3.1. Объектно-ориентированное проектирование

Сайт: IT Академия SAMSUNG
Курс: MDev @ IT Академия Samsung
Книга: 3.1. Объектно-ориентированное проектирование
Напечатано:: Егор Беляев
Дата: Суббота, 18 Апрель 2020, 19:23

Оглавление

3.1.1. Пример первый «Дробь»

Класс в Java — это абстрактный тип данных и состоит из полей (переменных) и методов (функций).

В процессе выполнения Java-программы JVM использует определения классов для создания экземпляров классов или, как их чаще называют, объектов. В соответствии с принципом инкапуляции эти объекты содержат в себе состояние (данные) и поведение (функции).

Для того чтобы создать новую программу, необходимо произвести проектирование. Проектирование программ в объектно-ориентированной парадигме отличается от классической (например, нисходящей).

Рассмотрим примеры проектирования на разных задачах.

Задача 1

Дроби. Вычисление и вывод дробей.

Прежде, чем рассмотреть процесс проектирования, рассмотрим пример. Ввести дроби A и B, вычислить и вывести на экран в виде правильной дроби. Решить без применения ООП и с применением ООП.

$$\left(\frac{An}{Ad}+3\right):\left(\frac{Bn}{Bd}-\frac{1}{3}\right)$$

Вариант 1. Решение без применения ООП

При решении задачи будем выполнять шаг за шагом последовательно, как изложено в задаче. То есть будет выполнена следующая последовательность действий.

1. Ввести значения для дробей — An, Ad, Bn, Bd.
2. Привести первую скобку к общему знаменателю и сложение числителей (An+3*Ad)/Ad.
3. Привести вторую скобку к общему знаменателю и вычитание числителей (3Bn–Bd)/3Bd.
4. Поделить результат первой операции на результат второй.
5. Вывести результат в виде числовой дроби и в виде десятичной дроби.

Получившийся код (FractionNonOOP.3.1.1) — класс Fraction1.java:

```
 Scanner sc = new Scanner(System.in);
System.out.println("Введите первую дробь");
int An = sc.nextInt();
int Ad = sc.nextInt();
System.out.println("Введите вторую дробь");
int Bn = sc.nextInt();
int Bd = sc.nextInt();
// считаем первую скобку (приводим к общему знаменателю и складываем числитель)
An = An + 3 * Ad;
// считаем вторую скобку (приводим к общему знаменателю и складываем числитель)
Bn = 3 * Bn - Bd;
Bd = 3 * Bd;
// считаем деление скобок
An = An * Bd;
Ad = Ad * Bn;
System.out.println("Результат:");
// печатаем в десятичном виде
System.out.println(1.0 * An / Ad);
if (An / Ad == 0) {
// печатаем в обычном виде
System.out.println(An);
System.out.println("---");
System.out.println(Ad);
} else {
// печатаем правильную дробь
System.out.println("   " + An % Ad);
System.out.println(An / Ad + "-----");
System.out.println("   " + Ad);
```

Результат работы программы:

```
Введите первую дробь
1
4
Введите вторую дробь
3
5
Результат:
12.1875
      3
12-----
     16
```

Вариант 2. Решение с применением ООП

Теперь решим (FractionOOP.3.1.2) эту же задачу с применением ООП. В отличие от предыдущего примера при разработке в ООП парадигме необходимо сначала спроектировать класс так, чтобы в полях хранился числитель и знаменатель, чтобы были методы для работы с дробью — сложить, вычесть, умножить, делить, вывести на экран, чтобы был конструктор для удобного создания дроби. Кроме того, нужно перегрузить эти методы, чтобы можно было вызывать их для аргументов разных типов. И только после того, как

получен этот класс, можно заниматься решением непосредственно самой задачи. Выше был словесно описан нужный класс, однако чаще используют более наглядный и компактный способ — UML диаграммы (см. рис. 3.1).

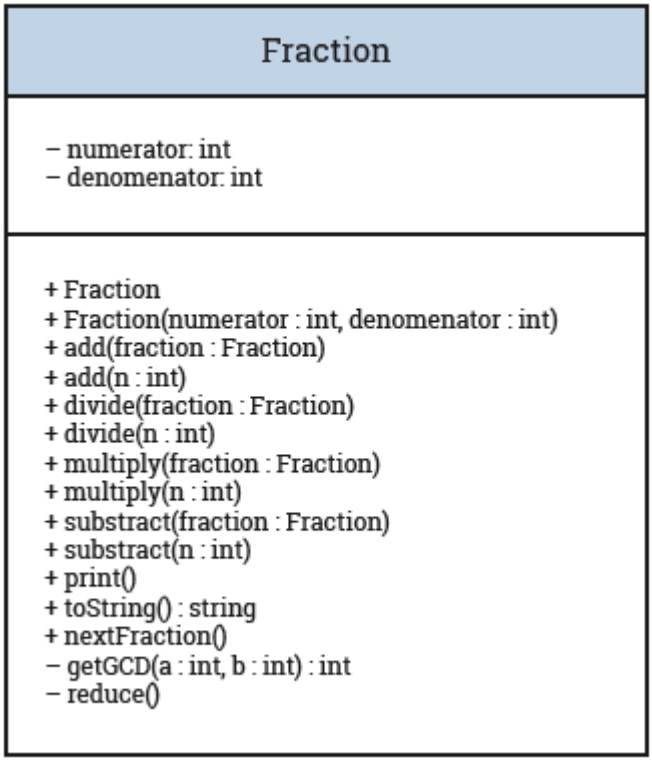


Рис. 3.1.

***UML (Universal Modeling Language*, *универсальный язык моделирования)* —** это способ графического описания, применяемый для разработки программного обеспечения. Несмотря на то что есть множество разновидностей диаграмм, здесь и далее будут представлены только диаграммы классов, которые позволяют дать графическое описание классов и их связей между собой. Обычно создание диаграммы классов знаменует собой окончание процесса анализа и начало процесса проектирования.

Для построения диаграмм UML существует множество инструментов, например, ArgoUML для всех платформ, NClass для Windows, Umbrello для Linux, есть также плагины, встроенные в среды разработки.

Каждый класс в диаграмме классов UML описывается как прямоугольник, состоящий из трех выделенных блоков: наименование класса, поля класса и методы класса. При этом перед именем поля или метода указываются модификаторы видимости (см. табл. 3.1).

Обозначение	**Модификатор Java**
+	public — открытый доступ
-	private — только из методов того же класса
#	protected — только из методов этого же класса и классов, создаваемых на его основе

Табл. 3.1.

Класс Fraction для данной диаграммы:

```
 import java.util.Scanner;
public class Fraction {
    private int numerator;
    private int denominator=1;
    public void add(Fraction fraction) {
          numerator = numerator * fraction.denominator + fraction.numerator * denominator;
          denominator = denominator * fraction.denominator;
          reduce();
    }
    public void add(int n) {
          add(new Fraction(n, 1));
    }
    public void substract(Fraction fraction) {
        numerator = numerator * fraction.denominator - fraction.numerator * denominator;
        denominator = denominator * fraction.denominator;
        reduce();
    }
    public void substract(int n) {
        substract(new Fraction(n, 1));
    }
    public void multiply(Fraction fraction) {
        numerator = numerator * fraction.numerator;
        denominator = denominator * fraction.denominator;
        reduce();
    }
    public void multiply(int n) {
        multiply(new Fraction(n, 1));
    }
    public void divide(Fraction fraction) {
        if (fraction.numerator == 0) {
            System.out.println("На эту дробь делить нельзя!");
            return;
        }
        multiply(new Fraction(fraction.denominator, fraction.numerator));
    }
    public void divide(int n) {
        divide(new Fraction(n, 1));
    }
    public void nextFraction() {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int d = sc.nextInt();
        if (d == 0) {
            System.out.println("Знаменатель не может быть нулевым!");
            return;
        }
        numerator=n;
        denominator=d;
        reduce();
    }
    Fraction(){
    }
    Fraction(int numerator, int denominator) {
```

```
        if (denominator == 0) {
            System.out.println("Знаменатель не может быть нулевым!");
            return;
        }
        this.numerator = numerator;
        this.denominator = denominator;
        reduce();
    }
    public String toString(){
        return (numerator*denominator<0?"-":" ")+ Math.abs(numerator)+"/"+Math.abs(denomina
tor);
    }
    public void print() {
        if(numerator % denominator == 0){
            System.out.println(numerator/denominator);
            return;
        }
        if (numerator / denominator == 0) {
            System.out.println(" " + Math.abs(numerator));
            System.out.println((numerator*denominator<0?"-":" ")+" ---- или "+ 1.0 * numera
tor / denominator);
            System.out.println(" " + Math.abs(denominator));
        } else {
            System.out.println(" " + Math.abs(numerator % denominator));
            System.out.println((numerator*denominator<0?"-":"")+numerator / denominator +
 "---- или"+1.0 * numerator / denominator);
            System.out.println(" " + Math.abs(denominator));
        }
    }
    private int getGCD(int a, int b) { return b==0 ? a : getGCD(b, a%b); }
    private void reduce(){
        int t=getGCD(numerator,denominator);
        numerator/=t;
        denominator/=t;
    }
}
```

Создавая по описанию класс, который содержит нужные поля и методы, программист, по сути, создает новый тип данных, который в дальнейшем можно использовать в любых программах. Поскольку класс дроби готов, то теперь можно, наконец, приступить к вычислению задачи:

```
public class Test {
   public static void main(String[] args) {
       Fraction A = new Fraction();
       Fraction B = new Fraction();
       A.nextFraction();
       B.nextFraction();
       A.add(3);
       B.substract(new Fraction(1,3));
       A.divide(B);
       A.print();
   }
}
```

Запустив на выполнение test.java и введя те же значения, получаем такой же результат, как и в прошлый раз

```
1 4
3 5
     3
12---- или 12.1875
    16
```

Выводы

Задача была решена, используя два разных подхода. Сравним (при одинаковом результате).

1. В первой программе порядка 30 строк. Процесс разработки прост и понятен.
2. Во второй программе порядка 100 строк. Процесс разработки более громоздкий. Результат стал доступен только в конце.

Казалось бы, первый вариант лучше? Однако рассмотрим другой пример.

3.1.2. Более сложные операции с дробями

Задача 2

Ввести пять правильных дробей и вычислить сумму и произведение дробей. Решить без применения ООП и с применением ООП.

$$\sum_{i=0}^{5} \frac{An_i}{Ad_i} \text{ и } \prod_{i=0}^{5} \frac{An_i}{Ad_i}$$

Вариант 1. Решение без применения ООП

Решение без ООП — аналогично решению, приведенному ранее. То есть практически все предыдущее решение необходимо удалить, и решать заново, последовательно, шаг за шагом, четко следуя заданию.

1. Первоначально выполняется ввод дробей в два массива — числители в массив n, а знаменатели в массив d.
2. Для нахождения суммы в программе используется дробь rez1n/rez1d (первоначально как 0/1) и в цикле к ней прибавляются все введенные дроби.
3. Для нахождения произведения дробей в программе используется дробь rez2n/rez2d (первоначально как 1/1) и в цикле на нее умножаются все введенные дроби.
4. В конце осуществляется печать дроби rez1n/rez1d как результат суммы и rez2n/rez2d как результат произведения.

Получившийся код (FractionNonOOP.3.1.1) — класс Fraction2.java:

```
 Scanner sc = new Scanner(System.in);
System.out.println("Введите дроби:");
int n[] = new int[5];
int d[] = new int[5];
for (int i = 0; i < n.length; i++) {
    System.out.println("==============");
    n[i] = sc.nextInt();
    d[i] = sc.nextInt();
}
int rez1n = 0;
int rez1d = 1;
int rez2n = 1;
int rez2d = 1;
for (int i = 0; i < n.length; i++) {
    rez1n = rez1n * d[i] + rez1d * n[i];
    rez1d = rez1d * d[i];
    rez2n = rez2n * n[i];
    rez2d = rez2d * d[i];
}
System.out.println("Результат 1:");
// печатаем в десятичном виде
System.out.println(1.0 * rez1n / rez1d);
if (rez1n / rez1d == 0) {
    // печатаем в обычном виде
    System.out.println(rez1n);
    System.out.println("---");
    System.out.println(rez1d);
} else {
    // печатаем правильную дробь
    System.out.println("    "+rez1n%rez1d);
    System.out.println(rez1n/rez1d+"-----");
    System.out.println("    "+rez1d);
}
System.out.println("Результат 2:");
// печатаем в десятичном виде
System.out.println(1.0*rez2n/rez2d);
if (rez2n / rez2d == 0) {
    // печатаем в обычном виде
    System.out.println(rez2n);
    System.out.println("---");
    System.out.println(rez2d);
} else {
    // печатаем правильную дробь
    System.out.println("   "+rez2n%rez2d);
    System.out.println(rez2n/rez2d+"-----");
    System.out.println("   " + rez2d);
}
```

Результат работы программы будет следующий:

```
 Введите дроби:
==============
1
2
==============
1
2
==============
1
2
==============
1
2
==============
1
2
==============
1
2
Результат 1:
2.5
   16
2-----
   32
Результат 2:
0.03125
1
---
32
```

Вариант 2. Решение с применением ООП

В отличие от решения этой задачи во втором варианте задачи 1 при решении с ООП на данный момент уже имеется класс дроби, и его можно просто использовать, as is, без каких бы то ни было переделок. Переписать придется только лишь ту часть, где использовались дроби, то есть метод main. Поэтому решение выглядит так (см. класс FractionOOP.3.1.2/Test1.java):

```
public class Test1 {
   public static void main(String[] args) {
       Fraction A[] = new Fraction[5];
       for (int i = 0; i < A.length; i++) {
           A[i] = new Fraction();
           A[i].nextFraction();
       }
       Fraction rez1 = new Fraction(0, 1);
       Fraction rez2 = new Fraction(1, 1);
       for (int i = 0; i < A.length; i++) {
           rez1.add(A[i]);
           rez2.multiply(A[i]);
       }
       System.out.println("Сумма");
       rez1.print();
       System.out.println("Произведение");
       rez2.print();
   }
}
```

Результат работы программы будет следующий:

```
 1 2
1 2
1 2
1 2
1 2
Сумма
   1
2---- или 2.5
   2
Произведение
1
---- или 0.03125
32
```

Выводы

Снова сравним решение в двух парадигмах.

1. В варианте без ООП было написано ~45 строк кода, и заново продумана реализация операций (вычисления).
2. В варианте ООП размер программы ~20 строк кода. Причем теперь уже не было нужды задумываться над тем, как именно делаются каждые операции, так как уже с прошлого решения имелась необходимая и достаточная модель дроби.

Вывод из примеров: по мере усложнения задач ООП парадигма показывает гораздо лучшие результаты в проектировании!

При разработке класса были описаны все поля и методы, которые нужны для работы с рассматриваемым объектом. Принцип инкапсуляции дает возможность скрывать часть данных и методов, то есть часть реализации класса от потребителя. Публичные же методы представляют собой интерфейс взаимодействия с этим объектом.

Степень сокрытия и тип воздействия на поля класса определяются в момент проектирования класса. Например, если разработчик не создает методы для доступа к полям класса (сеттеров), а прочие методы не меняют содержимое полей, то полученный класс называется ***immutable***. Ярким примером immutable-класса является стандартный класс String. Если в таком классе нужно выдать потребителю измененный объект (например, увеличенную дробь), то просто создается новый объект с измененным содержимым.

Чаще все же используются классы, которые позволяют менять свои данные. Для этого можно использовать методы сеттеры, либо методы, которые меняют поля по какому-то закону. В рассматриваемом случае это методы, реализующие математические операции с дробью. Наличие специальных методов для доступа (геттеров и/или сеттеров) необязательно и определяется ТЗ, однако их наличие (если это не противоречит ТЗ) скорее приветствуется, так как есть большое количество полезных фреймворков, которые используют аксессоры объектов. Например, спецификация JavaBeans требует наличие аксессоров.

3.1.3. Текстовый квест (мини-проект)

Задача 3

Написать игру — текстовый квест «Корпорация». Цель игрока — поднять социальный статус персонажа.

Описание

В процессе игры игроку рассказывают интерактивную историю. История начинается с первой сцены. Игрок выбирает вариант развития событий из ограниченного набора (2–3 шт.). Выбранная сцена становится текущей, и у нее также есть варианты развития событий. История заканчивается, когда у текущей ситуации нет вариантов развития событий. При выборе варианта у персонажа меняются характеристики: карьерное положение (К), активы (А), репутация Р.

Пример сюжета

К = 1, А = 100 тр, Р = 50%

Первая сцена «Первый клиент».
«Вы устроились в корпорацию менеджером по продажам программного обеспечения.
Вы нашли клиента и продаете ему партию MS Windows. Ему достаточно было взять версию „HOME“ 100 коробок». Далее игроку предлагается на выбор три варианта действий.

1. Вы выпишете ему счет на 120 коробок версии «ULTIMATE» по 50 тр. Клиент может немного и переплатить, подумали вы, но за такое ведение дел вас депремировали на 10 тр, да и в глазах клиента вы показали себя не в лучшем свете. В карьере никаких изменений пока нет. К +0, А –10, Р –10.
2. Вы выпишете ему счет на 100 коробок версии «PRO» по 10 тр. В принципе хорошая сделка, решили вы, и действительно вам выплатили приличный бонус, у вас намечаются положительные перспективы в карьерном росте, правда, в глазах клиента вы ничем не выделились. К +1, А +100, Р +0.
3. Вы выпишете ему счет на 100 коробок версии «HOME» по 5 тр. Размер сделки поменьше, решили вы, но зато вы абсолютно честны с клиентом и продали ему только то, что ему действительно нужно, с минимальными затратами. Такое ведение бизнеса принесет вам скромный бонус, не принесет продвижение по службе, но зато в глазах клиента вы зарекомендовали себя крайне честным продавцом. К +0, А +50, Р +1.

