типами так, как будто это один и тот же тип. В то же время поведение каждого типа будет уникальным в зависимости от его реализации. Рассмотрим это понятие подробнее на примере.

В классе Robot могут быть определены свои методы и поля. Полиморфизм предоставляет возможность изменения функциональности, которая унаследована от базового класса.

Полиморфизм позволяет абстрагироваться от типа конкретного объекта. Суть абстрагирования заключается в том, чтобы определять метод в том месте, где есть наиболее полная информация о том, как он должен работать. Полиморфизм реализуется путем переопределения (не путать с перегрузкой!) методов.

Например, переопределим метод printInfo() класса Unit в классе Robot:

```
class Robot extends Unit{
        int armor;

        public Robot(String name, int health, int armor) {
                super(name, health);
                this.armor = armor;
        }

        public void print(){
                super.printInfo();
                System.out.println("Armor:" + armor);
        }
}
```

В классе Robot определено дополнительное поле для хранения индикатора брони. Кроме того, оно определяется в конструкторе.

Обратите внимание на то, что поле name в базовом классе Unit объявлено с модификатором private. Поэтому мы не можем обратиться к нему напрямую из класса Robot. В данном случае можно использовать ключевое слово super. Чтобы установить значение поля name, мы обращаемся к конструктору суперкласса с помощью ключевого слова super. Так же мы можем обратиться ко всем членам базового класса, если они не определены с модификатором private.

В классе Robot переопределяется метод базового класса printInfo(). В нем с помощью ключевого слова super производится обращение к методу printInfo() базового класса. Затем выводится информация, относящаяся только к классу Robot.

Используя обращение к методам базового класса, можно было бы переопределить метод printInfo() следующим образом:

```
public void print(){
        super.printInfo();
        System.out.println("Armor:" + armor);
}
```

Возможности среды Eclipse позволяют в диалоговом режиме добавить переопределяемые методы в класс. Для этого необходимо, находясь в теле класса, выполнить команду Override/Implement Methods… из меню Source. В появившемся диалоговом окне отображаются доступные для переопределения в классе методы. Наиболее часто на практике переопределяются методы equals(Object) и toString().

В среде Android Studio аналогичные действия выполняются через меню Code->Override Methods, которое вызывается сочетанием клавиш Ctrl+O.

Если не переопределить метод суперкласса в классе-наследнике, то он будет использовать функционал базового класса.

Пример 2.12

Рассмотрим иерархию классов из предыдущего примера. Проект примера находится здесь. Опишем классы Wizard и Termanator по аналогии с классом Robot из раздела 2.6.4, скомпилируем следующий пример:

```
package ru.samsung.itschool.book;

import java.util.Random;

class Unit{
        private String name;
        private int health;

        public Unit(String name, int health) {
                this.name = name;
                this.health = health;
        }

        public void printInfo(){
                System.out.println("Name:" + name);
                System.out.println("Health:" + health);
        }
}

class Robot extends Unit{
        int armor;

        public Robot(String name, int health, int armor) {
                super(name, health);
                this.armor = armor;
        }

        public void printInfo(){
                super.printInfo();
                System.out.println("Armor:" + armor);
        }
}

class Wizard extends Unit{
        int mana;

        public Wizard(String name, int health, int mana) {
                super(name, health);
                this.mana = mana;
        }

        public void printInfo(){
        super.printInfo();
        System.out.println("Mana:" + mana);
        }
}

class Terminator extends Robot{
        String gun;
        public Terminator(String name, int health, int armor, String gun) {
                super(name, health, armor);
                this.gun = gun;
        }
```

```
        public void printInfo(){
                super.printInfo();
                System.out.println("Gun:" + gun);
        }
}

public class Example{
        public static void main(String [] args){
                Unit [] units = new Unit[5];
                for(int i=0;i<units.length;i++){
                        switch(new Random().nextInt(3)){
                                case 0:units[i]=new Robot("Robot", 100, 100);break;
                                case 1:units[i]=new Wizard("Wizard", 99, 80);break;
                                case 2:units[i]=new Terminator("Terminator", 150, 150, "AK-
47");break;
                        }
                }
                for(Unit u: units) {
                        u.printInfo();
                        System.out.println();
                }
        }
}
```

В итоге на экране появится

```
Name:Wizard Health:99 Mana:80
Name:Terminator Health:150 Armor:150 Gun:AK-47
Name:Wizard Health:99 Mana:80
Name:Wizard Health:99 Mana:80
Name:Robot Health:100 Armor:100
```

Здесь полиморфизм проявляется в том, что во втором цикле мы работаем с переменной u как с типом Unit. Более того, на этапе написания этой программы (Compile time) невозможно даже определить, что за типы данных там буду реально находится во время работы программы (Run time). Однако метод printInfo() в каждом конкретном случае будет вызываться не из класса Unit, а из класса фактического объекта. То есть полиморфизм заключается в том, что с объектом мы обращаемся как с юнитом, но «ведет» себя он, как конкретный экземпляр робота, волшебника или терминатора.

2.6.6. Абстрактные классы

Кроме обычных классов в программе могут содержаться абстрактные классы.
Абстрактными называются классы, которые имеют абстрактные методы. Абстрактными называются те методы, которые не имеют реализации в классе. После круглых скобок следует не блок описания метода, а точка с запятой. Перед именем метода указывается при этом ключевое слово abstract.

Абстрактный метод можно использовать, только переопределив его в производном классе. Для исключения возможности прямого использования абстрактного метода в Java требуется, чтобы класс, имеющий хотя бы один абстрактный метод, был абстрактным. То есть наличие абстрактного метода автоматически делает класс абстрактным. Класс-наследник абстрактного класса обязан переопределить все абстрактные методы базового класса.

Вернемся к примеру про класс Unit. Например, необходимо реализовать метод attack().
На практике данный метод у объекта Unit не всегда уместен, так как непонятно, как некий абстрактный юнит будет атаковать (мечом, магией, лазером). Однако иметь такой метод (кстати часто с целью полиморфизма!) крайне желательно. В таком случае метод объявляют его без реализации. Получается, что в классе он есть, но использовать его в этом классе нельзя. Во избежание путаницы с обычным методом его отмечают ключевым словом abstract. При этом сам класс Unit становится абстрактным. В классе конкретного игрового объекта вы можете переопределить этот метод в соответствии с логикой игры.

Покажем на примере, когда целесообразно использование абстрактных методов. Сделаем метод printInfo() класса Unit абстрактным, а также сгенерируем сеттеры и геттеры для доступа к полю name.

```
abstract class Unit{
        private String name;
        private int health;

        public abstract void printInfo();
}

class Robot extends Unit{
        int armor;

        @Override
        public void printInfo() {
                System.out.println("Name:" + super.getName());
                System.out.println("Health:" + super.getHealth());
                System.out.println("Armor:" + armor);
        }
}
```

Если мы захотим создать экземпляр класса Unit, то система предложит реализовать метод printInfo().

```
Unit unit = new Unit("Unit", 100) {
        @Override
        public void printInfo() {
                // TODO Auto-generated method stub

        }
};
```

2.6.7. Ключевое слово final

Ключевое слово final имеет различные интерпретации в зависимости от того, в каком месте программы оно используется. Но суть этого слова одна — запрет на изменение. Рассмотрим следующий пример:

```
public class MyProgram {
        public final int MAGIC_CONST = 100;
}
```

Здесь значение MAGIC_CONST нельзя изменить. Таким образом, если написать слово final возле примитивной переменной, то она является константой.