Вариант 1. Решение без применения ООП

Решаем простым последовательным выполнением поставленной задачи. Для этого будем от пользователя получать номер выбранного действия и в соответствии с ним выбирать ветку дальнейшего сюжета с изменением параметров К, А, Р. Получившийся код (QuestNonOOP.3.1.3) — класс Quest.java:

```
 public class Quest {

    public static void main(String[] args) {
        Scanner in  = new Scanner(System.in);
        int K=1,A=100,P=50;
        System.out.println("Вы прошли собеседование и вот-вот станете сотрудниуом Корпораци
и. \n Осталось уладить формальности - подпись под договором (Введите ваше имя):");
        String name;
        name=in.next();
        System.out.println("=====\nКарьера:"+K+"\tАктивы:"+A+"т.р.\tРепутация:"+P+"%\n====
=");
        System.out.println("Только вы начали работать и тут же удача! Вы нашли клиента и пр
одаете ему партию \n ПО МС Виндовс. Ему в принципе достаточно взять 100 коробок версии HOM
E.");
        System.out.println("- (1)у клиента денег много, а у меня нет - вы выпишете ему счет
на 120 коробок \n ULTIMATE по 50тр");
        System.out.println("- (2)чуть дороже сделаем, кто там заметит - вы выпишете ему сче
т на 100 коробок \n PRO по 10тр");
        System.out.println("- (3)как надо так и сделаем - вы выпишете ему счет на 100 короб
ок HOME по 5тр");
        int a1=in.nextInt();
        if(a1==1){
            K+=0;A+=-10;P+=-10;
            System.out.println("=====\nКарьера:"+K+"\tАктивы:"+A+"т.р.\tРепутация:"+P+"%\n=
====");
            // Следующие ситуации для этой ветки сюжета
        } else if(a1==2) {
            K+=1;A+=100;P+=0;
            System.out.println("=====\nКарьера:"+K+"\tАктивы:"+A+"т.р.\tРепутация:"+P+"%\n=
====");
            // Следующие ситуации для этой ветки сюжета
        } else {
            K+=0; A+=50; P+=1;
            System.out.println("=====\nКарьера:"+K+"\tАктивы:"+A+"т.р.\tРепутация:"+P+"%\n=
====");
            // Следующие ситуации для этой ветки сюжета
        }
        System.out.println("Конец");
    }

}
```

Такое решение, даже в этом небольшом примере, выглядит громоздким. А представьте, что будет, если у вас будет по три варианта в каждом подварианте, а в нем в свою очередь еще три и так далее. Эта программа станет совершенно нечитабельной даже для создателя. Еще хуже, если вы решите впоследствии изменить сюжет, удалив или добавив несколько сцен. Тут одно неосторожное изменение может вызвать такой дисбаланс скобок, что найти потерянную скобку будет очень сложно. Конечно, можно придумать какую-то хитрую структуру данных для удобной работы с сюжетом, но зачем изобретать велосипед, если есть ООП подход?

Вариант 2. Решение с применением ООП

«***Объектно-ориентированный анализ и проектирование*** — это метод, логически приводящий нас к объектно-ориентированной декомпозиции. Применяя объектно-ориентированное проектирование, мы создаем гибкие программы, написанные экономными средствами».
(Гради Буч «Объектно-ориентированный анализ и проектирование с примерами приложений на C++». Rational, Санта-Клара, Калифорния)

Прежде чем решать эту задачу в ООП парадигме, давайте разберемся, как это делать правильно. Объектно-ориентированное проектирование, как и обычное проектирование, проходит стадию анализа и синтеза. Анализ ориентирован на поиск отдельных сущностей (или классов, объектов). Синтез же воссоздаст полную модель предметной области задачи.

Анализ — это разбиение предметной области на минимальные неделимые сущности. Выделенные в процессе анализа сущности формализуются как математические модели объектов со своим внутренним состоянием и поведением.

Синтез — это соединение полученных в результате анализа моделей минимальных сущностей в единую математическую модель предметной области, с учетом взаимосвязей объектов в ней. Если же сказать простыми словами, то процесс проектирования можно описать так:

- из текста подробного описания предметной области выбираем все подлежащие — это сущности (классы);
- выбираем все глаголы (отглагольные формы, например, деепричастия) — это поведение сущностей (методы классов);
- выбираем дополнения, определения, обстоятельства — они, скорее всего, будут определять состояние сущностей (поля классов).

Проведя синтаксический разбор и последующий анализ описания нашей задачи, получим следующую таблицу 3.2.

Сущности (подлежащие)	**Свойства (дополнения)**	**Действия (глаголы)**
Игра	История, персонаж	Начать (основная программа)
Персонаж	Имя, Параметры К, А, Р	
История	Ряд сцен, текущая сцена, начальная сцена	Выбрать следующую сцену, проверить на окончание истории
Сцена	Название, описание, возможные варианты развития событий, модификаторы К, А, Р	

Табл. 3.2.

После вычленения из текста задания сущностей, их свойства и действий построим модели отдельных сущностей (это классы) и объединим их в общую модель с учетом их взаимосвязей. При описании каждой сущности следует учесть, являются ли ее поля простыми типами, или в свою очередь другими сущностями (отношение агрегирования или композиции). Это и есть разработка приложения. Сделаем это на UML и получим следующую диаграмму классов (см. рис. 3.2).

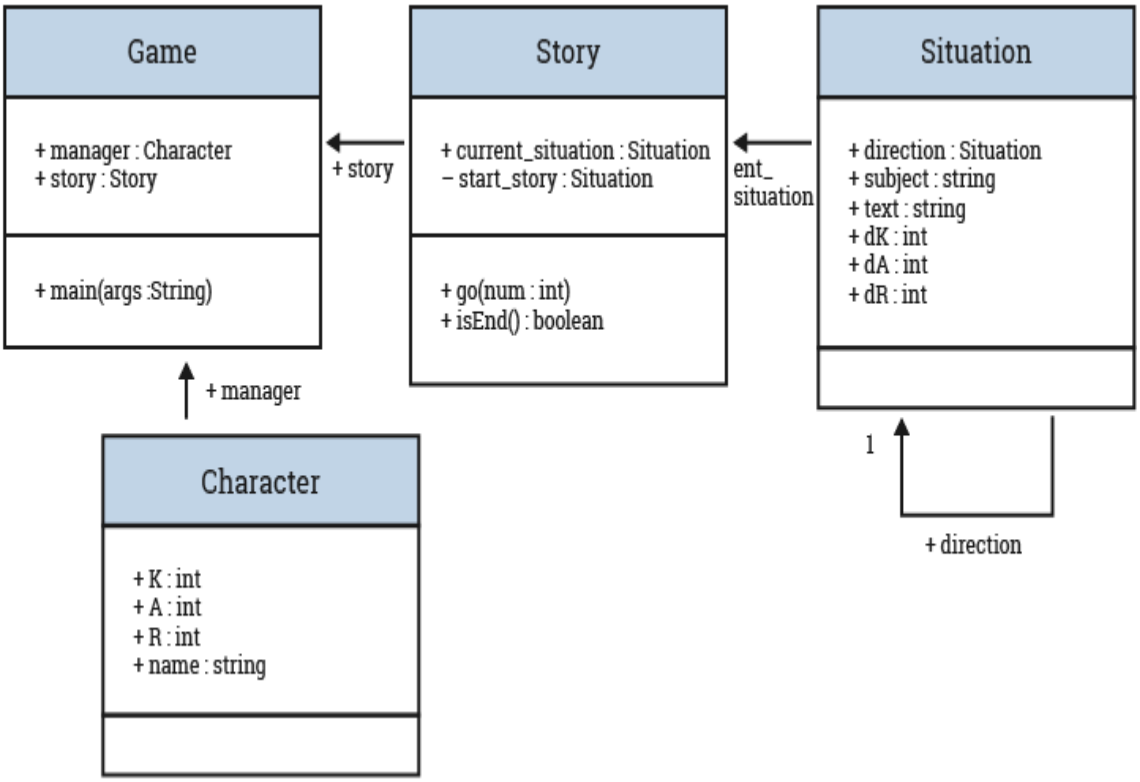


Рис. 3.2.

После построения модели можно прямо из UML-редактора сгенерировать исходный код (только нужно выбрать язык Java). Однако редактор зачастую дает правильный, но избыточный код. Конечно, по диаграмме можно написать классы и вручную, тем более что они небольшие.

Отдельно стоит отметить, что:

- классы Character и Situation почти пусты;
- класс Story в конструкторе создает всю историю как набор ситуаций;
- класс Game имеет точку входа для консольного выражения (он же главный метод main), где создается персонаж и история, происходит взаимодействие с пользователем и переключение сцен.

В случае же если приложение разрабатывалось для Android, то класс Game будет расширять класс Activity, и в методе onCreate (вместо main) будет создан объект персонажа и игры. Переключение сцен (метод go) должен вызываться в методах типа onClick этой активности. Кстати, если поле Direction сделать не обычным массивом, а ассоциативным, то написание самой истории (последовательности сцен) сильно упростится, так как шаги истории будут доступны не по номерам индексов в массивах вариантов, а по значащим именам сцен. Получившийся код (QuestOOP.3.1.4) -классы Character.java, Situation.java, Story.java, Game.java:

```
 //=======персонаж=======
public class Character {
    public int K;
    public int A;
    public int R;
    public String name;

    public Character(String name) {
        K = 1;
        A = 100;
        R = 50;
        this.name = name;
    }
}
//=======ситуация=======
public class Situation {
    Situation[] direction;
    String subject,text;
    int dK,dA,dR;
    public Situation (String subject, String text, int variants, int dk,int da,int dr) {
        this.subject=subject;
        this.text=text;
        dK=dk;
        dA=da;
        dR=dr;
        direction=new Situation[variants];
    }
}
// =======история=======
public class Story {

    private Situation start_story;
    public Situation current_situation;

    Story() {
        start_story = new Situation(
                "первая сделка (Windows)",
                "Только вы начали работать и тут же удача! Вы нашли клиента и продаете ему
 "
                        + "партию ПО MS Windows. Ему в принципе достаточно взять 100 коробо
к версии HOME.\n"
                        + "(1)у клиента денег много, а у меня нет - вы выпишете ему счет на
120 коробок ULTIMATE по 50тр\n"
                        + "(2)чуть дороже сделаем, кто там заметит - вы выпишете ему счет н
а 100 коробок PRO по 10тр\n"
                        + "(3)как надо так и сделаем - вы выпишете ему счет на 100 коробок
 HOME по 5тр ",
                3, 0, 0, 0);
        start_story.direction[0] = new Situation(
                "корпоратив",
                "Неудачное начало, ну что ж, сегодня в конторе корпоратив! "
                        + "Познакомлюсь с коллегами, людей так сказать посмотрю, себя покаж
у",
```

```
                0, 0, -10, -10);
        start_story.direction[1] = new Situation(
                "совещание, босс доволен",
                "Сегодня будет совещание, меня неожиданно вызвали,"
                        + "есть сведения что \n босс доволен сегодняшней сделкой.",
                0, 1, 100, 0);
        start_story.direction[2] = new Situation(
                "мой первый лояльный клиент",
                "Мой первый клиент доволен скоростью и качеством "
                        + "моей работы. Сейчас мне звонил лично \nдиректор компании,  сообщ
ил что скоро состоится еще более крупная сделка"
                        + " и он хотел, чтобы по ней работал именно я!", 0, 0,
                50, 1);
        current_situation = start_story;
    }

    public void go(int num) {
        if (num <= current_situation.direction.length)
            current_situation = current_situation.direction[num - 1];
        else
            System.out.println("Вы можете выбирать из "
                    + current_situation.direction.length + " вариантов");
    }

    public boolean isEnd() {
        return current_situation.direction.length == 0;
    }
}
//=======игра=======
public class Game {

    public static Character manager;
    public static Story story;

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.println("Вы прошли собеседование и вот-вот станете сотрудником Корпораци
и. \n "
                        + "Осталось уладить формальности - подпись под договором (Введите в
аше имя):");
        manager = new Character(in.next());
        story = new Story();
        while (true) {
            manager.A += story.current_situation.dA;
            manager.K += story.current_situation.dK;
            manager.R += story.current_situation.dR;
            System.out.println("=====\nКарьера:" + manager.K + "\tАктивы:"
                    + manager.A + "\tРепутация:" + manager.R + "\n=====");
            System.out.println("============="
                    + story.current_situation.subject + "==============");
            System.out.println(story.current_situation.text);
            if (story.isEnd()) {
                System.out
```

```
                        .println("====================the end!==================");
                return;
            }
            story.go(in.nextInt());
        }

    }

}
```

Выводы

В предыдущем примере была создана не ООП игра-квест. Обратите внимание, что при росте количества сцен сложность восприятия кода сильно возрастает и программа быстро становится нечитабельной. В то же время при использовании ООП описание сцен довольно компактное и код получается простой и прозрачный.

Можно сделать вывод об очевидном преимуществе ОО подхода при разработке ПО. Однако в этих примерах еще не был задействован весь арсенал ООП. По сути, использовалась лишь инкапсуляция. В следующей главе рассмотрим пример, в котором задействованы все преимущества ООП.

3.1.4. Электронный журнал (мини-проект)

Необходимо разработать электронный журнал для школы. Разработку будем проводить только в ОО парадигме.

Детализируем постановку задачи. Итак, есть школа как учебное заведение, находящееся по адресу 344000, г. Ростов-на-Дону, ул. Знаний, д.1, в которой происходит обучение детей по стандартной программе среднего общего образования (11 классов). В школе работают учителя, которые преподают по несколько дисциплин, причем некоторые имеют дополнительную нагрузку в виде классного руководства либо факультативных предметов, кружков. Помимо преподавателей, в школе есть прочий персонал: директор, завуч, охранники, повара, уборщики. Вход в школу осуществляется при предъявлении магнитной карты с уникальным ID. Есть множество документов, регламентирующих процесс образования.

Для реализации деятельности в этом задании выберем следующие документы:

- общий список преподавательского состава с указанием квалификации для ведения отчетности;
- общий список школьников с указанием возраста для ведения отчетности;
- общий список всех людей, имеющих доступ в школу, для выдачи магнитных карт;
- список учеников класса вместе с родителями для организации собраний;
- электронный журнал, каждая страница которого связывает отчетность о посещении/оценках школьников определенного класса по датам с учебным предметом и преподавателем.

Описывать процесс формирования журнала не будем — это общеизвестно. Давайте проведем анализ технического задания (сущность, свойства, действия) (см. табл. 3.3).

Сущности (подлежащие)	**Свойства (дополнения)**	**Действия (глаголы)**
Школа	Сотрудники, список классов, список кружков, список факультативов	Принять на работу сотрудника, принять ученика, выдать общий список имеющих доступ в школу (CardID, ФИО), общий список учителей (ФИО, квалификация), общий список школьников (ФИО, класс, возраст), ЭлЖур
Сотрудник	CardID, ФИО, должность	
Ученик	CardID, ФИО, должность	
Учитель	CardID, ФИО, должность, квалификация	
Родитель	ФИО, номер телефона	
Класс	Номер, список учеников, классный руководитель	Выдать список учеников, список учеников с родителями
Кружок	Название, список учеников, ведущий учитель	Выдать список учеников, список учеников с родителями
Факультатив	Предмет, список учеников, ведущий учитель	Выдать список учеников, список учеников с родителями

Табл. 3.3.

В соответствии с анализом построим UML диаграмму (см. рис.3.3).