Рассмотрим ситуацию использования ссылочного типа:

```
public class Example{
        public final String s;

        public Example() {
                s = new String();
        }
}
```

Данный код не будет вызывать ошибки, но только до тех пор, пока не присвоить новое значение переменной s. То есть слово final для ссылочных типов запрещает изменение указателя, а не значения по указателю. Также стоит отметить, что присваивание допустимо в любом месте программы, но только один раз.

Другие способы использования ключевого слова final касаются наследования. И первый из них — это слово final возле метода.

```
public class Example{
        public final int helloWorld() {
                System.out.println(“Hello world!!!”);
        }
}
```

Слово final возле метода запрещает переопределение этого метода в классах-наследниках. То есть, если создать класс-наследник от Example и написать в нем следующий код, то произойдет ошибка:

```
public class ExtendedExample extends Example{
        public int helloWorld() {
                System.out.println("Extended hello world!!!");
        }
}
```

Следующим примером, когда можно написать слово final — возле класса.

```
public final class Example{
}
```

Финальный класс — это класс, который не может быть суперклассом, то есть запрещается писать для него наследников.

2.6.8. Понятие интерфейса

В разработке программного обеспечения понятие интерфейса весьма широкое и зачастую имеет достаточно разный смысл в зависимости от контекста. Например, распространенная фраза «интерфейс программы» подразумевает «графический интерфейс пользователя программы», а фраза «интерфейс библиотеки» подразумевает «прикладной программный интерфейс библиотеки». Это абсолютно разные интерфейсы, но сложностей в восприятии обычно не возникает.

В данной главе мы рассмотрим интерфейсы с точки зрения объектно-ориентированного программирования. Начнем с общего понятия интерфейса.

Интерфейс — это совокупность возможностей взаимодействия двух систем.

Таким образом, интерфейс является способом коммуникации между двумя субъектами. Интерфейс обычно подразумевает некоторый протокол общения, то есть формальный подход к его описанию. Даже общение через телефон, как правило, начинается со слова «алло» (или его альтернативы), что предусмотрено протоколом общения по телефону. Отсюда попытаемся дать определение интерфейса в объектно-ориентированном программировании.

Интерфейс — это конструкция языка программирования, в рамках которой описываются только абстрактные публичные (abstract public) методы и статические константные поля (final static). Другими словами, интерфейс — это полностью абстрактный класс. И на него распространяются те же требования, что и на абстрактные классы.

То есть интерфейс — это описание того, какие методы должны присутствовать в классах, которые намереваются реализовывать этот интерфейс. Если класс C реализует интерфейс I, то другие классы смогут пользоваться некоторыми возможностями класса C через интерфейс I. Например, в Java существует интерфейс Comparable, который заставляет все классы, которые реализуют этот интерфейс, иметь внутри себя метод compareTo. Благодаря этому обязательству можно смело пользоваться методом compareTo, который сравнивает экземпляры этого класса.

Упрощенно можно считать, что интерфейсы — это классы, в которых методы не имеют реализации.

Так как все поля интерфейса константные и статические, а методы публичные, соответствующие модификаторы перед полями и методами можно не указывать.

Классы могут реализовывать (implements) интерфейсы (получать от интерфейса список методов и описывать реализацию каждого из них). В отличие от наследования классов возможна множественная реализация интерфейсов.

Перед описанием интерфейса необходимо написать ключевое слово interface. Если класс реализует интерфейс, то после следом за его именем указывается ключевое слово implements.

Рассмотрим пример интерфейса, который обязывает все классы, которые его реализуют, возвращать и устанавливать некоторое значение типа int:

```
public interface IntValuable {
        void setValue(int value);
        int getValue();
}
```

Этот интерфейс обязывает классы реализовывать два метода: setValue и getValue. Попробуем написать такой класс:

```
public class Example implements IntValuable {
        private int value = 0;

        public void setValue(int value) {
                this.value = value;
        }

        public void getValue() {
                return value;
        }
}
```

Обратите внимание, что если при наследовании мы использовали слово extends (расширяет), то в данном случае мы используем слово implements (реализует). То есть фраза «class Example implements IntValuable» означает «класс Example реализует IntValuable». А раз класс Example реализует интерфейс IntValuable, то в этом классе необходимо написать методы этого интерфейса: setValue и getValue. Классы, использующие один и тот же интерфейс, могут реализовывать его методы по-разному.

Реализуем интерфейс Playable (способный к игре), включив в него методы move — двигаться, attack — атаковать, defense — оборонятся. Описание интерфейса Playable и его реализация классами Robot и Wizard имеют вид:

```
interface Playable{
    void move();
    void attack();
    void defense();
}


class Robot implements Playable{
    public void move() {
        System.out.println("ride");
    }

    public void attack() {
        System.out.println("shoot");
    }

    public void defense() {
        System.out.println("block");
    }

}

class Wizard implements Playable{
    public void move() {
        System.out.println("walk");
    }

    public void attack() {
        System.out.println("magic shoot");
    }

    public void defense() {
        System.out.println("magic block");
    }
}
```

Часто всю открытую часть класса (общедоступные методы) предопределяют в интерфейсе. Тогда, взглянув на интерфейс, можно понять, какие методы могут использоваться для взаимодействия с объектами данного класса. То есть интерфейсы полностью соответствуют принципам инкапсуляции и полиморфизма. В разных классах метод некоторого интерфейса может быть реализован по-разному, хотя с одним и тем же именем.

Такая программа легко расширяема, поскольку можно добавлять новые возможности через наследование новых типов данных от общего базового класса. Методы, манипулирующие интерфейсом базового класса, не нуждаются в изменении в новых классах.

Класс может реализовывать несколько интерфейсов. Если в классе необходимо реализовать несколько интерфейсов, то они перечисляются через запятую после слова implements:

```
interface IntValuable {
        // методы интерфейса
}

interface DoubleValuable {
        // методы интерфейса
}

class Example implements IntValuable, DoubleValuable {
        // реализация класса
}
```

Если класс не реализует все методы интерфейса, то его нужно отмечать как абстрактный.

Если у метода есть входной параметр «тип интерфейса», например, Comparable, то мы можем передать в него объект любого типа, главное, чтобы объект реализовывал этот интерфейс.