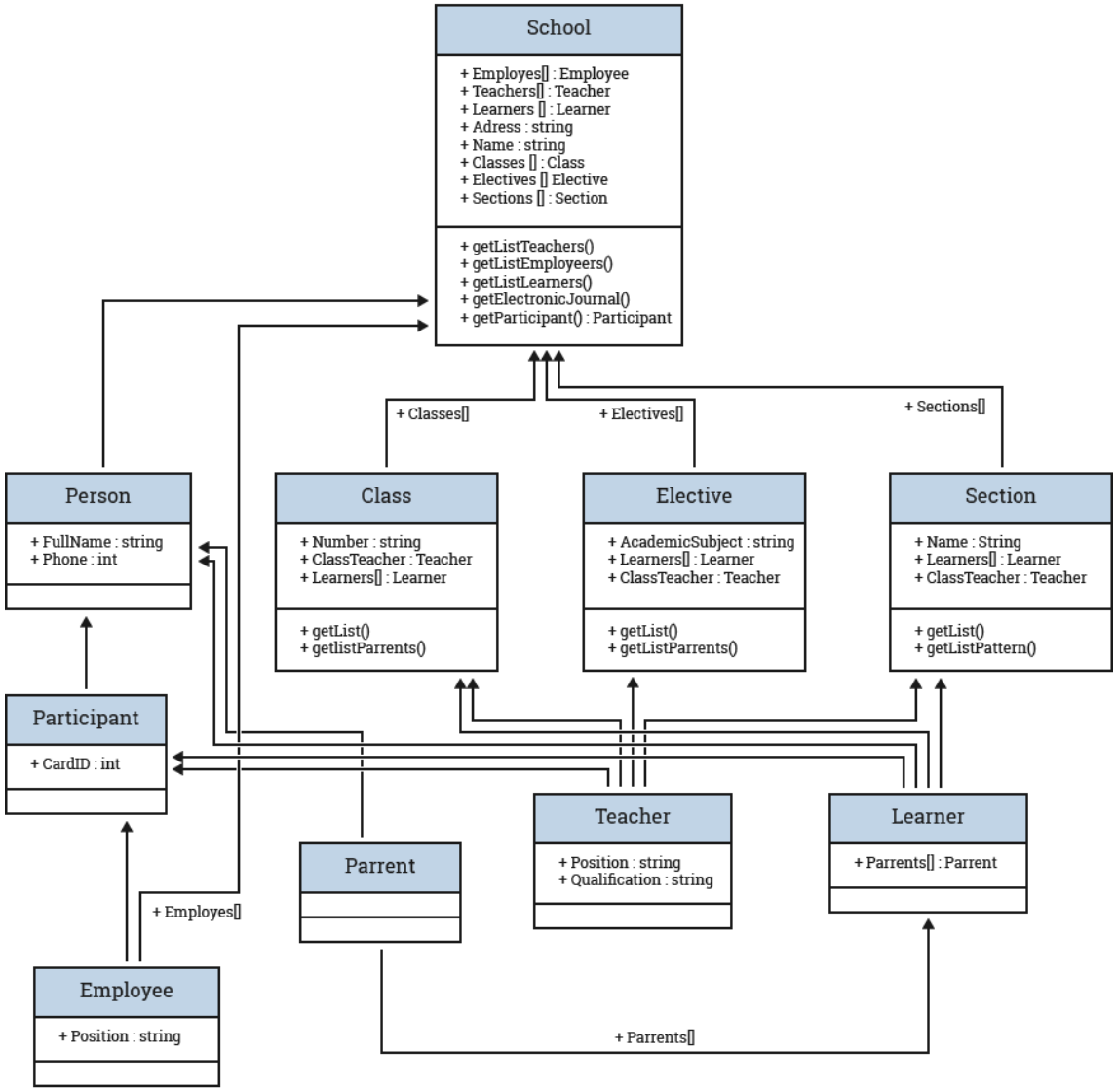


Рис. 3.3.

Итак, по описанной ранее методике была получена диаграмма классов нашей будущей программы. Однако в ней имеется ряд вопросов.

Во-первых, в некоторых классах есть повторяющиеся свойства (например, ФИО, CardID). Это наводит на мысль об общности этих классов.

Во-вторых, в постановке задачи был отчет — общий список всех людей, имеющих доступ в школу. Однако добавить такой метод в класс School не получается, так как в нашей программе нет обобщенного типа для всех участников учебного процесса, а следовательно невозможно вернуть единый массив объектов неодинаковых типов.

Для решения этих вопросов выделим суперклассы для всех участников, описанных в анализе ТЗ. Небольшой анализ показывает, что самый общий класс для всех — Person (персона, человек) с полями ФИО и телефон.

Также выделим две разновидности персон — имеющих доступ в школу (с карточкой — персонал, учителя, ученики) и не имеющих доступа (родители) (см. рис. 3.4).

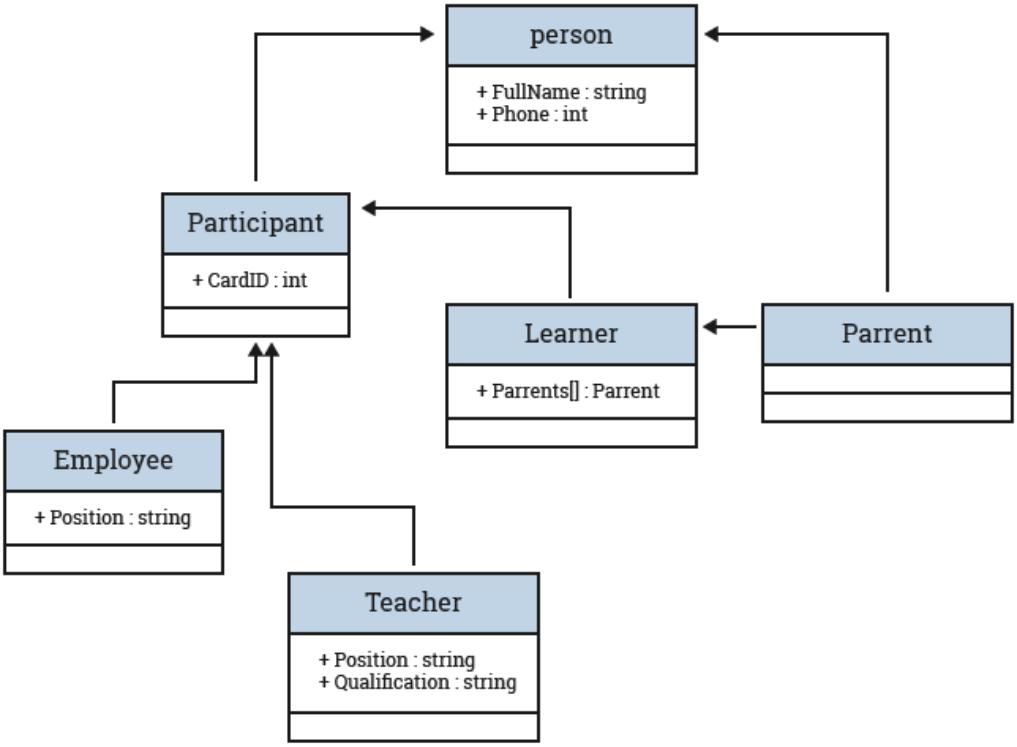


Рис. 3.4.

Изменим общую модель с учетом общности сущностей (наследования классов) (см. рис. 3.5).

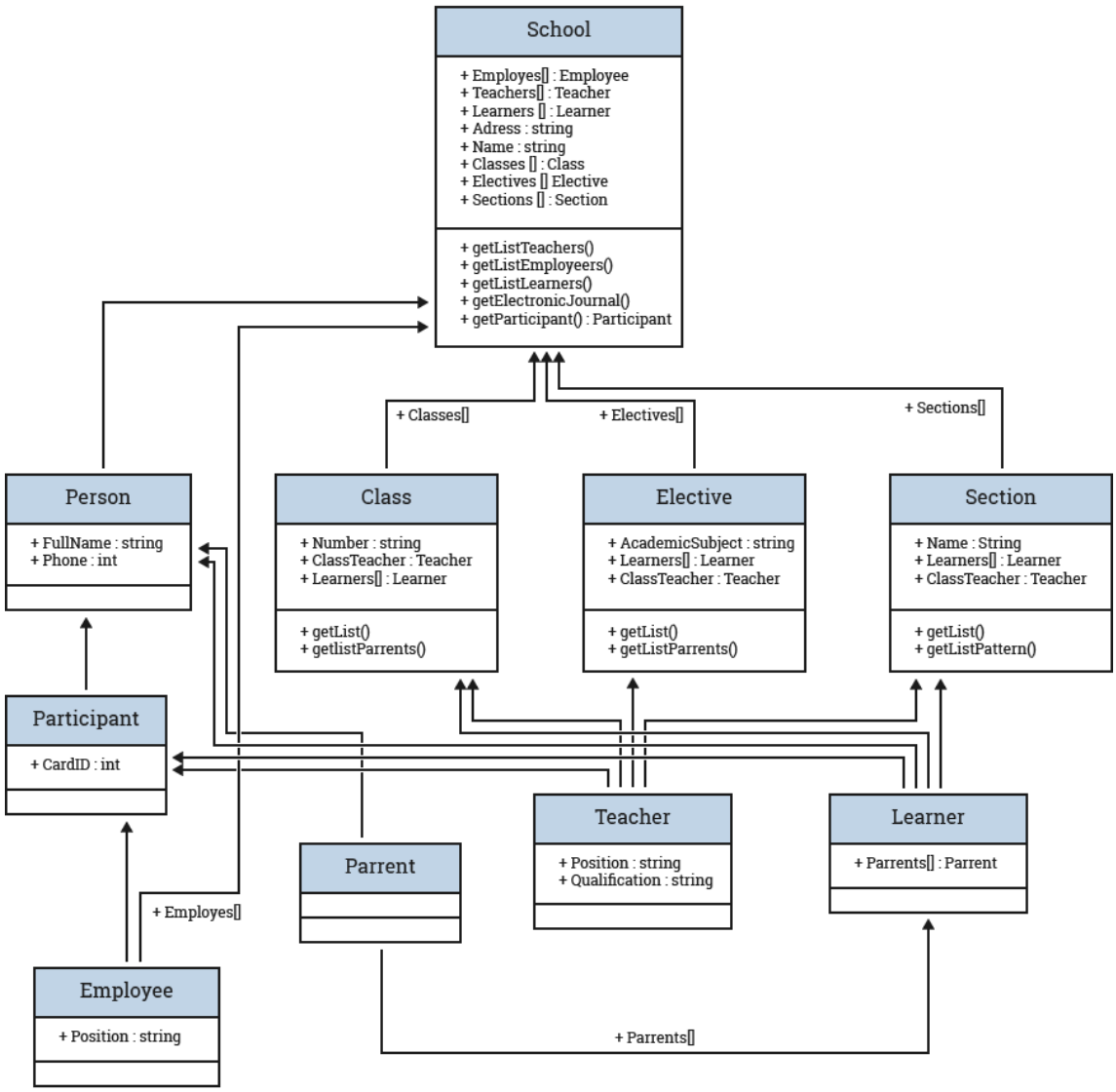


Рис. 3.5.

В приведенной диаграмме поднятые вопросы разрешены. Теперь в классе School есть метод, который вернет массив участников учебного процесса. Обратите внимание, что вследствие полиморфизма этот метод вернет список именно участников (поля ФИО, phone, CardID) независимо от того, какие роли будут у каждого конкретного участника — ученик, учитель или сотрудник. Однако общие методы этих объектов при вызове буду вызваны именно в соответствии фактическому классу.

Конечно, в вышеприведенном UML показано только начало проектирования реальной программы, но уже видно, что использование полиморфизма и наследования дает дополнительную гибкость в проектировании.

3.1.5.* Шаблоны и принципы проектирования

Парадигма ООП давно и устойчиво утвердилась как основная при разработке ПО.
В представленном материале автор старался рассказать просто о сложном, так как проектирование ПО — это очень сложный процесс, которым обычно занимаются самые высококвалифицированные специалисты — системные архитекторы. При проектировании профессионалы помимо известных основных принципов ООП (инкапсуляция, наследование, полиморфизм) опираются на:

- глубокое знание computer science, опыт программирования, построения и управления структурами данных;
- на знание всевозможных фреймворков, программных платформ, сред разработки, серверов баз данных и серверов приложений;
- опыт внедрения и сопровождения ПО.

Все это необходимо для того, чтобы созданный ими проект мог быть не только реализован в кратчайшие сроки наиболее эффективным способом, но и чтобы у заказчика он был наиболее эффективен, то есть выдавал заказанную функциональность с наименьшими затратами.

Конечно, лучшие практики были систематизированы, выработаны шаблоны и принципы проектирования.

Первоначально были выработаны так называемые шаблоны проектирования. Впервые шаблоны были описаны в книге «Шаблоны проектирования: элементы повторно используемого объектно-ориентированного программного обеспечения», написанной Эриком Гамма, Ричардом Хелмом, Ральфом Джонсоном и Джоном Влиссидесом, которых называли бандой четырех (Gang of Four). В книге GOF были представленные готовые «рецепты» для типовых задач. Если в процессе проектирования у вас встречается ситуация, описанная в шаблонах, вы можете не «изобретать велосипед», а взять готовую структуру классов и алгоритмы работы из шаблонов. В свое время очень большое внимание развитию шаблонов проектирования уделяла компания SUN Microsystem. Они разработали наборы шаблонов Java Blueprint. Сегодня известны десятки популярных шаблонов. Классическая литература о шаблонах:

1. Design Patterns: Elements of Reusable Object-Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley, 1995).
2. Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML by Mark Grand (Wiley, 1998).
3. Core J2EE Patterns: Best Practices and Design Strategies by Deepak Alur, John Crupi, and Dan Malks (Prentice Hall, 2001).
4. UML Distilled: Applying the Standard Object Modeling Language by Martin Fowler with Kendall Scott (Addison-Wesley, 2000).
5. The Unified Modeling Language User Guide by Grady Booch, Ivar Jacobson, and James Rumbaugh (Addison-Wesley, 1998).

Однако в процессе проектирования классов важно не только создать правильную архитектуру приложения, но и крайне важно, чтобы сами классы были разработаны правильно. Свод принципов проектирования классов появился совсем недавно, и на сегодня оформлен в виде пяти принципов, которые сокращенно называют **SOLID** (см. табл. 3.4).

Название сокращений	**Описание**
S SRP	Принцип единственной обязанности (Single responsibility principle)
O OCP	Принцип открытости/закрытости (Open/closed principle)

Название сокращений	Описание
L LCP	Принцип подстановки Барбары Лисков (Liskov substitution principle)
I ICP	Принцип разделения интерфейса (Interface segregation principle)
D DIP	Принцип инверсии зависимостей (Dependency inversion principle)

Табл. 3.4.

Принцип единственной обязанности SRP

На каждый объект должна быть возложена одна единственная обязанность. Обязанность — это набор методов, служащих одному действующему лицу. Для обязанности действующее лицо — единственный источник изменений. Например, неправильный код:

```
 public class MainActivity extends Activity implements View.OnClickListener {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Button btnSend = (Button) findViewById(R.id.btnSend);
        btnSend.setOnClickListener((View.OnClickListener) this);
    }

    @Override
    public void onClick(View v) {
        // Разослать рассылку
    }
}
```

Здесь MainActivity помимо своей основной работы создания интерфейса занимается рассылкой. Исправим, разделив на два класса:

```
 public class MainActivity extends Activity {
    private TestButtonHandler testButtonHandler;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        testButtonHandler = new TestButtonHandler(this, R.id.btnTest);
    }
}
public class TestButtonHandler implements View.OnClickListener {
    private Button btnSend;

    public TestButtonHandler(Activity activity, int btnID) {
        btnSend = (Button) activity.findViewById(R.id.btnSend);
        btnSend.setOnClickListener((View.OnClickListener) this);
    }

    @Override
    public void onClick(View v) {
        // Сделать рассылку
    }
}
```

Принцип открытости/закрытости OCP

Программные сущности (классы, модули, функции и т. п.) должны быть открыты для расширения, но закрыты для изменения. Достоинства применения такого подхода следующие:

- не нужно пересматривать уже существующий код, не нужно менять уже готовые для него тесты при доработке проекта;
- если нужно ввести какую-то дополнительную функциональность, то это не должно коснуться уже существующих классов или как-либо иначе повредить уже существующую функциональность.

Например, мы имеем класс, который выполняет необходимые операции, и класс логера, который записывает в файл журнала отметки о выполнении операций:

```
 public class Logger {
    void Log(String logText) {
    // сохранить лог в файле
    }
}
public class AnyStuff {
    private Logger logger;

    public AnyStuff() {
        logger = new Logger();
    }

    public void startAnyOperation(){
        logger.Log("Operation started");
    }
}
```

Теперь изменим код — будем сохранять записи журнала в базу данных:

```
 public class DatabaseLogger {
    void Log(String logText)
    {
    // сохранить лог в БД
    }
}
public class AnyStuff {
    private DatabaseLogger logger;

    public AnyStuff() {
        logger = new DatabaseLogger();
    }
    public void startAnyOperation() {
        logger.Log("Operation started");
    }
}
```

Однако по принципу единственности ответственности класс AnyStuff не отвечает за логирование, почему же изменения коснулись и его? Это произошло потому, что принцип открытости/закрытости был нарушен — AnyStuff не закрыт для модификации. В данном примере пришлось его изменить, чтобы изменить способ хранения логов. Защитить от изменений класс AnyStuff поможет выделение абстракции.

```
 public interface ILogger {
    void Log(String logText);
}

public class Logger implements ILogger {
    public void Log(string logText) {
    // сохранить лог в файле
    }
}

public class DatabaseLogger implements ILogger {
    public void Log(string logText) {
    // сохранить лог в базе данных
    }
}
public class AnyStuff {
    private ILogger logger;

    public AnyStuff(ILogger logger) {
        this.logger = logger;
    }

    public void startOperation() {
        logger.Log("Operation started");
    }
}
```

Теперь изменение реализации логирования уже не приведет к модификации класса AnyStuff.

Принцип подстановки Барбары Лисков LSP

Объекты в программе могут быть заменены их наследниками без изменения свойств программы. Иными словами, поведение наследуемых классов не должно противоречить поведению, заданному базовым классом, то есть поведение наследуемых классов должно быть ожидаемым для кода, использующего переменную базового типа. Классический пример — класс прямоугольник и его наследник квадрат:

```
 public class Rectangle {
    private double height;
    private double width;
    public double area();
    public void setHeight(double height);
    public void setWidth(double width);
}
```

Класс квадрата просто наследуется от класса прямоугольника с той лишь разницей, что поддерживает ширину и высоту эквивалентными.

```
 public class Square extends Rectangle {
    public void setHeight(double height) {
        super.setHeight(height);
        super.setWidth(height);
    }

    public void setWidth(double width) {
        setHeight(width);
    }
}
```

Однако приведенный пример продемонстрирует некорректность такого подхода:

```
 Rectangle rect = new Square();
rect.setWidth(3);
rect.setHeight(10);
System.out.println(rect.area());
```

Полученный результат будет 100 вместо 30. Конечно, видно, что создан был все-таки квадрат, а затем использован в качестве прямоугольника. А если использование объекта класса Rectangle будет совсем не в том месте, где объект был создан? Выход из такой ситуации только один — отказаться от наследования. Варианты исправить ситуацию:

- Rectange и Square будут разными типами;
- Square агрегирует объект типа Rectange.