2.6.9. Иерархия наследования и преобразование типов

В разделе 1.2.6 говорилось о преобразованиях простых типов. С объектными типами все происходит немного по-другому. Предположим, у нас есть следующая иерархия классов

```
abstract class Unit{
    private String name;
    private int health;

    public Unit(String name, int health) {
        this.name = name;
        this.health = health;
    }

    public abstract void printInfo();
}



class Robot extends Unit{
        int armor;

        public Robot(String name, int health, int armor) {
                super(name, health);
                this.armor = armor;
        }

        @Override
        public void printInfo() {
                System.out.println("Name:" + super.getName());
                System.out.println("Health:" + super.getHealth());
        System.out.println("Armor:" + armor);
        }
}

class Wizard extends Unit{
        private int mana;

        public Wizard(String name, int health, int mana) {
                super(name, health);
                this.mana = mana;
        }

        @Override
        public void printInfo() {
                System.out.println("Name:" + super.getName());
                System.out.println("Health:" + super.getHealth());
                System.out.println("Mana:" + mana);
        }
}
```

В представленной иерархии можно проследить следующую цепочку наследования: Object (так как все классы в Java неявно являются наследниками класса Object) -> Unit -> Robot/Wizard.

Используем классы в программе, произведя преобразования типов:

```
Object object = new Robot("R2D2", 100, 100);
Unit unit = new Wizard("Gendalf", 100, 1000);

//у Object нет метода printInfo, поэтому необходимо привести к Robot
((Robot)object).printInfo();

//у Unit есть метод printInfo
unit.printInfo();

//у Unit нет метода getMana, поэтому необходимо привести к Wizard
int num = ((Wizard)unit).getMana();
```

Сначала создается две переменные типа Object и Unit. В них помещаются ссылки на объекты типа Robot и Wizard соответственно. В данном примере происходит неявное преобразование типов. В Java допустимо неявное восходящее преобразование, то есть преобразование к типам, которые находятся выше по уровню иерархии. Однако при использовании этих переменных было произведено явное преобразование. Поскольку переменная object хранит ссылку на объект типа Robot, мы можем преобразовать к этому типу: ((Robot)object).printInfo(). То же самое относится и к переменной unit.

Добавим еще один класс Terminator, который сделаем наследником класса Robot:

```
class Terminator extends Robot{
        private String gun;
        public Terminator(String name, int health, int armor, String gun) {
                super(name, health, armor);
                this.gun = gun;
        }
}
```

Так как объект класса Terminator в то же время является роботом (то есть объектом Robot), мы можем написать следующий код:

```
Robot robot = new Terminator("T-100", 100, 100, "AK-47");
robot.printInfo();
```

Это еще один пример восходящего преобразования Terminator к Robot. Наоборот, если мы попробуем хранить ссылку на Robot в объекте класса Terminator, то компилятор выдаст ошибку:

```
Terminator robot = new Robot ("T-100", 100);
robot.printInfo();
```

В данном случае происходит попытка неявного преобразования объекта Robot к типу Terminator. Если Terminator является объектом типа Robot, то объект Robot не является объектом типа Terminator.

Перед тем, как произвести преобразование типов, мы можем сделать проверку с помощью оператора instanceof:

```
Robot robot = new Robot("T-100", 100);
if(robot instanceof Terminator){
        ((Terminator) robot).printInfo();
}
else{
        System.out.println("Недопустимое преобразование типов");
}
```

Логическое выражение robot instanceof Terminator проверяет, является ли переменная robot объектом типа Terminator. В данном случае явно не является, и такое выражение вернет false.

2.7. Намерения

Сайт: IT Академия SAMSUNG
Курс: MDev @ IT Академия Samsung
Книга: 2.7. Намерения
Напечатано:: Егор Беляев
Дата: Суббота, 18 Апрель 2020, 19:17

Оглавление

2.7.1. Контекст в Android

Ранее в курсе понятие контекста уже встречалось. Например, когда необходимо вывести всплывающее сообщение:

```
Toast.makeText(this,”qwert”,Toast.LENGTH_SHORT).show();
```

Класс android.context. Context представляет из себя интерфейс для доступа к глобальной информации об окружении приложения. Это абстрактный класс, реализация которого обеспечивается системой Android.

Context позволяет получить доступ к ресурсам и классам данного приложения, а также необходим для вызова операций на уровне приложения, таких как запуск активностей (Activity), отправка широковещательных сообщений, получение намерений (Intent) и прочее.

Данный класс также является базовым для классов Activity, Application и Service.

Получить доступ контексту можно с помощью методов getApplicationContext, getContext, getBaseContext, а также просто с помощью свойства this (изнутри активности или сервиса).

Именно Context является родительским классом для Activity, Service, что является отличным примером применения наследования в Android-разработке.

Объект Context часто применяется в программировании под Android.

1. Для динамического пользовательского интерфейса, создания новых объектов, адаптеров и т. д.

   ```
   TextView myTextView = new TextView(this);
   ListAdapter adapter = new SimpleCursorAdapter(getApplicationContext(),...);
   ```

2. Для доступа к стандартным глобальным ресурсам.

   ```
   //Доступ из класса Activity -- наследника Context
   getSystemService(LAYOUT_INFLATER_SERVICE);
   //Доступ с использованием Контекста Приложения
   SharedPreferences prefs = getApplicationContext().
   getSharedPreferences("PREFS", MODE_PRIVATE);
   ```

   В Android существует несколько видов контекста, отличающихся длительностью жизненного цикла, а также областью применимости. Неоправданное использование контекста с более длительным жизненным циклом может привести к утечке ресурсов в приложении.

2.7.2. Намерения (Intent)

Намерения (Intent) в Android используются в качестве механизма передачи сообщений, который может работать как внутри одного приложения, так и между приложениями. Намерения могут применяться для:

- объявления о желании (необходимости) запуска какой-либо активности или сервиса для выполнения определенных действий;
- извещения о том, что произошло какое-то событие;
- явного запуска указанного сервиса или активности.

Намерения бывают двух видов:

- неявные;
- явные.

Явные намерения

При использовании явного Intent указываются:

- контекст, предоставляющий нужную информацию об окружающей среде приложения;
- класс целевого компонента (будь то сервис, активность), который нужно будет запустить.

Рассмотрим три варианта работы с явными намерениями.

1. Переход из одного Activity в другой Activity.
2. Открытие и передача каких-либо данных в открываемую Activity.
3. Открытие и обратный возврат какого-то результата при закрытии открываемого Activity.

Для перехода из одной активности в другую нужно выполнить метод startActivity, передав ему в качестве входного параметра намерение:

```
i = new Intent(MainActivity.this, MainActivity2.class);
startActivity(i);
```

Где i — переменная Intent — представляет собой намерение. Конструктор Intent() принимает два аргумента:

- первый типа Context — текущий контекст, например, текущая активность;
- класс вызываемой сущности, например, класс активности, которую мы хотим запустить. Стоит обратить внимание на то, как в системе определяется, что за компонент ему передан в качестве цели. В Android путеводителем для системы является файл AndroidManifest. При создании записи Activity в манифест-файле указывается имя класса. Если указать тот же класс в Intent, то система, просмотрев манифест-файл, обнаружит соответствие и покажет соответствующий Activity. Но если такого компонента не окажется в манифест-файле, или тип компонента не будет соответствовать совершаемому действию, система не сможет проделать нужные действия.