Принцип разделения интерфейса ISP

Несколько небольших специализированных интерфейсов лучше, чем один большой, универсальный. Принцип разделения интерфейсов состоит в том, что слишком универсальные интерфейсы необходимо разделять на более маленькие и специфические, чтобы при использовании маленьких интерфейсов потребителю нужно было бы реализовать только методы, необходимые ему в работе. Пример интерфейса, созданного с нарушением принципа ISP:

```
public interface Messenger {
   askForCard();
   tellInvalidCard();
   askForPin();
   tellInvalidPin();
   tellNotEnoughMoneyInAccount();
   tellAmountDeposited();
   tellBalance();
}
```

Правильно будет его разделить на два, которые могут быть использованы независимо:

```
public interface LoginMessenger {
   askForCard();
   tellInvalidCard();
   askForPin();
   tellInvalidPin();
}
```

и

```
public interface WithdrawalMessenger {
   tellNotEnoughMoneyInAccount();
   askForFeeConfirmation();
   tellBalance();
}
```

Принцип инверсии зависимостей DIP

Зависимости внутри системы строятся на основе абстракций. Модули верхнего уровня не зависят от модулей нижнего уровня. Абстракции не должны зависеть от деталей. То есть следует проектировать классы таким образом, чтобы различные модули были автономными, и соединялись друг с другом с помощью абстракции. Пример неправильной реализации:

```
public Keyboard {
   char getchar();
}

public Printer {
    void putchar(char c);
}

class CharCopier {
    void copy(Keyboard reader, Printer writer) {
        int c;
        while ((c = reader.getchar()) != EOF) {
                writer.putchar();
        }
    }
}
```

Исправим так, чтобы класс CharCopier не зависел от классов Printer и Keyboard:

```
 public interface Reader {
    char getchar();
}

public interface Writer {
    void putchar(char c)
}

public Keyboard implements Reader {...}

public Printer implements Writer {...}

class CharCopier {
    void copy(Reader reader, Writer writer) {
        int c;
        while ((c = reader.getchar()) != EOF) {
            writer.putchar();
        }
    }
}
```

Заключение

Даже если разработчик создает маленькую программку, которой кроме него, возможно, никто и не воспользуется, все равно не стоит пренебрегать проектированием. Этот этап поможет заранее устранить возможные противоречия в ТЗ, уменьшить количество повторных разработок, которые возникают, когда выясняется, что изначально был выбран неверный/неэффективный способ решения задачи. Кроме того, как известно, многие крупные проекты начинались как маленькие программки для личного пользования. И именно грамотное проектирование позволило им вырасти во что-то серьезное.

3.2. Ввод, вывод и исключения

Сайт: IT Академия SAMSUNG
Курс: MDev @ IT Академия Samsung
Книга: 3.2. Ввод, вывод и исключения
Напечатано:: Егор Беляев
Дата: Суббота, 18 Апрель 2020, 19:23

Оглавление

3.2.1. Зачем нужна обработка исключений

Мысль изреченная есть ложь
Ф.И.Тютчев
Любая написанная программа будет содержать в себе проблемы, которые могут не позволить работать ей, как ожидает автор. Такие проблемы, несмотря на все их многообразие, делят на два основных класса:

- на момент написания кода имеется достаточно информации в текущем контексте, чтобы как-то справиться с трудностью. Например, ошибка синтаксиса приведет к ошибке компиляции (compile time error), которую можно легко исправить;
- исключительная ситуация — это проблема, которая мешает исполнению заданной последовательности действий в алгоритме (runtime error). Исключительная ситуация — это не ошибка в привычном понимании этого слова, она, например, может возникнуть из-за каких-то неблагоприятных внешних условий.

В Java встроен механизм, позволяющий программисту обработать потенциально возможные исключительные ситуации, то есть написать программный код, который выполнится в случае возникновения исключительной ситуации и корректно разрешит ее.

Когда во время выполнения программа попадает в исключительную ситуацию, то у нее нет возможности продолжать обработку данных, потому что нет необходимой информации, чтобы преодолеть проблему в текущем методе (в текущем контексте). Например, программа считывает файл со счетом в игре, а кто-то удалит этот файл. Все, что можно сделать в этом случае, — это выйти из текущего контекста и отослать эту проблему к высшему контексту. Именно так и работает Java-машина (JVM) в этой ситуации. Говорят, что она «выбрасывает исключение», то есть она прерывает исполнение текущего контекста и передает управление вышестоящему контексту (функцией из которого была вызвана текущая), посылая ей дополнительную информацию о причине возникновения ситуации.

Простой пример выбрасывания исключения — деление. Если в процессе работы программы происходит ситуация делителя на ноль, вы должны это предусмотреть. Делитель, не равный нулю, не вызовет у программы никаких затруднений, и она будет выполняться по плану, заложенному вами. Если же в переменную b будет введен 0, до перехода к блоку выполнения деления будет выброшено исключение, то есть передано обработчику исключения дальнейшее разрешение проблемы, а программа продолжит выполнение дальше.

```
 // .....
int a = scanner.nextInt();
int b = scanner.nextInt();
try {
    System.err.println(a/b);
} catch (ArithmeticException e) {
    System.err.println("Произошла недопустимая арифметическая операция");
}
// тут выполняется код после исключительной ситуации
```

Когда выбрасывается исключение, JVM выполняет следующие действия:

- во-первых, создает объект исключения тем же способом, что и любой Java-объект — в куче (Heap Memory), при помощи new;

- затем текущий поток исполнения (который невозможно продолжать) останавливается, и ссылка на объект исключения «выталкивается» из текущего контекста. В этот момент механизм обработки исключений JVM начинает искать подходящее место для продолжения выполнения программы. Это подходящее место — обработчик исключения, чья задача — извлечь проблему, чтобы в алгоритме реакции на эту ситуацию решить проблему, или просто продолжиться, если реакция не нужна.

Также можно послать информацию об ошибке в вышестоящий контекст с помощью создания объекта исключения, представляющего вашу информацию и «выбросить» его из вашего контекста принудительно. Это называется выбрасыванием исключения. Это выглядит так:

```
if(t == null)
   throw new NullPointerException();
```

3.2.2. Ключевое слово throw

JVM может обрабатывать большое количество разнообразных исключительных ситуаций. Но можно программно сгенерировать исключительную ситуацию при помощи оператора throw. Синтаксис следующий:

```
throw <объект_Throwable>
```

Для этого необходимо создать экземпляр класса Throwable или его наследников. Сделать это можно, например, стандартным способом через оператор new. Поток выполнения останавливается непосредственно после оператора throw и следующие за ним операторы не выполняются. При этом ищется ближайший обработчик (блок catch), соответствующего типа исключения. Еще иногда говорят, что объект исключения был «брошен» оператором throw и «пойман» в соответствующем блоке catch.

```
try {
   NullPointerException exception=new NullPointerException("Объекта не существует");
   throw exception;
} catch (NullPointerException e) {
   Toast.makeText(this, e.getMessage(), Toast.LENGTH_LONG).show();
}
```

В этом примере создан новый объект класса NullPointerException. Многие классы исключений кроме стандартного конструктора по умолчанию без параметров имеют второй конструктор со строковым параметром, в котором можно разместить поясняющую информацию об исключительной ситуации. Получить текст из него можно через метод getMessage(), что продемонстрировано в блоке catch.

3.2.3. Ключевое слово throws

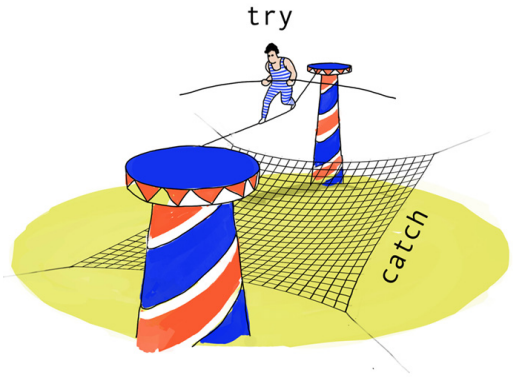
Не всегда можно предпринять нужные действия для правильной реакции на исключительную ситуацию в том месте, где она возникла. Например, в этом месте недостаточно информации для выполнения действий для исправления ситуации, или разработчик не посчитал целесообразным делать обработку исключения в текущем методе. В таких случаях обрабатывают исключение не в том методе, где оно возникло, а в том методе, который его вызвал. Тогда в описании метода необходимо объявить (предупредить), что он может стать причиной возникновения исключения указанного типа. Это делается с помощью специального ключевого слова throws.

Например, разрабатываемый метод getAllScores() отвечает за чтение результатов игры из файла. Так же имеем в виду, что использованный в нем метод чтения из файла read() может вызывать исключение IOException. Разработчик обязан либо указать блок обработки потенциального исключения в месте его возникновения, либо отметить свой метод как потенциально «опасный» возникновением исключения. В примере ниже объявлено, что метод getAllScores() может вызывать исключение IOException:

```
class MySuperGame{
   void getAllScores() throws IOException{
        file.read();
   }
```

3.2.4. Обработка исключения с помощью конструкции try-catch

Если при исполнении тела метода где-то произошло (было выброшено) исключение, предполагается что оно будет где-то «поймано» и приняты меры по устранению обнаружившейся проблемы. Для демонстрации «поимки» исключения сначала рассмотрим концепцию критического блока.



Критический блок — секция кода, в котором может возникнуть исключение и за которой следует код, обрабатывающий это исключение.

Если в процессе выполнения тела метода вы выбросили исключение (или другой метод, вызванный вами внутри этого метода, выбросил исключение), то такой метод перейдет в состояние «бросания». Если ничего не предпринимать, вы будете выброшены из текущего метода. Если же вы не хотите быть выброшенными из него, вы можете установить специальный блок внутри текущего метода для поимки исключения. Это так называемый блок проверки. Он называется так, потому что вы «проверяете» ваш программный код, выполняемый в нем. Блок проверки — это обычный блок, предваряемый ключевом словом try:

```
try {
   // здесь расположен код
   // который может сгенерировать
   // исключение
}
```

При написании программного кода все потенциально «проблемные» операторы и вызовы методов помещают в блок проверки и все исключения ловятся в одном месте. Такой код становится гораздо компактнее и понятнее, так как код программы не смешивается с проверкой ошибок.

После того как описан блок проверки (в котором могут быть брошены исключения), необходимо выполнить действия по результатам проверки. Эти действия задаются в обработчике исключений. Для каждого типа исключения, которые вы хотите поймать, можно определить отдельный обработчик. Обработчики исключений следуют сразу за блоком проверки и объявляются как обычный блок, предваряемый ключевым словом catch:

```
 try {
    // Код, который может сгенерировать исключение
} catch(ExceptionType1 ex1) {
    // Обработка исключения ExceptionType1
} catch(ExceptionType2 ex2) {
    // Обработка исключения ExceptionType2
} catch(ExceptionType3 ex3) {
    // Обработка исключения ExceptionType3
}
// и т.д.
```

Синтаксис catch-предложения (обработчика исключения) похож на запись метода, который принимает один и только один аргумент определенного типа. Переданные ему как аргументы объекты (ex1, ex2 и так далее) могут быть использованы внутри обработчика. Конечно, необязательно использовать эти объекты, так как часто достаточно самого факта вызова данного обработчика для его обработки, но идентификатор все равно должен быть. Обработчики должны располагаться сразу после блока проверки. Если выброшено исключение, механизм обработки исключений ищет первый обработчик с таким аргументом, тип которого совпадает (или являться родительским) с типом исключения. После нахождения соответствующего обработчика. исполняется его тело. Поиск обработчика, после остановки на предложении catch, заканчивается.

Даже если внутри блока проверки могут возникать несколько исключений одного типа, соответствующий блок обработчика должен быть всего один.

Стратегия обработки исключения. Механизм исключений Java позволяет разработчику перехватывать потенциально некорректную работу программы, которая может возникнуть в какой-то нештатной ситуации и вызвать аварийное завершение программы либо некорректную ее работу. Однако как именно использует этот механизм разработчик, зависит только от него.

Например, худшая практика в обработчике — ничего не предпринять и просто продолжить работу. При этом пользователь не будет уведомлен о проблеме и продолжение работы программы может привести к серьезным проблемам. К примеру, вследствие ошибки файлового доступа могут быть потеряны (не сохранены) данные о текущей финансовой операции, но при подобной реакции в обработчике пользователь даже не узнает, что его финансовые данные уже неверны, хотя программа визуально продолжит работать нормально.

Правильнее, даже если невозможно исправить ситуацию, просто уведомить пользователя о проблеме и завершить работу программы (метода). Такая модель обработки исключительных ситуаций называется ***прерыванием***.

Альтернативная модель называется ***возобновлением***. При такой модели предполагается, что в обработчике предпринимаются меры по предотвращению причины нештатной ситуации, после чего начинается повторная попытка выполнения критичного блока. Например, если при считывании данных из сети происходит исключительная ситуацию, связанная с обрывом соединения, в обработчике можно попытаться возобновить соединение (либо подключиться к альтернативному серверу), после чего попытаться опять считывать данные. Для реализации такой модели можно использовать рекурсию или поместить блок try-catch в цикл while:

```
while(true) {
   try {
       //код, выбрасывающий исключение
       break;
   }catch(IOException e) {
       // Здесь обрабатывается исключение для восстановления
   }
}
```

Обработчик «заглушка». Можно создать обработчик, «ловящий» все типы исключений. Для этого в блоке catch необходимо перехватить исключение базового типа Exception:

```
catch(Exception e) {
   System.err.println("Что то пошло не так!");
}
```

Этот код поймает любое исключение, поэтому его нужно помещать в конце списка обработчиков для предотвращения перехвата у любого более специализированного обработчика.

3.2.5. Ключевое слово finally

Появление механизма обработки исключительных ситуаций приводит к тому, что исполнение программы может происходить не по намеченному алгоритму. Например, в алгоритме предусмотрена следующая последовательность операций: открытие файла, запись в файл, закрытие файла. Однако в процессе выполнения программы при записи в файл может произойти исключительная ситуация, после чего заданный поток исполнения будет прерван и управление будет передано найденному обработчику. Но при этом до операции закрытия файла дело не дойдет. Это может быть проблемой. Для таких ситуаций, чтобы не упустить критически важные операции из-за исключений, был предложен механизм finally.

Ключевое слово finally предваряет блок кода, который будет выполнен после завершения блока try-catch, но перед кодом, следующим за ним. Этот блок будет выполнен в любом случае, независимо от того, возникнет исключение или нет. Синтаксически оператор finally не обязателен, однако каждый оператор try требует наличия catch и/или finally.

```
 try {
    // блок проверки
}
catch (ExceptionType1 ex1) {
    // обрабатываем ошибку первого типа
}
catch (ExceptionType2 ex2) {
    // обрабатываем ошибку второго типа
}
finally {
    // код, который нужно выполнить после завершения блока try-catch
}
```

3.2.6. Основные методы класса Exception

Функция	Описание
String getMessage( )	Возвращает сообщение об исключении
String toString( )	Возвращает короткое описание исключения
void printStackTrace( ), void printStackTrace(PrintStream), void printStackTrace(PrintWriter)	Выдача в стандартный или указанный поток полной информации о точке возникновения исключения

Табл. 3.5.

Кроме того, в базовом классе Throwable Object (базовый тип для всего) существуют другие методы. Один из них, который может быть удобен для исключений, это getClass(), который возвращает объектное представление класса этого объекта. Вы можете спросить у объекта этого класса его имя с помощью getName() или toString(). Обычно эти методы не используются при обработке исключений.

Пример 3.1

Приведем пример использования основных методов Exception:

```
public class ExceptionMethods {
   public static void main(String[] args) {
       try {
           throw new Exception("Пробное исключение");
       } catch(Exception e) {
            System.err.println("Обрабатываем исключение");
            System.err.println("e.getMessage(): " + e.getMessage());
            System.err.println("e.getLocalizedMessage(): " +e.getLocalizedMessage());
            System.err.println("e.toString(): " + e);
            System.err.println("e.printStackTrace():");
            e.printStackTrace(System.err);
       }
   }
}
```

Вывод этой программы:

```
 Обрабатываем исключение
e.getMessage(): Пробное исключение
e.getLocalizedMessage(): Пробное исключение
e.toString(): java.lang.Exception: Пробное исключение
e.printStackTrace():
java.lang.Exception: Пробное исключение
               at ExceptionMethods.main(ExceptionMethods.java:7)
```

В этом примере методы обеспечивают больше информации, чем просто тип исключения. Таким образом, имея на вооружении данные методы, мы всегда можем точно определить, где и почему произошло исключение, что может помочь не столько в обработке исключений, сколько в отладке программы.

3.2.7. Работа с файлами

Класс `File` и его методы

Абстракция ввода/вывода (как система классов) в Java достаточно сложна, гораздо сложнее, чем, например, в классических C или Pascal. Это связано с тем, что ввод/вывод ориентирован не только на простую побайтную запись в файлы/чтение, но и на I/O в консоль, сетевые соединения, объекты в памяти и еще более сложные конструкции. Кроме того, сам обмен помимо последовательного доступа может осуществляться: в случайном порядке, буферизованный, бинарный, посимвольный, построчный, пословный и т. п.). Также фреймворк I/O в Java активно развивается, что вызывает появление новых возможностей.