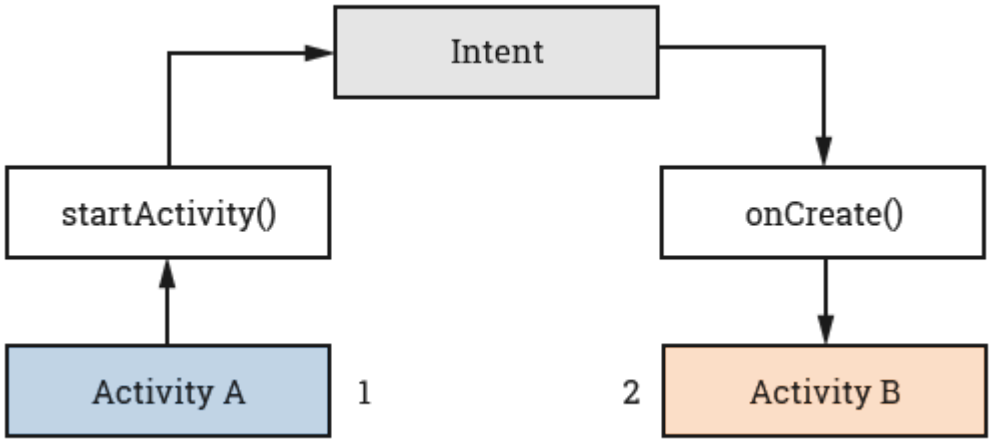


Рис. 2.7.

На рисунке 2.7 изображен процесс передачи явного объекта Intent по системе для запуска другой активности: [1] активность A создает объект Intent с описанием действия и передает его методу startActivity(). [2] Система Android запускает соответствующую классу, указанному в интенте, активность B, вызвав ее метод onCreate() и передав ей объект Intent. Активность A при этом перешла в остановленное состояние (paused), но не уничтожена, то есть остается в памяти, а верней, в стеке активностей. Активность B находится в состоянии Activity Running. Если же активность B завершит работу, то активность, сохраненная на верху стека активностей, то есть активность A, будет переведена в рабочее состояние.

Как было показано выше, намерение представляет собой объект обмена сообщениями между операциями (активностями, сервисами) Android. В Intent можно вложить или извлечь данные при помощи методов putExtra / getExtra. Таким образом, Intent можно представить посылкой, которую Android доставляет от абонента A к абоненту B. Метод putextra получает в качестве параметров ключ, по которому значение можно извлечь, и само значение. Ключ имеет строковый тип, а значение может иметь либо примитивный тип, либо строковый тип, либо являться массивом, либо объектный тип, реализующий интерфейс Parcelable или Serializable.

```
i = new Intent(MainActivity.this, ToInfActivity.class);
String eText = "information to send";
i.putExtra("et", eText);
startActivity(i);
```

В запущенной активности данные извлекаются, здесь ключ для извлечения — строка «et»:

```
String str = getIntent().getStringExtra("et");
```

Для остальных типов по аналогии: get<Тип>Extra. Также через намерения можно передавать и полноценные объекты собственных классов. Однако для такой возможности класс этого объекта должен реализовать интерфейс Serializable или Parcelable.

Последний рассматриваемый аспект взаимодействия активностей при помощи намерения — возврат результатов из вызванной активности. В случае, когда ожидается возврат данных из запускаемой активности, необходимо использовать другой метод вызова активности. Это метод startActivityForResult. Первым аргументом ему передают подготовленный объект намерения, а вторым целочисленное значение requestCode. Это значение используют для идентификации операции, чтобы понять, на какую именно вызванную операцию был получен ответ в вызывающей активности.

```
i = new Intent(MainActivity.this, ComeBackActivity.class);
startActivityForResult(i, 1);
```

После выполнения нужных действий и задач вызванная активность завершает свою работы следующим образом:

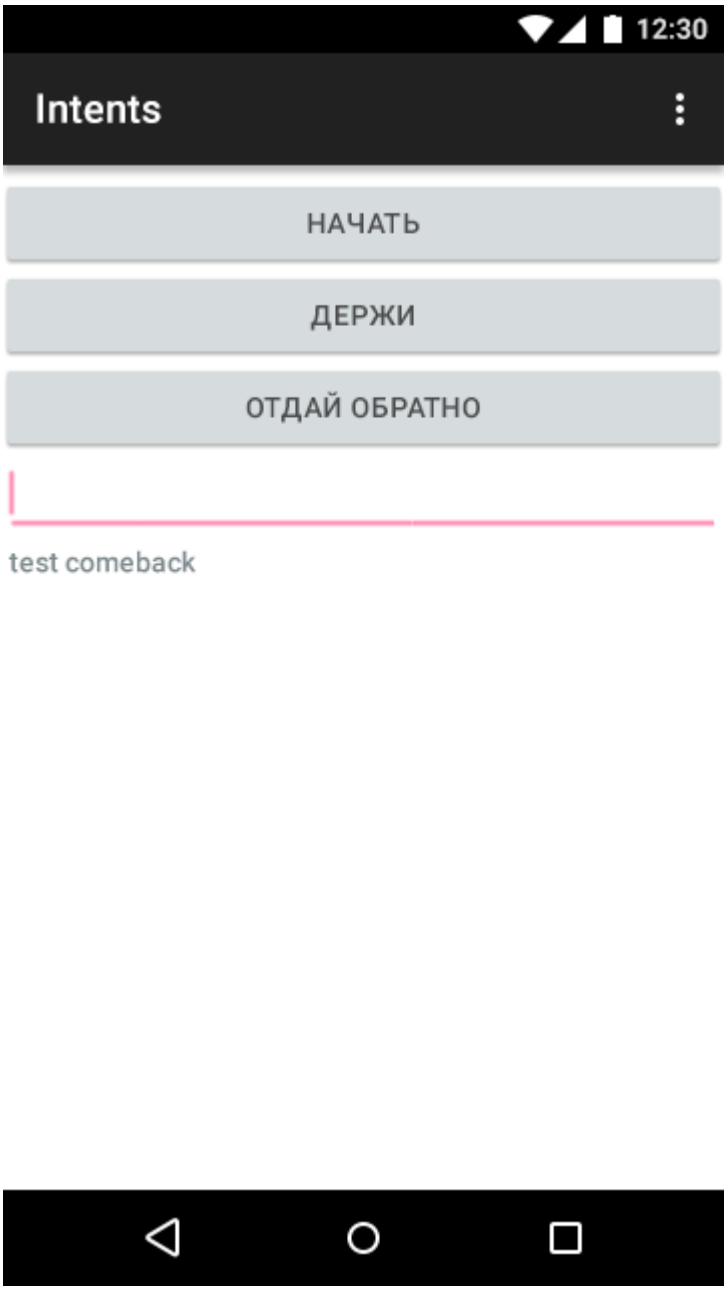
```
Intent i = new Intent();
i.putExtra("et", "inforamation to send comeback to first activity");
setResult(RESULT_OK, i);
finish();
```

В данном случае объект Intenet создается для передачи результата обратно в родительскую Activity. Опять же своего рода посылка, только для передачи данных обратно. Методом setResult в качестве результата передаются код результата и намерение. Код результата нужен для обработки возвращаемых данных в первой активности.