В связи с таким богатством возможностей и гибкостью подсистемы ввода/вывода в Java представлено множество классов, реализующих эти возможности. В этой главе будет рассмотрена небольшая их часть. Ниже представлен один из фреймворка — утилита, использующаяся в работе с директориями и файлами.

Класс File, в отличие от языков C и Pascal, не предназначен для записи и чтения из файла. Он скорее предназначен для работы с файлами и папками как объектами файловой системы. Вы можете совершать операции с файлами — создавать, удалять, получать параметры файла или список вложенных файлов, если это папка. Также класс File может использоваться для создания каталога или дерева каталогов. У класса очень много методов. Вот некоторые из них (см. табл. 3.6).

Метод	Описание
getAbsolutePath()	Абсолютный путь файла, начиная с корня системы. В Android/Linux/MacOS корневым элементом является символ слеша (/)
canRead()	Доступно для чтения
canWrite()	Доступно для записи
exists()	Файл существует или нет
getName()	Возвращает имя файла
getParent()	Возвращает имя родительского каталога
getPath()	Путь
lastModified()	Дата последнего изменения
isFile()	Объект является файлом, а не каталогом
isDirectory ()	Объект является каталогом
isAbsolute()	Возвращает true, если файл имеет абсолютный путь
renameTo(File newPath)	Переименовывает файл. В параметре указывается имя нового имени файла. Если переименование прошло неудачно, то возвращается false
delete()	Удаляет файл. Также можно удалить пустой каталог
length()	Возвращает размер файла в байтах

Табл. 3.6.

Приведем пример использования некоторых из этих методов:

```
 File fl = new File("/example");
System.out.println ("Имя файла:" + fl .getName());
System.out.println ("Путь:" + fl.getPath());
System.out.println ("Полный путь:" + fl.getAbsolutePath());
System.out.println ("Родительский каталог:" + fl.getParent());
System.out.println (fl.exists() ? "существует" : "не существует");
System.out.println (fl.canWrite() ? "можно записывать" : "нельзя записывать");
System.out.println (fl.canRead() ? "можно читать" : "нельзя читать");
System.out.println ("is" + ("Директория? "+(fl.isDirectory() ? "да": "нет")));
System.out.println (fl.isFile() ? "обычный файл" : "не обычный файл");
System.out.println ("Последняя модификация файла:" + fl.lastModified());
System.out.println ("Размер файла:" + fl.length() + " Bytes");
```

3.2.8. Чтение/запись в файл

Обратите внимание, что большинство методов, связанных с созданием файлов, записью и т. д., обычно отмечены разработчиками Java как throws, то есть как потенциальные источники исключительных ситуаций, связанных с файловыми операциями. В связи с этим необходимо каким-либо образом выполнять обработку исключений.

Центральным элементом системы ввода/вывода является поток (stream). Это сущность, описывающая возможность получения данных из какого-либо источника (например, файла), либо записи в какой-либо источник. В Java это наследники абстрактных классов InputStream/OutputStream. Например, это классы FileInputStream/FileOutputStream. Их можно использовать для побайтного чтения/записи в файлы. Ниже приведен фрагмент кода — классический пример побайтного копирования файла:

```
 in = new FileInputStream("a.txt");
out = new FileOutputStream("b.txt");
int c;
while ((c = in.read()) != -1) {
    out.write(c);
}
```

Конечно, следует обеспечить обработку потенциальных исключительных ситуаций, например, при помощи блока try-catch, а также закрыть в конце потоки методом close().

Однако обычно побайтно считывать информацию неудобно, поэтому существует множество дополнительных классов. Можно использовать более высокоуровневые классы — например, PrintStream. Ярким примером потоков является стандартные потоки System.in (типа InputStream), System.out (типа PrintStream), System.err (типа PrintStream), которые всегда существуют как объекты в работающей java-программе и обычно используются для вывода сообщений в консоль:

```
System.out.println("Это сообщение в консоль");
```

Класс PrintWriter

Еще один высокоуровневый класс PrintWriter очень близок к стандартным возможностям языка C. При создании объекта этого класса входным параметром для него служит уже открытый поток. Методы print(), println() и printf() определены для всех стандартных типов и очень похожи на соответствующие процедуры языка C. Например, ниже приведен фрагмент кода, позволяющий записать в файл FF (шестнадцатеричное представление 255):

```
try {
   File file = new File("d:/testfiles.txt");
   PrintWriter writer = new PrintWriter(new FileWriter(file));
   writer.printf("%x", 255); //Записываем текст в файл
   writer.close(); // Закрываем файл
} catch (IOException e) {
   e.printStackTrace();
}
```

В этом примере создается объект файла с именем *testfiles.txt* (но не сам файл), затем создается поток FileWriter (в это момент файл реально создастся), и, наконец, он оборачивается в объект PrintWriter. Обратите особое внимание на то, что после записи каких-либо данных в файл его необходимо закрыть, только после этого данные действительно запишутся в файл.

Класс Scanner

Начиная с версии 1.5, в Java появился класс Scanner в пакете *java.util*. При создании объекта ему в качестве аргумента могут передаваться файл или поток для считывания. Далее для каждого из базовых типов **<T>** имеется пара методов:

- hasNext() проверяет можно ли далее прочесть данные типа T;
- next() для считывания данных этого типа.

Например, метод nextInt() считывает очередной int, а метод hasNextDouble() возвращает истину или ложь в зависимости от того, есть ли в потоке следующее значение double для чтения. Например, чтение строк файла *a.txt* и вывод их в консоль построчно:

```
 try {
    File file = new File("d:/a.txt");
    Scanner scanner = new Scanner(file);
    while (scanner.hasNext()) {
          System.out.println(scanner.next());
    }
    scanner.close();
} catch (FileNotFoundException e) {
    e.printStackTrace();
}
```

Хотя Scanner и не является потоком, у него тоже необходимо вызывать метод close(), который закроет используемый за основной источник поток.

Несмотря на то что обычно при завершении программы на Java, открытые файлы закрываются автоматически, все же рекомендуется обеспечить корректное закрытие потоков даже при возможном возникновении исключений. Для этого удобнее всего метод close() включать в блок finally.

Пример 3.2

В рассматриваемом примере необходимо написать функцию, которая принимает строки с именами файлов, исходного и целевого. Необходимо скопировать содержимое исходного текстового файла в выходной с заменой слова «School» на слово «Школа». Например, файл *in.txt*:

```
IT School SAMSUNG – программа дополнительного образования по основам IT и программир
ования. Она создана компанией Samsung с целью обучить 5 000 школьников в более чем 2
0 городах России в течение 5 лет.
```

то выходной файл *out.txt*:

```
IT Школа SAMSUNG – программа дополнительного образования по основам IT и программиро
вания. Она создана компанией Samsung с целью обучить 5 000 школьников в более чем 20
городах России в течение 5 лет.
```

Для решения поставленной задачи в создаваемом методе, назовем его replace, воспользуемся низкоуровневыми инструментами Scanner и PriontWriter:

```
 void replace(String fileIn,String fileOut) {
    File in=new File(fileIn);
    File out=new File(fileOut);
    Scanner sc=new Scanner(in);
    PrintWriter pw=new PrintWriter(out);
        // код копирования будет здесь
}
```

Далее в цикле будем пословно считывать исходный файл и писать в целевой, с заменой слова, конечно. Ниже приведен код копирования:

```
 while(sc.hasNext()){
    String word=sc.next();
    if(word.equals("School"))
              word="Школа";
    pw.print(word+" ");
}
```

Важно не забыть закрыть файлы в конце:

```
 sc.close();
pw.close();
```

При работе с файлами часто имеются критические блоки кода, требующие обработки исключений. В имеющейся постановке задачи недостаточно данных для качественной обработки, поэтому принимаем решение делегировать обработку вышестоящему контексту, добавив в описание функции ключевого слова throws и типа делегируемого исключения.
В итоге функция имеет следующий вид:

```
 void replace(String fileIn,String fileOut) throws FileNotFoundException{
    File in=new File(fileIn);
    File out=new File(fileOut);
    Scanner sc=new Scanner(in);
    PrintWriter pw=new PrintWriter(out);
    while(sc.hasNext()){
        String word=sc.next();
        if(word.equals("School"))
                      word="Школа";
        pw.print(word+" ");
    }
    sc.close();
    pw.close();
}
```

Протестировать использование функции replace можно следующим образом:

```
 public static void main(String[] args) {
    FileIO test=new FileIO();
    try{
     test.replace("in.txt", "out.txt");
    }catch(Exception ex){
        System.out.println("Что то пошло не так: "+ex.getMessage());
    }
}
```

3.3. Внутренние и анонимные классы

Сайт: IT Академия SAMSUNG
Курс: MDev @ IT Академия Samsung
Книга: 3.3. Внутренние и анонимные классы
Напечатано:: Егор Беляев
Дата: Суббота, 18 Апрель 2020, 19:23

Оглавление

3.3.1. Понятие внутреннего класса

При разработке программы иногда бывает необходимо создать какой-то небольшой вспомогательный класс, который нецелесообразно выделять в какую-то отдельную сущность, создавать для него отдельный файл и т. д. Например, класс Destination (назначение) внутри класса Parcel (посылка). В Java есть соответствующая синтаксическая возможность. Можно объявлять одни классы внутри других — вложенные классы (nested classes). Например так:

```
 public class Parcel {     // посылка
//...
    class Destination {     // место назначения
        public String street;     // улица
        public int homeNumber;     // номер дома
        public int roomNumber;     // номер квартиры
    }
//...
}
```

В примере выше был объявлен класс Destination внутри класса Parcel. Вложенные классы могут оказаться полезными в следующих ситуациях:

- когда вложенный класс слишком незначителен, чтобы выделять его как отдельный полноценный класс — улучшение читаемости кода;
- когда вложенный класс используется только в одном другом классе и больше нигде в программе — группировка связанных классов;
- когда вложенному классу необходим доступ к полям и методам другого класса — инкапсуляция.

Как можно заметить, синтаксис объявления совершенно идентичен синтаксису объявления обычного класса, только он пишется внутри класса — внешнего класса.

Вложенные классы могут быть объявлены с ключевым словом static, в таком случае это статические вложенные классы (static nested), а могут быть объявлены без ключевого слова static, такие вложенные классы называется внутренними (inner).

Таким образом, в примере выше приведен частный случай вложенного класса — внутренний класс. Именно о таких классах речь пойдет далее.

Важной особенностью именно внутренних классов (в отличие от статических) является то, что они имеют доступ ко всем полям и методам внешнего класса. Внутренние классы в Java делятся на 3 таких типа:

- внутренние классы-члены (member inner classes) — в примере выше как раз такой класс;
- локальные классы (local inner classes);
- анонимные классы (anonymous inner classes).

Обсудим все 3 типа по порядку, уделив особое внимание анонимным классам как самым часто используемым на практике в Android-программировании.

3.3.2. Внутренние классы-члены

Внутренние классы-члены описываются внутри внешнего (охватывающего) класса наряду с полями и методами внешнего класса. Синтаксис объявления такого класса:

```
 // охватывающий класс
public class OuterClass{
    /* здесь объявление методов и полей класса OuterClass*/

    // внутренний класс-член
    class InnerMemberClass{
        /* здесь объявление методов и полей класса InnerClass*/
    }

    /* здесь объявление методов и полей класса OuterClass*/
}
```

Такой класс ассоциируются не с самим внешним классом, а с конкретными его экземплярами. Это проявляется, в частности, в том, как необычно будет выглядеть создание объекта такого класса:

```
 Parcel parcel = new Parcel(); // создаем посылку - внешний класс
Parcel.Destination destination = parcel.new Destination(); // для данного экземпляра класса
Посылка
                                                           // создаем экземпляр класса Назна
чение
```

На внутренние классы-члены в Java действуют некоторые ограничения:

- они не могут содержать статических объявлений, за исключением static final полей-констант;
- внутри таких классов нельзя объявлять перечисления.

3.3.3. Локальные внутренние классы

Этот тип внутреннего класса отличается тем, что он объявляется внутри методов класса. Синтаксис объявления локального внутреннего класса следующий:

```
public class OuterClass{

   <MethodType> <MethodName>([Arguments]){
       //код метода здесь
       class LocalInnerClass{
       }
       //код метода здесь
   }

}
```

Например:

```
public class Sample {

       public void createNewPerson(String name){
           String nameFIO=name;
           // локальный внутренний класс
           class GenericName {
               List<String> names=new ArrayList<String>();
               public GenericName() {
                   for(String str: nameFIO.split(" ")) names.add(str);
               }
               public String getFirstName(){return names.get(0);}
               public String getLastName(){return names.get(names.size()-1);}
           }
           GenericName gname=new GenericName();
           System.out.println("Person:"+gname.getFirstName().charAt(0)+"."+gname.getLastNa
me());
       }

   public static void main(String[] args) {
       Sample sample=new Sample();
       sample.createNewPerson("Андрей Иванович Петров");
   }

}
```

Здесь в методе createNewPerson объявляется вспомогательный класс GenericName, который помогает разобрать полное имя человека на компоненты. Это достаточно типичное применение внутреннего локального класса. Локальные классы (как и все внутренние классы) имеют доступ ко всем полям и методам обрамляющего класса, кроме этого, они еще имеют доступ к переменным метода, в котором объявлены, если те являются final.

Требование объявлять переменную, к которой есть обращения из локального внутреннего класса final, имеется в Java только до версии 8. Начиная с Java 8 это делать необязательно.

У локальных классов свои ограничения:

- они видны только в пределах блока, в котором объявлены (внутри фигурных скобок);
- они не могут содержать в себе статических объявлений за исключением констант — static final.

3.3.4. Анонимные классы

Самый часто используемый на практике вид внутреннего класса. Синтаксис объявления/использования:

```
new <ИМЕНИ НЕТ> <класс родитель/реализуемый интерфейс>([параметры конструктора родителя])
{
   <тело анонимного класса>
}
```

Пример использования при программировании для Android:

```
public class MainActivity extends AppCompatActivity {
   Button button;

   @Override
   protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.activity_main);
       button= (Button) findViewById(R.id.button);
       View.OnClickListener listener=new View.OnClickListener() {
           @Override
           public void onClick(View view) {
               Toast.makeText(MainActivity.this, "Bingo!", Toast.LENGTH_SHORT).show();
           }
       };
       button.setOnClickListener(listener);
   }
}
```

Из примера видно, что создание анонимного класса представляет из себя создание объекта безымянного типа наследника родительского класса (либо реализация интерфейса) с последующим написанием за ним в фигурных скобках тела анонимного класса — наследника. Поскольку имя описываемого класса не указывается, то становится понятно важное ограничение анонимных классов — они не могут содержать конструкторы. Аргументы, указанные в скобках, автоматически используются для вызова конструктора базового класса с указанными параметрами. В остальном анонимные классы имеют те же ограничения, что и локальные внутренние классы.

3.3.5. Текстовый квест (мини-проект, продолжение)

Теперь перепишем мини-проект нашего карьерного квеста под Android с использованием анонимных классов — проект AndroidQuest.3.3.1. Для начала создадим разметку:

```
 <?xml version="1.0" encoding="utf-8"?>
<LinearLayout 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:orientation="vertical"
    tools:context="ru.samsung.itschool.book.androidquest.MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Статус"
        android:id="@+id/status"
        />
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Заголовок ситуации"
        android:id="@+id/title"
        android:textStyle="bold"
        />
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Описание ситуации"
        android:id="@+id/desc"
        />
    <LinearLayout
        android:orientation="horizontal"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:id="@+id/layout"/>
</LinearLayout>
```

Как видно, в приведенном лайоуте просто 3 текстовых поля для отображения соответствующих компонентов ситуации (текущий статус персонажа, название ситуации и ее детальное описание). LinearLayout в нижней части нужен для того, чтобы в ходе выполнения программы на нем размещать кнопки, служащие для выбора желаемого варианта развития событий.
Вот как эта разметка будет выглядеть уже непосредственно в ходе игры (см. рис. 3.6).

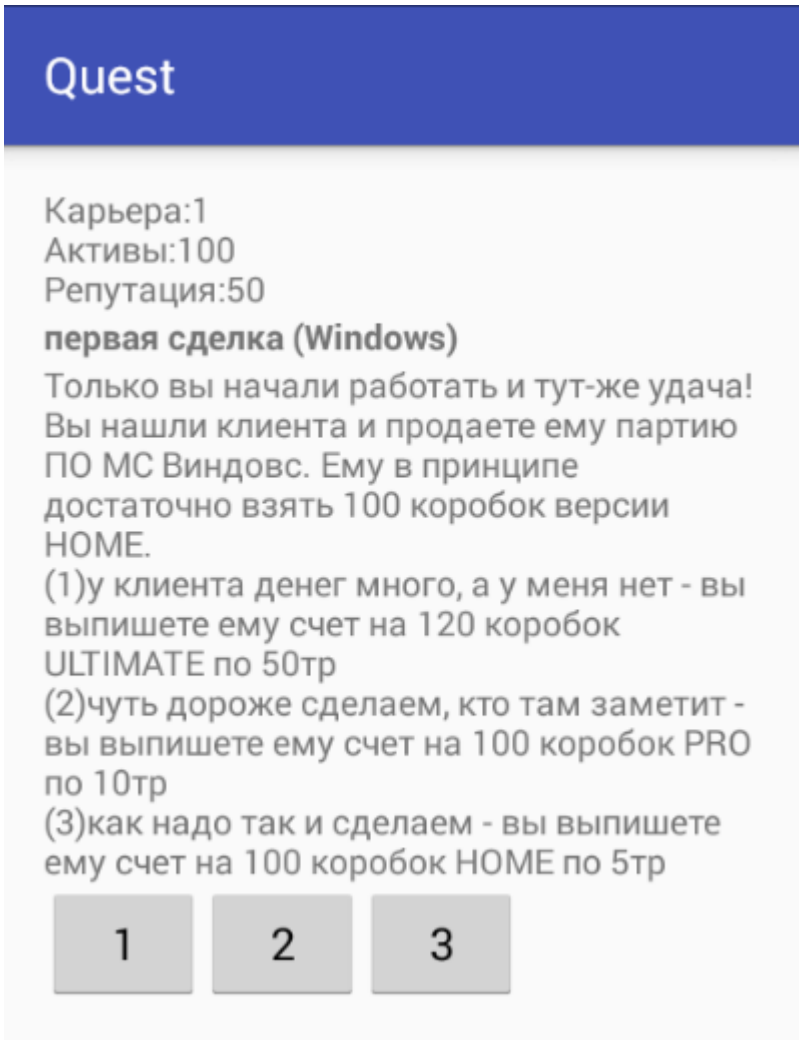


Рис. 3.6.