Для того чтобы получить обратную передачу интента, в первом классе необходимо переопределить метод onActivityResult, который будет вызван, когда завершится выполнение вызванной активности (B). В этот же метод будут переданы requestCode (ранее переданный из A в B), resultCode (код возврата) и data, передаваемый обратно Intent:

```
public void onActivityResult(int requestCode, int resultCode, Intent data) {
        switch (resultCode) {
            case RESULT_OK:
                tv.setText(data.getStringExtra("et"));
                break;
        }
}
```

Пример 2.13



Рассмотрим все три варианта использования явных намерений на примере приложения. Создадим приложение Intents. Разметку первой активности activity_main.xml приводим к такому виду:

```
<Button
        android:id="@+id/button"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_alignParentTop="true"
        android:layout_centerHorizontal="true"
        android:text="Начать" />
    <Button
        android:id="@+id/button2"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_alignParentEnd="true"
        android:layout_alignParentRight="true"
        android:layout_below="@+id/button"
        android:text="Держи" />
    <Button
        android:id="@+id/button3"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_alignParentEnd="true"
        android:layout_alignParentRight="true"
        android:layout_below="@+id/button2"
        android:text="Отдай обратно" />
    <EditText
        android:id="@+id/et"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_below="@+id/button3" />
    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_below="@id/et"
        android:id="@+id/tv"/>
```

- Первая кнопка рассчитана просто на открытие Activity.
- Вторая кнопка предназначена для демонстрации вызова новой активности с передачей данных в нее. Поле для ввода нужно для ввода строки, передаваемой в новую Activity.
- Третья кнопка предназначена для демонстрации обратного возврата данных в первую активность из второй. TextView здесь — для отображения полученного из второй Activity текста. Создадим дополнительно три активности: MainActivity2, ToInfActivity, ComeBackActivity. Для каждой добавленной Activity в AndroidManifest.xml прописываем внутрь элемента application:

```
<activity
            android:name=".MainActivity2"
            android:label="MainActivity2" >
</activity>
<activity
            android:name=".ToInfActivity"
            android:label="ToInfActivity" >
</activity>
<activity
            android:name=".ComeBackActivity"
            android:label="ComeBackActivity" >
</activity>
```

Напомним, что без этого Android не сможет найти активности для запуска startActivity, даже если вы корректно описали сами классы активностей.

Объявив и инициализировав все элементы интерфейса, подключим обработчик нажатия к кнопкам. Ниже фрагмент обработки первой кнопки — просто запуск активности 2 с явным намерением:

```
public void onClick(View v) {
               Intent i;
               switch (v.getId()) {
                   case R.id.button:
                       i = new Intent(MainActivity.this, MainActivity2.class);
                       startActivity(i);
                      break;
```

Где i — переменная Intent — представляет собой явное намерение. Конструктор Intent принимает два аргумента — объект типа Context (текущий контекст) и класс запускаемой активности.

Теперь рассмотрим следующий случай — передача параметра в вызываемую активность. В качестве вызываемой возьмем следующую имеющуюся активность ToInfAct. Для передачи информации используется намерение, у которого есть метод putExtra. Этот метод получает в качестве параметров ключ «et», по которому значение можно извлечь и само значение — строку из поля ввода.

```
case R.id.button2:
                       i = new Intent(MainActivity.this, ToInfActivity.class);
                       String eText = et.getText().toString();
                       i.putExtra("et", eText);
                       startActivity(i);
                       break;
```

В запускаемой активности ToInfActivity извлекаем переданную в интенте строку и устанавливаем на экран:

```
public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.toinfactivity);
        tv= (TextView) findViewById(R.id.tv);
        String str = getIntent().getStringExtra("et");
        tv.setText(str);
}
```

Рассмотрим теперь возврат результатов из вызываемой активности. В этом качестве здесь будет ComeBackActivity. Запуск несколько видоизменился:

```
case R.id.button3:
                        i = new Intent(MainActivity.this, ComeBackActivity.class);
                        startActivityForResult(i, REQ_C);
                        break;
```

После выполнения нужных действий и задач ComeBackActivity завершает свою работу с передачей строки обратно в MainActivity. В примере по нажатию кнопки выполнится следующий код:

```
public void onClick(View view) {
               Intent i = new Intent();
                i.putExtra("et",et.getText().toString());
                setResult(RESULT_OK, i);
                finish();
}
```

Однако, когда активность ComeBackActivity завершится и из стека будет поднята MainActivity, нужно каким-то образом получить ответную «посылку». Для этого переопределим метод onActivityResult класса MainActivity. Именно этот метод и будет вызван после завершения ComeBackActivity. В качестве параметров будут переданы коды запроса, ответа и сам интент с передаваемой строкой. После извлечения этой строки установим ее на экран в соответствующую TextVIew:

```
public void onActivityResult(int requestCode, int resultCode, Intent data) {
        switch (resultCode) {
            case RESULT_OK:
                tv.setText(data.getStringExtra("et"));
                break;
        }
}
```

2.7.3. Неявные намерения (Intent)

Неявные намерения не содержат имени конкретного компонента. Вместо этого они в целом объявляют действие, которое требуется выполнить, что дает возможность компоненту из любого приложения обработать этот запрос. Таким образом, при создании объекта Intenet вместо явного указания класса компоненты заполняются определенными значениями следующие параметры — действие, категория, данные. Комбинация этих значений определяют цель, которую нужно достичь. Например: отправка письма, распознавание текста, просмотр картинки, телефонный звонок и т. д. В свою очередь для компонентов (например, Activity) приложений в манифесте приложения описывается ***Intent Filter*** — это набор подобных же параметров: action, data, category в соответствии с целью каждой конкретной активности. И если параметры нашего Intent совпадают с условиями какого-либо фильтра, то Activity вызывается. Следует отметить, что поиск идет по всем Activity всех приложений в системе. Если находится несколько соответствующих условию активностей, то система предоставляет вам выбор, какой именно программой вы хотите воспользоваться. Схематично вызов активности можно изобразить так (см. рис. 2.8).

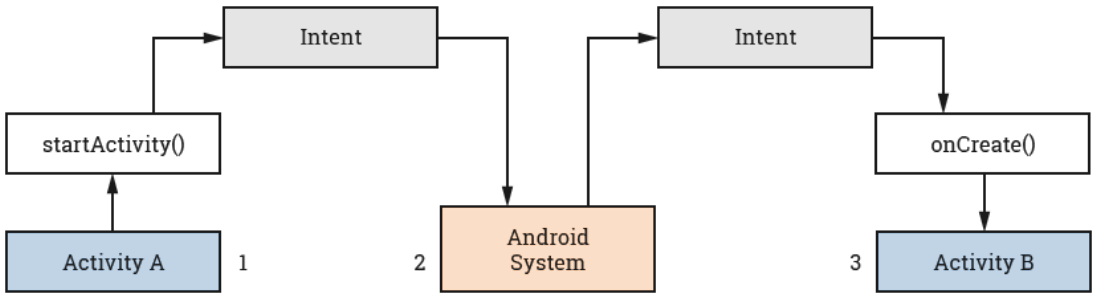


Рис. 2.8.