Пришло время написать код приложения, которое создаст на экране мобильного устройства эту картинку. Обратите внимание, что здесь используются те же классы, что и в текстовой версии игры, и в них ничего переделывать не надо. В этом и проявляются преимущества объектного подхода к программированию. Ядро игры получилось независимым от платформы. Переписав лишь десяток-другой строчек кода, получаем версию игры для другой платформы. Фактически в предыдущей реализации был заменен только класс Game на класс MainActivity. В первоначальном подходе, без использования ООП, код игры пришлось бы переписывать полностью, потому что логика игрового процесса была бы перемешана с кодом, специфичным именно для данной платформы.

```
public class MainActivity extends AppCompatActivity {
   Character manager; // персонаж
   Story story; // история (сюжет)

   @Override
   protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.activity_main);
       // создаем нового персонажа и историю
       manager = new Character("Вася");
       story = new Story();
       // в первый раз выводим на форму весь необходимый текст и элементы
       // управления
       updateStatus();
   }

   // метод для перехода на нужную ветку развития
   private void go(int i) {
       story.go(i + 1);
       updateStatus();
       // если история закончилась, выводим на экран поздравление
       if (story.isEnd())
           Toast.makeText(this, "Игра закончена!", Toast.LENGTH_LONG).show();
   }

   // в этом методе размещаем всю информацию, специфичную для текущей
   // ситуации на форме приложения, а также размещаем кнопки, которые
   // позволят пользователю выбрать дальнейший ход событий
   private void updateStatus() {
       // не забываем обновить репутацию в соответствии с новым
       // состоянием дел
       manager.A += story.current_situation.dA;
       manager.K += story.current_situation.dK;
       manager.R += story.current_situation.dR;
       // выводим статус на форму
       ((TextView) findViewById(R.id.status)).
               setText("Карьера:" + manager.K +
                       "\nАктивы:" + manager.A + "\nРепутация:" + manager.R);
       // аналогично для заголовка и описания ситуации
       ((TextView) findViewById(R.id.title)).
               setText(story.current_situation.subject);
       ((TextView) findViewById(R.id.desc)).
               setText(story.current_situation.text);
       ((LinearLayout) findViewById(R.id.layout)).removeAllViews();
       // размещаем кнопку для каждого варианта, который пользователь
       // может выбрать
       for (int i = 0; i < story.current_situation.direction.length; i++) {
           Button b = new Button(this);
           b.setText(Integer.toString(i + 1));
           final int buttonId = i;
           // Внимание! в анонимных классах
           // можно использовать только те переменные метода,
           // которые объявлены как final.
```

```
            // Создаем объект анонимного класса и устанавливаем его
            // обработчиком нажатия на кнопку
            b.setOnClickListener(new View.OnClickListener() {
                @Override
                public void onClick(View v) {
                    go(buttonId);
                    // поскольку анонимный класс имеет полный
                    // доступ к методам и переменным родительского,
                    // то просто вызываем нужный нам метод.
                }
            });
            // добавляем готовую кнопку на разметку
            ((LinearLayout) findViewById(R.id.layout)).addView(b);
        }
    }

}
```

В приведенном примере анонимный класс используется для обработки события нажатия на кнопку, а информация в него передается при помощи константы, созданной в методе перед созданием анонимного класса. Это достаточно типичное для Android-программирования применение анонимных классов.

3.4. Параллелизм и синхронизация, потоки

Сайт: IT Академия SAMSUNG
Курс: MDev @ IT Академия Samsung
Книга: 3.4. Параллелизм и синхронизация, потоки
Напечатано:: Егор Беляев
Дата: Суббота, 18 Апрель 2020, 19:24

Оглавление

3.4.1. Общие понятия

С постоянным повышением сложности задач повышается и требование к вычислительным способностям (мощностям) компьютеров. Компьютерные технологии постоянно совершенствуются, и одним из требований к производству вычислительных систем является соблюдение закона Гордона Мура (основателя корпорации Intel), в соответствии с которым «производительность вычислительных систем должна удваиваться каждые 18 месяцев». Это способствовало появлению многоядерных процессоров. Если когда-то многоядерность была не распространена среди персональных компьютеров, а использовалась для специализированных серверных приложений, то на сей день даже мобильные устройства содержат в себе многоядерные процессоры (2-х, 4-х ядерные), а компьютерные процессоры насчитывают десятки ядер.

С приходом многоядерности появилось понятие ***параллелизма*** — одновременного выполнения нескольких вычислений. До этого все задачи процессор выполнял последовательно. Конечно, и при одноядерных процессорах можно было просматривать сайты в браузере, слушать музыкальный плеер и в это время копировать файлы на флешку, для пользователя все это выглядело как одновременное (параллельное) выполнение, но на самом над этими процессами работал всего один процессор. Для того чтобы создать видимость параллельного выполнения, процессы (программы) разделялись на потоки. Процессор поочередно выполнял потоки из разных процессов, таким образом создавалось впечатление параллелизма, это называется — псевдопараллелизмом.

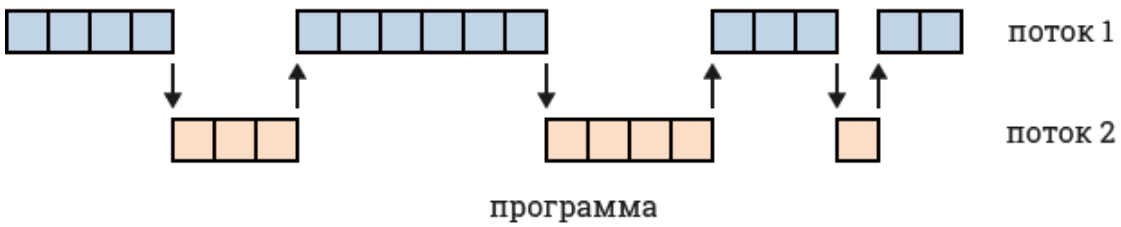


Рис. 3.7.

Так что же такое поток?

Поток можно представить как последовательность команд программы, которая претендует на использование процессора вычислительной системы для своего выполнения. Потоки одной и той же программы работают в общем адресном пространстве и, тем самым, разделяют (совместно используют) данные программы (см. рис. 3.8).

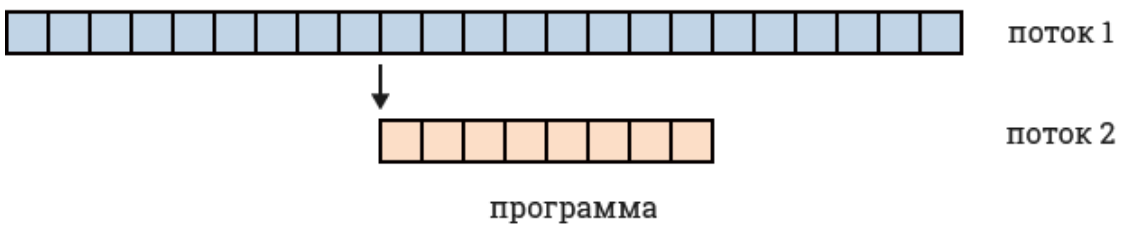


Рис. 3.8.

Разработка многопоточных программ — непростая задача. При планировании многопоточной программы оптимальным вариантом является разделение последовательности вычислений на потоки, число которых равно количеству процессоров. Если сделать много потоков, то это не ускорит выполнение программы, а затормозит ее, потому что переключение между потоками требует некоторого дополнительного времени.

Дополнительно разработка усложняется рисками нарушения целостности данных. Зачастую потоки нуждаются в одних и тех же ресурсах, при параллельном изменении этих ресурсов могут возникнуть ошибки. Например, два потока должны записать данные в файл, если они попытаются сделать это одновременно, то в итоге получим совершенно непредсказуемый результат. Чтобы этого не произошло, потоки нужно синхронизировать. Пока над записью в файл работает первый поток, второй поток должен ожидать своей очереди.

Одним из средств организации параллельных потоков в Android является класс AsyncTask. Его назначение состоит в том, чтобы выполнять задачи в фоновом потоке, параллельно с UI. Этот вариант идеально подходит для легковесных задач. Также данный класс имеет метод для обновления прогресс-индикатора выполнения задачи в потоке UI.

Рассмотрим некоторые методы класса AsyncTask (см. табл. 3.7).

Метод (public)	**Описание**
execute()	Метод, который нужно вызвать для активации потока
doInBackground ()	В этом обязательном методе описываются команды, которые нужно выполнить в фоновом потоке
onPreExecute ()	Метод запускается перед методом doInBackground()
onPostExecute ()	Метод запускается после метода doInBackground()

Табл. 3.7.

Ниже приведен небольшой пример — проект AsyncTask.3.4.1. В нем используется задержка времени в качестве имитации, к примеру, загрузки файла из интернета. При этом есть возможность запустить процесс «скачивания» как в потоке, так и без. Обратите внимание, что при нажатии кнопки «Начать не в потоке» процесс загрузки затормозит основной поток, в связи с чем экран устройства станет полностью некликабельным (в том числе даже не будет нажиматься CheckBox) и, скорее всего, выведется сообщение ANR (Application Not Responding). В случае нажатия кнопки «Начать в потоке» процесс загрузки начнется в асинхронном потоке и основной интерфейс не будет заморожен, в том числе CheckBox будет свободно нажиматься. Кроме того, на лайоуте располагается ProgressBar и TextView для вывода состояния асинхронной задачи:

```
 <?xml version="1.0" encoding="utf-8"?>
<LinearLayout 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:orientation="vertical"
    tools:context="ru.samsung.itschool.asynctask.MainActivity">

    <Button
        android:id="@+id/btJustDoIt"
        android:text="Начать не в потоке"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

    <Button
        android:id="@+id/btStart"
        android:text="Начать в потоке"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
    <ProgressBar
        style="?android:attr/progressBarStyleHorizontal"
        android:layout_width="fill_parent"
        android:layout_height="20pt"
        android:id="@+id/progressBar" />
    <TextView
        android:id="@+id/text"
        android:text="Выполнено : 0 / 100"
        android:layout_width="fill_parent"
        android:layout_height="20pt" />
    <CheckBox
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="проверка"/>

</LinearLayout>
```

Класс активности следующий:

```
 public class MainActivity extends AppCompatActivity {
    Button bStart, btJustDoIt;
    ProgressBar progressBar;
    TextView text;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        bStart = (Button) findViewById(R.id.btStart);
        btJustDoIt= (Button) findViewById(R.id.btJustDoIt);
        progressBar = (ProgressBar) findViewById(R.id.progressBar);
        text = (TextView) findViewById(R.id.text);
        // устанавливаем обработчик на кнопку "Начать в потоке"
        bStart.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                Toast.makeText(MainActivity.this, "Делаем операцию в потоке (10с)", Toast.L
ENGTH_SHORT).show();
                new LoadImage().execute();
            }
        });
        // устанавливаем обработчик на кнопку "Начать не в потоке"
        btJustDoIt.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                Toast.makeText(MainActivity.this, "Делаем операцию не в потоке (10с)", Toas
t.LENGTH_SHORT).show();
                try {
                    Thread.sleep(10000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });
    }
    // объект потока
    private class LoadImage extends AsyncTask<Void, Integer, Void> {
        @Override
        protected void onPreExecute() {
            super.onPreExecute();
        }
        protected Void doInBackground(Void... args) {
            for (int i = 0; i < 100; i += 10) {
                try {
                    Thread.sleep(1000);
                    publishProgress(i);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            publishProgress(100);
            return null;
        }
```

```
        protected void onPostExecute(Void image) {
            text.setText("Задача завершена");
        }
        @Override
        protected void onProgressUpdate(Integer... values) {
            super.onProgressUpdate(values);
            progressBar.setProgress(values[0]);
            text.setText("Выполнено : " + values[0] + "/100");
        }
    }
}
```

Метод OnProgressUpdate запускается в UI потоке после вызова метода publishProgress(). С помощью него можно обновлять данные о ходе выполнения задачи. К примеру, можно выводить их для пользователя на ProgressBar, как показано в примере.

Метод объекта AsyncTask может быть вызван только внутри UI потока. Если создать еще один объект задачи и выполнить для нее метод execute, во время выполнения первой будут работать две задачи, такого допускать нельзя.

При повороте экрана в приложении во время выполнения задачи AsyncTask будет создана новая задача, которая начнет выполняться заново, но старая задача также продолжит свое выполнение. Это произойдет из-за того, что Android при повороте экрана создает новое Activity и метод onCreate будет вызван заново.

3.4.2. Общий способ создания потоков в Java

Класс AsyncTask легок в использовании и служит для выполнения несложных операций в Android. Однако есть более универсальный способ создания потоков. Он применим не только в Android, но и в любых java-приложениях. Для этого в java предусмотрены класс Thread, а также интерфейс Runnable. Предлагается два варианта создания параллельной программы. Можно создать объект, наследуемый от класса Thread, переопределив в нем метод run(), или можно создать объекты, реализующие интерфейс Runnable.

Ознакомиться со всеми конструкторами и методами класса Thread можно на официальном сайте Oracle.

Ниже рассмотрен вариант реализации класса Thread — проект Thread.3.4.2:

```
public class MainActivity extends AppCompatActivity {
   String str="";
   TextView text1;

   @Override
   protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.activity_main);
       text1= (TextView) findViewById(R.id.text1);
       AnotherThread anotherThread=new AnotherThread();
       anotherThread.start();
   }

   class AnotherThread extends Thread {
       @Override
       public void run() {
           for (int i = 0; i < 10; i++) {
               try{
                   Thread.sleep(1000); //Приостанавливает поток на 1 секунду
               } catch(InterruptedException e){}
               str = str + "2";
           }
       }
   }

   public void refresh(View view) {
       text1.setText(str);
   }
}
```

В этом приложении при старте Android-программы в потоке начинают в строку str добавляться символы «2». Если пользователь будет нажимать на кнопку «обновить», он увидит на экране, как в потоке растет строка символов.

При реализации класса, наследуемого от Thread, теряется возможность наследования от других классов. Это связанно с тем, что в Java запрещено множественное наследование классов (через extends), то есть невозможна ситуация, когда класс является потомком нескольких родительских классов.

Второй вариант создания нового потока через реализацию интерфейса Runnable. Интерфейс Runnable содержит один единственный метод run(). Конструкторы класса Thread могут взаимодействовать с этим интерфейсом, один из конструкторов Thread(Runnable target). Ниже рассмотрен вариант с применением имплементации интерфейса Runnable — проект Runnable.3.4.3:

```
public class MainActivity extends AppCompatActivity {
   String str="";
   TextView text1;

   @Override
   protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.activity_main);
       text1= (TextView) findViewById(R.id.text1);
       Thread thread=new Thread(new AnotherRunnable());
       thread.start();
   }

   class AnotherRunnable implements Runnable {
       @Override
       public void run() {
           for (int i = 0; i < 10; i++) {
               try{
                   Thread.sleep(1000); //Приостанавливает поток на 1 секунду
               } catch(InterruptedException e){}
               str = str + "3";
           }
       }
   }

   public void refresh(View view) {
       text1.setText(str);
   }
}
```

Работает этот пример идентично предыдущему, то есть в строке str в потоке накапливаются символы, а изменения можно увидеть, нажав кнопку «обновить». Целевой класс потока должен реализовывать интерфейс Runnable. В конструктор Thread нужно передать объект этого класса. Вариант с использованием интерфейса Runnable лучше применять, когда не нужно наследовать от класса Thread ничего лишнего, а только реализовать метод run(). Также если создаваемый класс для потока должен наследоваться от другого класса, не являющимся потомком Thread, то использование Runnable — подходящий вариант.

В Android нельзя обращаться к экранным элементам (виджетам) не из главного потока активности. В связи с этим нельзя из дополнительного потока напрямую менять, например, TextView на лайоуте работающей программы. При попытке такого обращение произойдет исключение CalledFromWrongThreadException: Only the original thread that created a view hierarchy can touch its views. Для решения этой проблемы в Android предназначен специальный класс Handler, который позволяет управлять передачей сообщений между потоками. Таким образом, можно в потоке выполнять нужные операции и при необходимости передать сообщение в главный поток (поток активности/GUI) и оттуда уже изменить нужные виджеты.