1. Компонента (активность) A создает объект `Intent` с описанием нужного действия и вызывает метод `startActivity()` с этим интентом.
2. Система Android проверяет во всех приложениях фильтры `Intent`, которые соответствуют данному интенту.
3. Когда приложение с подходящим фильтром найдено, система запускает соответствующую компоненту B, передав ей объект `Intent`.

Принцип отбора системой нужных компонент можно проиллюстрировать следующим образом (см. рис. 2.9):

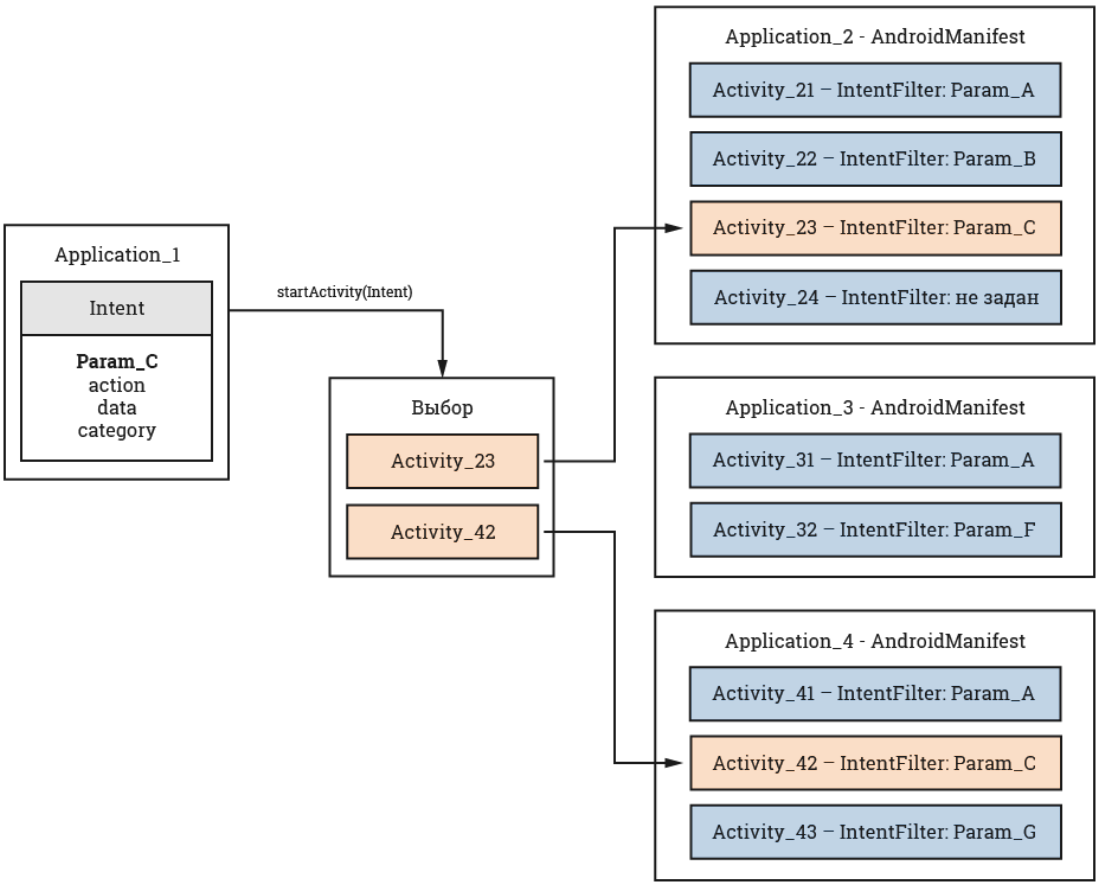


Рис. 2.9.

В Application_1 создается Intent, заполняются параметры action, data, category, назовем этот набор параметров Param_C. С помощью startActivity этот Intent отправляется системе для подбора подходящей Activity, которая сможет выполнить то, что требуется (то есть то, что определено с помощью Param_C). В системе есть различные приложения, и в каждом из них один или более Activity. Для некоторых Activity определен Intent Filter (в примере наборы Param_A, Param_B и т. д.), для некоторых нет. Android сверяет набор параметров Intent и наборы параметров Intent Filter для каждой Activity. Если наборы совпадают (Param_C для обоих), то Activity считается подходящей.

Если в итоге нашлась только одна Activity, она и будет вызвана. Если же нашлось несколько подходящих активностей, то пользователю выводится их список для выбора, какое из приложений ему использовать.

Например, если в системе установлено несколько приложений, осуществляющих обмен сообщениями, а ваше приложение затребует отправку сообщения, то система выведет вам список Activity, которые умеют отправлять сообщения (смс, почтовый клиент, Skype и т. д.), и попросит выбрать, какое из них использовать. Те активности, которые предназначены для другого (например, показывать картинки, воспроизводить музыку и т. п.) будут проигнорированы.

Если для какой-либо активности в манифесте не задан Intent Filter (например, Activity_24), то Intent с параметрами ему не будет сопоставляться, то есть эта активность не будет отобрана.

Пример 2.14

Рассмотрим пример работы с неявными намерениями. Создадим приложение ImplicitIntent для распознавания речи, которое будет представлять из себя одну активность с одной кнопкой и текстовым полем, куда будет выводиться распознанная речь. По нажатию на кнопку создается неявный Intent, который и запустит активность распознавания. Приведем код layout:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:gravity="center"
    android:orientation="vertical">
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:paddingBottom="5dp"
        android:text="Задать вопрос"
        android:fontFamily="calibri"
        android:textColor="@android:color/white"
        android:textSize="18sp"
        android:textStyle="bold" />
    <ImageButton
        android:id="@+id/question"
        android:layout_width="55dp"
        android:layout_height="55dp"
        android:src="@android:drawable/ic_btn_speak_now"
        android:tint="@android:color/white" />
    <ScrollView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content">
        <TextView
            android:id="@+id/text"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_margin="10dp"
            android:text="(после нажатия произнесите вопрос)"
            android:textAlignment="center"
            android:textColor="@android:color/darker_gray"
            android:textSize="12sp" />
    </ScrollView>
</LinearLayout>
```

Вызов активности с неявным намерением производится следующим образом:

```
                Intent intent = new Intent(
                        RecognizerIntent.ACTION_RECOGNIZE_SPEECH);
                intent.putExtra(RecognizerIntent.EXTRA_LANGUAGE_MODEL,
                        "en-US");
                try {
                    startActivityForResult(intent, RESULT_SPEECH);
                } catch (ActivityNotFoundException a) {
                    Toast.makeText(getApplicationContext(),
                            "текст не распознан",
                            Toast.LENGTH_SHORT).show();
                }
```

Как видим, задается фильтр по action распознавания речи. Через putExtra передаются дополнительные параметры. Следует обратить внимание на то, что если система не найдет нужных активностей, то возникнет исключение ActivityNotFoundException, которое нужно

обработать. Результат распознавания будет возвращен как список строк найденной активностью при помощи вызова функции onActivityResult:

```
protected void onActivityResult(int requestCode, int resultCode,Intent
                                    data) {
        super.onActivityResult(requestCode, resultCode, data);
        switch (requestCode) {
            case RESULT_SPEECH: {
                if (resultCode == RESULT_OK && null != data) {
                    ArrayList<String> text = data.getStringArrayListExtra(RecognizerIntent.
EXTRA_RESULTS);
                    textView.setText(text.get(0));
                }
                break;
            }
        }
}
```

Полный текст класса:

```
public class MainActivity extends AppCompatActivity {

    protected static final int RESULT_SPEECH = 1;
    private ImageButton btnSpeak;
    private TextView textView;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        btnSpeak = (ImageButton) findViewById(R.id.question);
        textView = (TextView) findViewById(R.id.text);
        View.OnClickListener listener= new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                Intent intent = new Intent(
                        RecognizerIntent.ACTION_RECOGNIZE_SPEECH);
                intent.putExtra(RecognizerIntent.EXTRA_LANGUAGE_MODEL,
                        "en-US");
                try {
                    startActivityForResult(intent, RESULT_SPEECH);
                } catch (ActivityNotFoundException a) {
                    Toast.makeText(getApplicationContext(),
                            "текст не распознан",
                            Toast.LENGTH_SHORT).show();
                }

            }
        };
        btnSpeak.setOnClickListener(listener);
    }
    @Override
    protected void onActivityResult(int requestCode, int resultCode,Intent
                                    data) {
        super.onActivityResult(requestCode, resultCode, data);
        switch (requestCode) {
            case RESULT_SPEECH: {
                if (resultCode == RESULT_OK && null != data) {
                    ArrayList<String> text = data.getStringArrayListExtra(RecognizerIntent.
EXTRA_RESULTS);
                    textView.setText(text.get(0));
                }
                break;
            }
        }
    }
}
```

В некоторых случаях необходимо добавить в манифест разрешение:

```
<uses-permission android:name="android.permission.INTERNET" />
```

2.8. Параметризованные типы

Сайт: IT Академия SAMSUNG
Курс: MDev @ IT Академия Samsung
Книга: 2.8. Параметризованные типы
Напечатано:: Егор Беляев
Дата: Суббота, 18 Апрель 2020, 19:17

Оглавление

2.8.1. Понятие обобщенного класса

Ранее в этом модуле мы познакомились с основами объектно-ориентированного программирования. В данном разделе описаны некоторые аспекты еще одного популярного подхода — обобщенного программирования.

Обобщенное программирование (generic programming) — это подход к описанию данных и алгоритмов, который предусматривает их использование с различными типами данных без изменения их описания.

Начиная с Java SE 5.0 появилась возможность использования средств обобщенного программирования. Ниже будут рассматриваться generics (дженерики) — одно из подмножеств таких средств.

Дженерики — это параметризованные типы. С помощью параметризованных типов можно объявлять классы, интерфейсы и методы, при этом тип данных выступает в виде параметра.

Эффект от использования параметризованных типов проявляется в больших проектах особенно. Их использование улучшает читаемость и надежность кода.

К основным свойствам дженериков можно отнести:

- строгую типизацию;
- единую реализацию;
- отсутствие информации о типе.

Класс-дженерик определяется следующим образом:

```
class name<T1, T2, ..., Tn> { /* ... */ }
```

Типы-параметры, ограниченные угловыми скобками (<>), следуют за именем класса.

По соглашению, имена типов параметров — одиночные заглавные буквы. Без этого соглашения было бы трудно отличить переменную параметризованного типа от обычного имени класса или интерфейса.

Наиболее часто используемые имена параметризованных типов:

- **E** — элемент (широко используется в Java Collection Framework);
- **K** — ключ;
- **N** — число;
- **T** — тип;
- **V** — значение;
- **S, U, V** и т. д. — 2-й, 3-й, 4-й типы.

Рассмотрим пример создания параметризованного типа:

```
class Generic<T>{
        T t; // объявляем объект типа T

        //передаем в конструктор параметр типа T
        public Generic(T t) {
                this.t = t;
        }

        //получаем параметр типа T
        public T getT() {
                return t;
        }

        //метод вывода параметризованного типа на экран
        public void print(){
                System.out.println("Value T: " + getT());
                System.out.println("Type T: " + t.getClass().getName());
        }
}
```

Если скомпилировать следующий код,

```
Generic<Integer> integerGeneric = new Generic<Integer>(new Integer(5));
integerGeneric.print();

System.out.println();

Generic<String> stringGeneric = new Generic<String>("Hello");
stringGeneric.print();
```

получим следующий результат:

```
Value T: 5 Type T: java.lang.Integer
Value T: Hello Type T: java.lang.String
```

Рассмотрим код подробнее. В программе класс объявлен следующим образом:

```
class Generic<T>{}
```

В угловых скобках используется T — имя параметра типа. В некотором роде имя T используется в качестве «наполнителя» для класса. Во время создания реальных объектов вместо T будет подставлено имя реального типа, переданного классу Generic. Угловые скобки указывают, что параметр может быть обобщен. Далее тип T используется для объявления объекта по имени t:

```
T t; // объявляем объект типа T
```

Вместо T подставится реальный тип, который будет указан при создании объекта класса Generic. Объект t будет объектом типа, переданного в параметре типа T. Например, если в параметре T передать тип Integer, то экземпляр t будет иметь тип Integer.

Рассмотрим конструктор класса.

```
public Generic(T t) {
        this.t = t;
}
```

Здесь также параметр t имеет тип T, значит реальный тип параметра t определяется тем типом, который будет передан при создании объекта класса Generic.

Параметр T может быть использован и для указания типа возвращаемого значения метода:

```
public T getT() {
        return t;
}
```

Далее в программе можно создать версию класса Generic для целых чисел:

```
Generic<Integer> integerGeneric = new Generic<Integer>(new Integer(5));
```

Или строк:

```
Generic<String> stringGeneric = new Generic<String>("Hello");
```

В обоих случаях мы создаем объект класса Generic, в котором все ссылки на тип T становятся ссылками на тип, переданный в угловых скобках. При этом переданный тип должен быть объектным. Невозможно использовать в качестве параметра примитивные типы, например int или double:

```
Generic<int> integerGeneric = new Generic<int>(new Integer(5)); //ошибка
```

Хотя объекты integerGeneric и stringGeneric имеют тип Generic<T>, их нельзя сравнивать, так как они являются ссылками на разные типы.

```
integerGeneric = stringGeneric; // ошибка
```

При определении шаблона можно использовать ключевое слово extends. Например, в следующем классе в качестве параметра могут выступать только наследники класса Number (Integer, Double и т. д.).