Пример использование Handler можно посмотреть в примере проект Threads.3.4.4:

```
 // в главном потоке создаем handler
handler = new Handler() {   // создание хэндлера
            @Override
            public void handleMessage(Message msg) {
                super.handleMessage(msg);
                text.setText(text.getText().toString() + msg.what);
                text.invalidate();
            }
        };
```

```
 // в фоновом потоке отправляем сообщение handler, но ни в коем случае не пытаемся напрямую обращаться к виджетам
@Override
        public void run() {
            for (int i = 0; i < 20; i++) {
                try {
                    Thread.sleep(1000); //Приостанавливает поток на 1 секунду
                } catch (InterruptedException e) {
                }
                handler.sendEmptyMessage(1);  // отправка сообщения хендлеру
            }
        }
```

3.4.3. Реализация логики потока

Итак, при создании нового потока мы переопределяем метод run(). В нем находится код, который нужно выполнить в новом потоке. Поток начинает свою работу с началом этого метода и заканчивает свое выполнение по окончанию метода. В этом методе должна быть реализована логика потока.

Метод start() запускает новый поток и сам запустит метод run(). В таком случае мы получим асинхронные потоки, это значит, что мы не знаем заранее, в какой последовательности будут выполняться потоки. Не следует пытаться вызвать метод run() для объекта самостоятельно.

Ниже приведен пример работы трех асинхронных потоков, которые работают различное время, — проект LogicThreads.3.4.5.

```
class Work extends Thread {
   String threadName;

   public Work(String name) {
       threadName = name;
   }

   @Override
   public void run() {
       int max=(int) (20*Math.random());
       for (int i = 0; i < max; i++) {
           try {
               sleep((long)(1000*Math.random()));
               System.out.print(threadName);
           } catch (InterruptedException e) {
               e.printStackTrace();
           }
       }
   }
}
```

Запускаем три потока с именами A, B и C:

```
public class ParalleLogic {
   public static void main(String[] args) {
       Work work1 = new Work("A");
       Work work2 = new Work("B");
       Work work3 = new Work("C");
       work1.start();
       work2.start();
       work3.start();
   }
}
```

На экран будет выведено:

```
ABBAACBACCCACBACACCCCCC
```

Как видно, программа работает до завершения всех потоков, и они работают абсолютно независимо друг от друга, timeline (рис. 3.9).

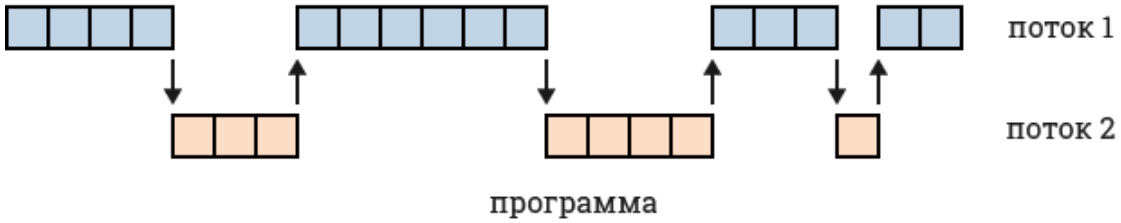
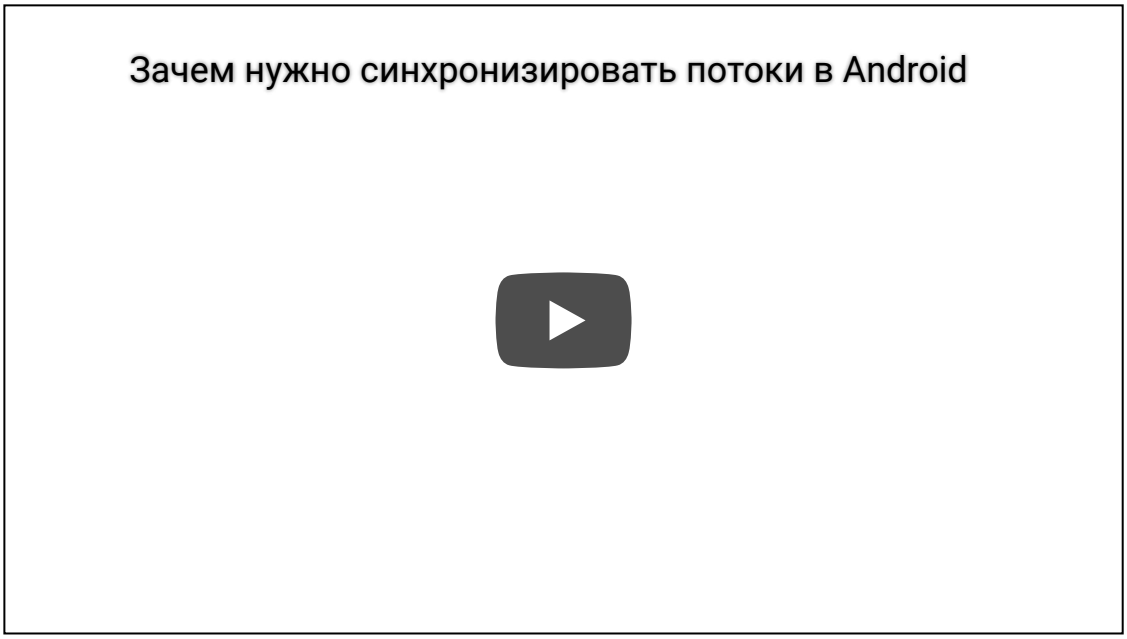


Рис. 3.9.

Тезисы о потоках

- Описывая поток, необходимо переопределить метод run(), в котором записывается код, выполняющий то, что нужно делать асинхронно.
- Когда поток создан при помощи оператора new, нужно запустить его на исполнение, вызвав из него метод start().
- Закончится исполнение запущенного потока, когда закончится исполнение переопределенного метода run().
- Запускать можно любое количество потоков.
- Выполнение программы закончится, когда закончится выполнение всех работающих в ней потоков.

3.4.4. Синхронизация потоков



Использование асинхронных потоков может привести к ошибочному выполнению.

Рассмотрим пример. Есть общий баланс money = 100 и два пользователя, которые захотели снять определенное количество средств с баланса amount1 = 80 и amount2 = 90. Существует функция для проведения этой операции, в которой указано, что если пользователь пытается снять сумму, которая больше баланса, то операция выполнена не будет. При использовании асинхронных потоков возможна такая ситуация, что проверка на возможность снятия средств с баланса пройдет одновременно и при текущем значении баланса в обоих случаях провести операцию снятия будет разрешено. В итоге мы можем получить непредсказуемый результат, баланс может уйти в минус, хотя разработчик был уверен, что обезопасил себя от этого (см. табл. 3.8).

Шаги	Процесс 1	Процесс 2	Переменная
0	money – amount1 > 0		money == 100
1		money – amount2 > 0	money == 100
2	money -= amount1		money == 20
3		money -= amount2	money == -70

Табл. 3.8.

При попытке двух асинхронных потоков одновременной записи данных в поле (переменную) какого-либо объекта возможно нарушение логической целостности данных. Такую ситуацию называют еще Data races (гонки за данными).

Для разрешения таких проблем в Java используется синхронизация. Механизм синхронизации основывается на концепции монитора.

Монитор — это специальный механизм в Java, обеспечивающий управление взаимодействием процессов и их состоянием.

Монитор можно представить себе как ключ от комнаты сейфовых ячеек в банке, а потоки можно представить как клиентов в банке. Когда первый клиент получил ключ от комнаты, дверь за ним закрывается, и все остальные желающие работать с сейфами ждут, пока первый клиент не вернет ключ. Это называется «поток захватил монитор». Когда первый клиент вышел, ключ (монитор) может быть передан следующему клиенту. Это называется «поток освободил монитор».

Монитор встроен в объект класса Object. А поскольку в Java все классы наследуются от Object, то у каждого объекта в Java имеется свой собственный неявный монитор.

Для того чтобы включить режим монопольного доступа к данным из одного потока, нужно использовать ключевое слово synchronized (синхронно). Этот модификатор может быть применен:

- к методу;
- к его части (блоку).

Когда первый поток начинает выполнять код, отмеченный synchronized, он захватывает монитор объекта синхронизации. И если любой другой поток попытается тоже исполнять synchronized код (на этом же мониторе), то он будет приостановлен до тех пор, пока первый поток не выйдет из synchronized блока кода. Таким образом, монитор Java помогает при необходимости вводить поочередность в параллельную обработку.

Объявление метода synchronized не подразумевает, что только один поток может одновременно выполнять этот метод. Имеется в виду, что в любой момент времени только один поток может вызвать этот или любой другой synchronized-метод для конкретного объекта. Таким образом, мониторы Java связаны с объектами, но не с блоками кода. Два потока могут параллельно выполнять один и тот же метод типа synchronized при условии, что этот метод вызван для разных объектов.

В следующем примере рассмотрена синхронизация внутри блока. В такой конструкции следует указывать объект, одновременный доступ к которому должен быть заблокирован. Проект Sync.3.4.6

```
 class Client extends Thread {
    Account account;

    public Client(Account acc) {
        account = acc;
    }

    @Override
    public void run() {
//      synchronized (account) {
            if (account.money - 70 > 0) { // достаточно ли средств ?
                try {
                    sleep((long) (1000 * Math.random()));  // имитируем задержку банкомата
                } catch (InterruptedException e) {}
                account.money -= 70;  // снимаем со счета нужную сумму
                System.out.println(account.money);
            }
            else System.out.println("there are not enough funds on your account");
//      }
    }
}
```

Теперь имитируем одновременную попытку снятия денег:

```
 public class SynchronizedLogic {
    public static Account account;

    class Account{
        public double money;
    }

    public static void main(String[] args) {
        account=new SynchronizedLogic().new Account();
        account.money=100.;
        Client client1=new Client(account);
        Client client2=new Client(account);
        client1.start();
        client2.start();
    }

}
```

При выполнении этого примера на экран выведется:

```
30.0
-40.0
```

Что демонстрирует классический Data Races. Однако если раскомментировать конструкцию synchronized (account), которая принуждает потоки выполняться последовательно на участке проверки и уменьшения счета, то в этом случае вывод будет совершенно правильный с точки зрения логики работы банка:

```
30.0
there are not enough funds on your account
```

Применять блочную синхронизацию удобно, когда нужно синхронизировать не весь метод целиком. Например, вам нужно обеспечить чтение/изменение массива из разных потоков. Также если разработчик не имеет доступа к какому-либо методу, потому что не он его разработал. Тогда можно поместить вызов такого метода внутрь блока с синхронизацией.

3.4.5. Управление потоками

У объектов также существуют несколько методов для организации работы потоков (см. табл. 3.9). Данные методы можно применять только внутри блока synchronized к блокирующемуся объекту.

Метод	Назначение
wait()	Вызывается внутри синхронизированного блока или метода, останавливает выполнение текущего потока и высвобождает захваченный им объект
notify()	Метод также может быть вызван только внутри синхронизированного блока. Возвращает блокировку объекта потоку, из которого был вызван метод wait(). Если их несколько, то поток выбирается случайно
notifyAll()	Пробуждает все потоки, из которых был запущен метод wait() для данного объекта

Табл. 3.9.

Также существует метод join(), применяемый к потоку. Этот метод ожидает завершение потока, для которого он вызван. В параметрах метода join() можно указать максимальное время ожидания.

Метод wait останавливает выполнение текущего потока. Пробудиться остановленный поток может после вызова notify или notifyAll где-то в другом месте, но на этом же мониторе. Если указанные методы будут использованы не в synchronized блоках, то будет выброшено исключение IllegalMonitorStateException. Ниже пример управления потоками — проект ThreadControl.3.4.7. В методе eatPizza() произойдет остановка текущего потока (на мониторе — объект myHouse).

```
public class MyHouse {
   private boolean pizzaArrived = false;

   public void eatPizza() {
       synchronized (this) {
           while (!pizzaArrived) {
               try {
                   wait();
               } catch (InterruptedException e) {
               }
           }
       }
       System.out.println("yumyum...");
   }

   public void pizzaGuy() {
       synchronized (this) {
           pizzaArrived = true;
           notifyAll();
       }
   }
}
```

По прошествии трех секунд в другом потоке будет вызван метод pizzaGuy(), в нем выполнится метод notify, который разбудит ранее остановленный поток и на экран выведется «yumyum…».

```
public class Test {

    MyHouse myHouse;

    class MyThread extends Thread {
        @Override
        public void run() {
            super.run();
            myHouse.eatPizza();
        }
    }

    public static void main(String[] args) {
        Test test = new Test();
        test.myHouse = new MyHouse();
        MyThread thread = test.new MyThread();
        thread.start();
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        test.myHouse.pizzaGuy();
    }

}
```

3.4.6. Блокировки

Существует понятие мертвой (взаимной) блокировки (deadlock). Ситуация, когда потоки находятся в бесконечном ожидании ресурсов. Рассмотрим причину ее возникновения. Например, есть два потока. И первому, и второму потоку нужны объекты, *a* и *b*. Возможно такое, что первый поток сперва заблокирует, к примеру, объект *a*, в это время второй поток заблокирует объект *b*. В итоге первый поток не сможет начать выполнение, пока не получит объект *b*, а второй поток уже не сможет его освободить, так как ничего не может сделать, пока не получит объект *a*. Таким образом, возникнет состояние взаимной блокировки (deadlock). Потоки будут ждать нужных ресурсов бесконечно (см. табл. 3.10).

Шаг	**Поток 1**	**Поток 2**
0	Блокировка объекта a	Блокировка объекта b
1	Ожидание объекта b	Ожидание объекта a
2	deadlock	deadlock

Табл. 3.10.

3.5. Сервисы в Android. Типы сенсоров

Сайт: IT Академия SAMSUNG
Курс: MDev @ IT Академия Samsung
Книга: 3.5. Сервисы в Android. Типы сенсоров
Напечатано:: Егор Беляев
Дата: Суббота, 18 Апрель 2020, 19:24

Оглавление

3.5.1. Сервисы в Android

Сервис в Android — это компонент приложения, который может выполнять длительные операции в фоновом режиме и не содержит пользовательского интерфейса (UI).

Сервисы (службы) в Android представляют собой фоновые процессы. Они не имеют интерфейса пользователя (UI) и работают без его вмешательства. Службы являются компонентами приложения, но способны выполнять действия, даже когда оно невидимо, пассивно или вовсе закрыто.

В отличие от активностей, сервисы предназначены для длительного существования и обладают более высоким приоритетом, чем скрытые активности. Тем самым, вероятность закрытия сервиса системой при нехватке ресурсов намного ниже. К тому же, служба может быть настроена таким образом, что будет автоматически запущена, как только найдутся необходимые ресурсы.

Сегодня, мы можем встретить массу приложений, использующих сервисы. Работая в фоновом режиме, службы имеют возможность выполнять сетевые запросы, вызывать уведомления, обрабатывать информацию, проигрывать музыку и выполнять множество иных «закулисных» задач.

Многие компоненты приложения могут запускать сервисы, обмениваться данными и закрывать их. В роли таких компонентов могут выступать как активности, так и другие сервисы. Стоит подчеркнуть, что сервис может быть закрыт как компонентом приложения, так и самостоятельно инициировать собственное завершение.

Обратите внимание, что, несмотря на работу в фоновом режиме, сервисы работают в основном потоке приложения (main UI thread). В случае выполнения «тяжелых» задач их необходимо запускать в отдельном потоке, так как это может привести к «неприятным» задержкам в UI пользователя.

3.5.2. Жизненный цикл сервиса

Жизненный цикл сервиса, на первый взгляд, более прост, чем у активностей или фрагментов (см. рис. 3.10).

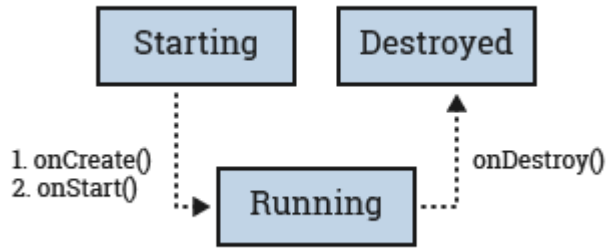


Рис. 3.10.

Все сервисы наследуются от класса Service и проходят следующие этапы жизненного цикла:

- onCreate() — вызывается при создании сервиса;
- onStartCommand() — вызывается при получении сервисом команды, отправленной с помощью метода startService();
- onBind() — вызывается при закреплении клиента за сервисом с помощью метода bindService();
- onDestroy() — вызывается при завершении работы сервиса.

На самом деле, сервис имеет два вложенных цикла во время своей жизни и два различных варианта поведения.