```
class Generic<T extends Number>{}
```

Использование дженериков гарантирует безопасность типов во всех операциях, где они участвуют. Это мощнейший инструмент, который широко используется программистами Java.

2.8.2. Обобщенный класс с несколькими параметрами

По аналогии с аргументами метода в параметризованных классах возможно указывать более одного параметра. При этом параметры типов записываются в угловых скобках через запятую. Ниже приведен пример обобщенного класса Pair<K, V> с двумя параметрами.

```
class Pair<K, V>{
        K k;
        V v;

        //Передаем в конструктор ссылки на K и V
        public Pair(K k, V v) {
                super();
                this.k = k;
                this.v = v;
        }

        public void print(){
                System.out.println("Value K: " + getK());
                System.out.println("Type K: " + k.getClass().getName());
                System.out.println("Value V: " + getV());
                System.out.println("Type V: " + v.getClass().getName());
        }

        public K getK() {
                return k;
        }

        public V getV() {
                return v;
        }
}
```

Если скомпилировать следующий код,

```
Pair<String, Integer> pair = new Pair <String, Integer> ("Hello", new Integer(5));
pair.print();
```

то на экране появится:

```
Value K: Hello Type K: java.lang.String Value V: 5 Type V: java.lang.Integer
```

Отдельно отметим, что начиная с JDK 7 появилась возможность сократить код. Следующие две конструкции идентичны:

```
Pair<String, Integer> pair = new Pair<String, Integer>("Hello", new Integer(5));
Pair<String, Integer> pair = new Pair<>("Hello", new Integer(5)); //новый способ
```

2.8.3. Универсальные методы (generic methods)

По аналогии с универсальными классами (дженерик-классами) в Java существует возможность создания универсальных методов (дженерик-методов). При этом, такие методы принимают обобщенные типы параметров. Не стоит путать дженерик-методы и методы в обобщенных классах. Универсальные методы очень удобны в тех случаях, когда одни и те же действия должны применяться к различным типам.

Например, перед нами стоит задача запрограммировать метод, который выводит на экран элементы массива. Ранее, чтобы написать такой метод, мы должны были знать тип массива, который будем выводить на экран. Усложним задачу. Пусть нам нужно написать метод, который выводит на экран массивы типа Integer, Double, Character и Boolean. Так как с дженерик-методами мы еще не знакомы, мы бы воспользовались механизмом перегрузки и написали четыре функции для каждого типа в отдельности:

```
public void printArr(Integer [] iArr){
        for(int i = 0; i < iArr.length; i++){
                System.out.print(iArr[i] + " ");
        }
        System.out.println();
}

public void printArr(Double [] dArr){
        for(int i = 0; i < dArr.length; i++){
                System.out.print(dArr[i] + " ");
        }
        System.out.println();
}

public void printArr(Boolean [] bArr){
        for(int i = 0; i < bArr.length; i++){
                System.out.print(bArr[i] + " ");
        }
        System.out.println();
}

public void printArr(Character [] cArr){
        for(int i = 0; i < cArr.length; i++){
                System.out.print(cArr[i] + " ");
        }
        System.out.println();
}
```

Таким образом, мы имеем четыре перегруженных функции. А если бы количество различных типов массива заранее не было известно? Здесь нам на помощь могут прийти дженерик-методы. Создадим один обобщенный метод, в котором опишем вывод массива для объектных типов.

```
public <T> void printArr(T [] tArr){
        for(int i = 0; i < tArr.length; i++){
                System.out.print(tArr[i] + " ");
        }
        System.out.println();
}
```

Параметр T расположен в угловых скобках после всех модификаторов, затем следуют тип возвращаемого значения, имя метода и его аргументы. Таким образом, исходный код не будет захламлен методами с одинаковой функциональностью. После выполнения следующего кода

```
Integer [] iArr = {1,2,3,4,5};
Double [] dArr = {1.0, 2.0, 3.0, 4.0, 5.0};
String [] sArr = {"a", "b", "c", "d", "e"};
printArr(iArr);
printArr(dArr);
printArr(sArr);
```

На экране появится:

```
1 2 3 4 5 1.0 2.0 3.0 4.0 5.0 a b c d e
```

Ниже приведен еще один пример обобщенного метода, который возвращает средний по порядку элемент в массив.

```
public <T> T middleArr(T [] tArr){
    return tArr[tArr.length/2];
}
```

Если мы попробуем напрямую реализовать, например, простейший алгоритм поиска максимума из двух чисел вот так,

```
public <T> T max(T x, T y) {
        return x > y ? x : y;
}
```

то компилятор выдаст ошибку, так как операции сравнения не определены для объектных типов. Выход из данной ситуации достаточно прост:

```
public <T extends Comparable<T>> T max(T x, T y) {
        return x.compareTo(y)>0 ? x :y;
}
```

Необходимо указать, что тип T расширяет интерфейс Comparable, в котором реализована функция compareTo. Функция возвращает положительное число, если x>y, отрицательное число, если x<y, и ноль, если они равны.

Тема параметризованных типов достаточно обширна и требует отдельного внимания. Следует отметить, что также существуют обобщенные конструкторы и интерфейсы.

Пример 2.15*

Разберем пример, в котором необходимо создать метод, возвращающий максимальное и минимальное значение массива целых чисел в виде объекта Pair<Integer, Integer>.

```
class Pair<K, V>{
	public K k;
	public V v;
	public Pair(K k, V v)
	{
		this.k = k;
		this.v = v;
	}
}

public class Example{
	public static Pair<Integer, Integer> getMinAndMax(Integer[] arr){
		if (arr == null || arr.length == 0){
			return new Pair<Integer, Integer>(null, null);
		}

		Integer max = arr[0];
		Integer min = arr[0];
		for (int i = 1; i < arr.length; i++){
			if (arr[i] > max) max = arr[i];
			if (arr[i] < min) min = arr[i];
		}

		return new Pair<Integer, Integer>(min, max);
	}


	public static void main(String [] args){

		Integer[] arr = new Integer[]{1, 2, 3, 10, -4, -10, 0, 57, 5, 3};

		Pair<Integer, Integer> result =  getMinAndMax(arr);

		System.out.println("Minimum in integer array " + result.k);
		System.out.println("Maximum in integer array " + result.v);
	}
}
```

2.8.4. Методы с параметризованными типами

Параметризованные типы могут выступать в качестве аргументов в методах. Например:

```
<T> void method(Generic<T> generic){}
```

или так:

```
void method(Generic<?> generic){}
```

Использование неизвестного параметра <?> рекомендуется только в тех случаях, когда от него не зависят другие аргументы и возвращаемое значение. На неизвестный параметр можно наложить ограничения:

- **? extends T** — определяет множество классов потомков T;
- **? super T** — определяет множество родительских классов T.

Например, если нам нужна функция для вывода чисел, потомков класса Number, то метод может выглядеть так:

```
public void print(Generic<? extends Number> generic){
    generic.print();
}
```