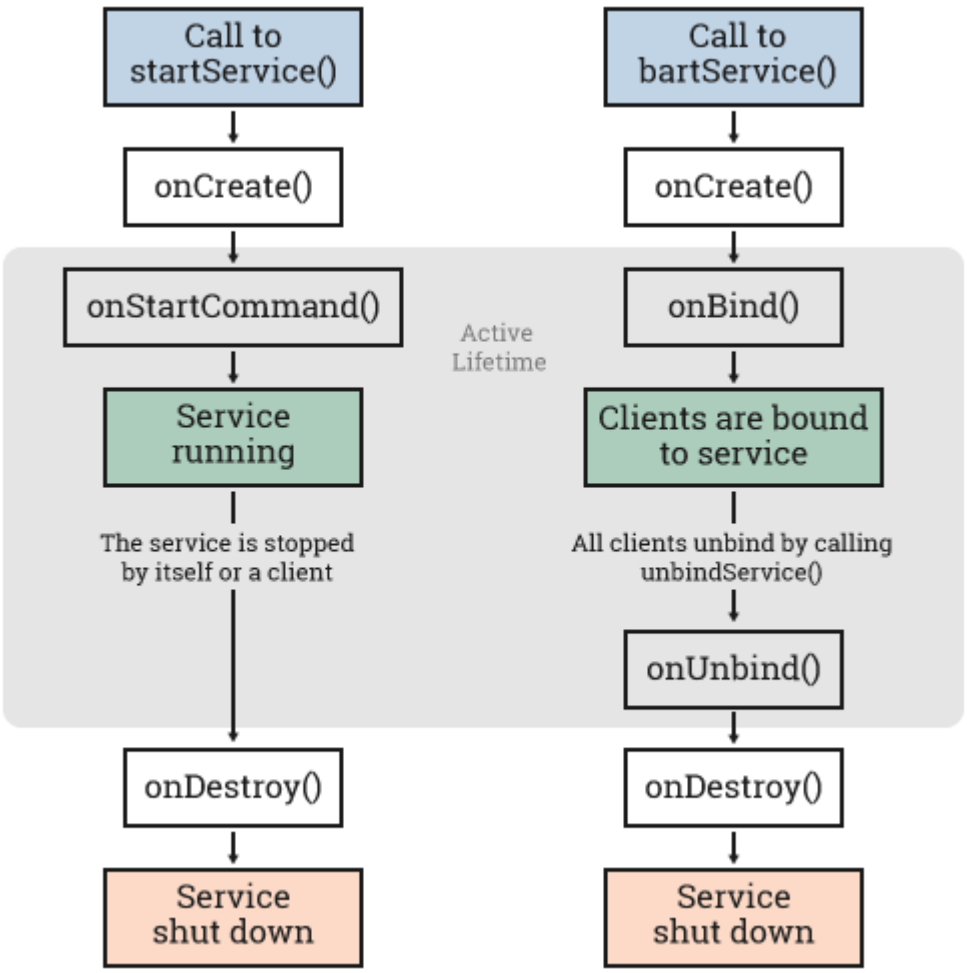


Рис. 3.11.

Сервисы могут работать в двух режимах:

- ***Запущенный (started).*** Сервис, запущенный с помощью метода startService(). Будучи запущенным, сервис может работать неопределенно долгое время, но чаще служба запускается для выполнение разовой операции, например, загрузки информации по сети. Когда необходимые действия выполнены, сервис должен самостоятельно завершить работу.
- ***Привязанный (bound).*** Сервис, к которому другой компонент приложения привязался с помощью метода bindService(). Привязанный сервис предоставляет клиент-серверный интерфейс, через который с ним можно взаимодействовать: посылать запросы и получать ответы. Привязанный сервис уничтожается, когда от него отвязывается последний привязанный клиент.

В реальных приложениях службы могут работать и в «смешанном» режиме: быть запущенными с помощью startService(), чтобы работать неограниченно долго, в то же время разрешая «привязку» (binding) для взаимодействия с клиентами.

Можно установить подключение к работающей службе и использовать это подключение для взаимодействия со службой. Подключение устанавливают вызовом метода Context.bindService() и закрывают вызовом Context.unbindService(). Если служба уже была остановлена, вызов метода bindService() может ее запустить.

Методы onCreate() и onDestroy() вызываются для всех сервисов независимо от того, запускаются ли они через Context.startService() или Context.bindService().

Если служба разрешает другим приложениям связываться с собой, то привязка осуществляется с помощью дополнительных методов обратного вызова:

- IBinder onBind(Intent intent)
- onUnbind(Intent intent)
- onRebind(Intent intent)

В метод onBind() передают объект класса Intent, который был параметром в методе bindService(), а в метод обратного вызова onUnbind() — объект класса Intent, который передавали в метод unbindService(). Если сервис разрешает связывание, то onBind() возвращает канал связи, который используют клиенты, чтобы взаимодействовать с сервисом. Метод обратного вызова onRebind() может быть вызван после onUnbind(), если новый клиент соединяется со службой.

3.5.3. Создание сервисов

Создание класса сервиса

Для создания сервиса требуется создать дочерний класс для класса android.app. Service и переопределить несколько методов, которые будут, в итоге, определять характеристики создаваемого сервиса.

```
public class MyService extends Service {

        @Override
        public IBinder onBind(Intent intent) {
                return null;
        }

}
```

Наиболее часто так же переопределяют следующие методы:

- onStartCommand() — вызывается, когда сервис запускается с помощью метода startService(). Такой сервис может существовать бесконечно долгое время, и его необходимо останавливать с помощью методов stopSelf() или stopService(), когда данный сервис больше не нужен. Если сервис должен работать только в режиме «привязанного» сервиса, этот метод переопределять не нужно;
- onBind() — вызывается, когда служба запускается с помощью метода bindService(). Этот метод нужно реализовывать всегда, но возвращаемое значение зависит от режима работы — при создании привязанного сервиса требуется возвратить объект типа IBinder, в противном случае — просто null;
- onCreate() — вызывается при создании сервиса, перед вызовом onStartCommand() и onBind(). Если сервис уже существует, метод onCreate() не вызывается;
- onDestroy() — вызывается перед уничтожением сервиса.

Описание сервисов в манифесте приложения

Как и активности, сервисы должны быть продекларированы и описаны в манифесте приложения, для этого используется тег, являющийся дочерним тегом для:

```
<service android:name=".MyService"/>
```

Стоит подчеркнуть, что тег <service> может включать целый набор атрибутов. Например, такой как enabled, дает возможность другим приложениям получать доступ к вашему сервису. Подробнее со списком атрибутов и правилом их использования можно ознакомиться по ссылке.

При использовании Android Studio для создания сервисов можно использовать пункты меню *New -> Service -> Service.* Это позволит автоматически создать класс наследник Service и задекларировать его в манифесте приложения.

Кроме создания собственных сервисов, можно использовать системные сервисы. Приведем некоторые из них:

- **Account Service** — сервис для управления пользовательскими учетными записями;
- **Activity Service** — сервис для управления активностями;
- **Alarm Service** — сервис для отправки разовых или периодических оповещений в заданное время;
- **Bluetooth Service** — сервис для Bluetooth;
- **Clipboard Service** — сервис для управления буфером обмена;
- **Connectivity Service** — сервис для управления сетевыми соединениями;
- **Download Service** — сервис для управления загрузками;
- **Input Method Service** — сервис для управления текстовым вводом;
- **JobScheduler** — сервис для планирования задач;
- **Location Service** — сервис для отслеживания координат;
- **Layout Inflater Service** — сервис для управления компоновкой экрана при динамическом создании из кода;
- **NFC Service** — сервис для управления NFC;
- **Notification Service** — сервис для управления уведомлениями;
- **Power Service** — сервис для управления энергопотреблением;
- **Search Service** — сервис для управления глобальным поиском;
- **Sensor Service** — сервис для доступа к датчикам;
- **Telephony Service** — сервис для управления телефонными функциями;
- **Vibrator Service** — сервис для доступа к виброзвонку;
- **Wallpaper Service** — сервис для управления обоями на домашнем экране;
- **Wifi Service** — служба для управления соединениями Wi-Fi.

3.5.4. Запуск и остановка сервисов

Запуск сервиса

Запуск сервиса можно осуществить из активности или любого другого компонента приложения путем передачи в Intent, указав сервис для запуска. Чтобы запустить сервис, в приложении необходимо вызывать метод startService(). На практике существует два способа запуска сервиса:

- явный вызов;
- неявный вызов.

Ниже приведен пример для явного вызова службы с именем MyService:

```
 Intent intent = new Intent(this, MyService.class);
startService(intent);
```

Кроме того, можно явно определить сервис, создав экземпляр класса этого сервиса. Приведем пример неявного вызова сервиса:

```
 startService(new Intent(MyService.SERVICE_ACTION));
```

Чтобы использовать неявный вызов, необходимо включить константу SERVICE_ACTION в класс MyService. Она служит идентификатором для сервиса. Она может быть проинициализирована, например, следующим образом:

```
 private static String SERVICE_ACTION = " ru.samsung.itschool.book.media.PLAYER";
```

При этом также необходимо использовать фильтр намерений, чтобы зарегистрировать его как провайдера SERVICE_ACTION. Если сервис потребует разрешений, которые не имеет ваше приложение, будет выброшено исключение SecurityException.

Как только сервис завершил выполнение тех действий, для которых он запускался, вы должны вызвать метод stopSelf() либо без передачи параметра, чтобы ускорить остановку работы, либо передав значение startId, чтобы убедиться, что задачи выполнены для всех экземпляров, запущенных с помощью вызова startService().

В обоих случаях система Android вызывает метод службы onStartCommand() и передает ей Intent. Не стоит вызывать метод onStartCommand() напрямую.

Остановка сервисов

Запущенный сервис должен обслуживать свой жизненный цикл, то есть система Android не останавливает и не удаляет сервис за исключением ситуаций, требующих освобождения памяти системы, и сервис продолжит свою работу даже после завершения метода onStartCommand(). Таким образом, сервис должен останавливать сам себя. Остановка происходит при помощи вызова метода stopSelf(). Кроме того, сервис может быть остановлен из другого компонента приложения вызовом метода stopService().

Как только поступил запрос на остановку сервиса одним из вышеописанных способов, система уничтожит службу так быстро, как это только возможно. Очень важно, чтобы приложение останавливало свои сервисы по окончанию их работы, чтобы избежать потери ресурсов системы, и сохранить энергию батареи.

Управление перезагрузкой сервисов

Во многих случаях при работе с сервисами необходимо переопределить метод onStartCommand(). Как уже было сказано ранее, он вызывается каждый раз, когда сервис начинает работать с помощью метода startService(). Таким образом сервис может быть выполнен несколько раз на протяжении работы приложения, и необходимо убедиться, что ваш сервис предусматривает эту возможность.

Следует обратить внимание, что метод onStartCommand() возвращает целое число, которое описывает, как система должна продолжить работу сервиса, когда его необходимо остановить. Возвращаемое значение метода onStartCommand() может быть одной из следующих констант:

- **START_STICKY** — описывает стандартное поведение. Если вы вернете это значение, обработчик onStartCommand() будет вызываться при повторном запуске сервиса после преждевременного завершения работы. Обратите внимание, что аргумент Intent, передаваемый в onStartCommand(), получит значение null. Данный режим обычно используется для сервисов, которые сами обрабатывают свои состояния, явно стартуя и завершая свою работу при необходимости (с помощью методов startService() и stopService()). Это относится к сервисам, которые проигрывают музыку или выполняют другие задачи в фоновом режиме;
- **START_NOT_STICKY** — этот режим используется в сервисах, которые запускаются для выполнения конкретных действий или команд. Как правило, такие сервисы используют stopSelf() для прекращения работы, как только команда выполнена. После преждевременного прекращения работы сервисы, работающие в данном режиме, повторно запускаются только в том случае, если получат вызовы. Если с момента завершения работы сервиса не был запущен метод startService(), он остановится без вызова обработчика onStartCommand(). Данный режим идеально подходит для сервисов, которые обрабатывают конкретные запросы, особенно это касается регулярного выполнения заданных действий (например, обновления или сетевые запросы). Вместо того, чтобы перезапускать сервис при нехватке ресурсов, часто более целесообразно позволить ему остановиться и повторить попытку запуска по прошествии запланированного интервала;
- **START_REDELIVER_INTENT** — в некоторых случаях нужно убедиться, что команды, которые вы посылаете сервису, выполнены. Этот режим — комбинация предыдущих двух. Если система преждевременно завершила работу сервиса, он запустится повторно, но только когда будет сделан явный запрос на запуск или если процесс завершился до вызова метода stopSelf(). В последнем случае вызовется обработчик onStartCommand(), он получит первоначальное намерение, обработка которого не завершилась должным образом.

Такой подход позволяет методу onStartCommand() быстро завершить работу и дает возможность контролировать поведение сервиса при его повторном запуске.

Пример 3.3*

Реализуем приложение, показывающее погоду в вашем городе.

Первое, что необходимо для решения этой задачи, — это источник данных о погоде в вашем городе. Подобную информацию можно получить в интернете, но представлена она там в виде структуры, понятной пользователю, а не программе. Для удобства работы целесообразно использовать API любого погодного сервиса для получения информации в виде XML или JSON. Для данной задачи вполне подойдет данный сервис. Там же на сайте есть информация о формате выдаваемого JSON. Прежде чем приступить к написанию самого приложения, выберите свой город и скопируйте ссылку в удобное для вас место (см. рис. 3.12). Стоит подчеркнуть, что при желании есть возможность использовать любой другой сервис или вовсе парсить HTML-страницу, но, как правило, все это более трудоемкий процесс.

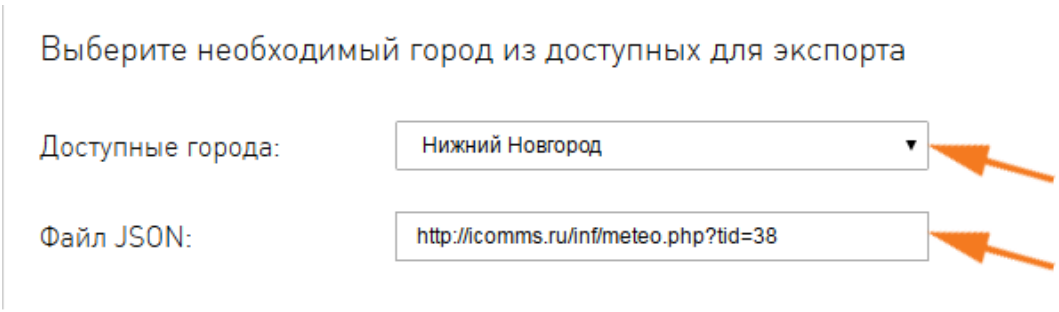


Рис. 3.12.

Далее создадим проект с одной активностью. Первоначально, просто создадим сервис и покажем полученные о ссылке данные. В качестве разметки установим TextView с возможностью прокрутки, так как первоначальный набор данных будет достаточно большим (прогноз на 2 недели вперед).

Файл разметки activity_main.xml может выглядеть следующим образом:

```
 <?xml version="1.0" encoding="utf-8"?>
<LinearLayout 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:gravity="center"
    android:orientation="vertical"
    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="ru.samsung.itschool.ebook.serviceexample.MainActivity">

    <ScrollView
        android:id="@+id/SCROLLER_ID"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:fillViewport="true"
        android:scrollbars="vertical">

        <TextView
            android:id="@+id/temp_txt"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:gravity="center" />

    </ScrollView>

</LinearLayout>
```

При написании сервиса нам понадобиться вспомнить, что для загрузки данных из интернета мы можем использовать класс URL. Также не стоит забывать, что сегодня Android запрещает работу с сетью в основном потоке, а работа с интернетом требует добавления разрешения в манифест приложения.

```
 <uses-permission android:name="android.permission.INTERNET" />
```

Создадим сервис.

```
 package ru.samsung.itschool.ebook.serviceexample;

import android.app.Service;
import android.content.Intent;
import android.os.AsyncTask;
import android.os.IBinder;
import android.widget.Toast;
import java.io.InputStream;
import java.net.URL;
import java.util.Scanner;

public class GisService extends Service {

        public static final String CHANNEL = "GIS_SERVICE";
        public static final String INFO = "INFO";

        @Override
        public void onCreate() {
                // сообщение о создании службы
                Toast.makeText(this, "Service created", Toast.LENGTH_SHORT).show();
        }

        @Override
        public int onStartCommand(Intent intent, int flags, int startId) {
                // сообщение о запуске службы
                Toast.makeText(this, "Service started", Toast.LENGTH_SHORT).show();

                // создаем объект нашего AsyncTask (необходимо для работы с сетью)
                GisAsyncTask t = new GisAsyncTask();
                t.execute(); // запускаем AsyncTask

                return START_NOT_STICKY;
        }

        @Override
        public void onDestroy() {
                //сообщение об остановке службы
                Toast.makeText(this, "Service stoped", Toast.LENGTH_SHORT).show();
        }

        @Override
        public IBinder onBind(Intent intent) {
                return null;
        }

        //поток работы с сетью
        private class GisAsyncTask extends AsyncTask<Void, Void, String> {

                @Override
                protected void onPostExecute(String aVoid) {
                        Intent i = new Intent(CHANNEL); // интент для отправки ответа
                        i.putExtra(INFO, aVoid); // добавляем в интент данные
                        sendBroadcast(i); // рассылаем
```

```
                }

                @Override
                protected String doInBackground(Void... voids) {
                        String result;
                        try {
                                //загружаем данные
                                URL url = new URL("http://icomms.ru/inf/meteo.php?tid=38");

                                // ”оборачиваем” для удобства чтения
                                Scanner in = new Scanner((InputStream) url.getContent());

                                // читаем и добавляем имя JSON массива
                                result = "{\"gis\":" + in.nextLine() + "}";
                        } catch (Exception e) {
                                result = e.toString();
                        }
                        return result;
                }
        }
}
```

Зарегистрируем наш сервис в манифесте.

```
 <service android:name=".GisService"
            android:enabled="true"
            android:exported="true"/>
```

Теперь напишем код для главной активности приложения. Для начала создадим объект класса BroadcastReceiver. Объект этого класса представляет собой приемник широковещательных сообщений. Приемник широковещательных сообщений — — это компонент приложения для получения внешних событий и реакции на них. В данном примере мы будем получать сообщения от нашего сервиса.

В объекте класса BroadcastReceiver необходимо реализовать метод onRecive(). Когда широковещательное сообщение прибывает к получателю, Android вызывает его методом onReceive() и передает в него объект типа Intent, содержащий сообщение. Данный метод может быть реализован следующим образом:

```
 protected BroadcastReceiver receiver = new BroadcastReceiver() {
        @Override
        public void onReceive(Context context, Intent intent) {
                try {
                        JSONObject json = new JSONObject(intent.getStringExtra(GisService.I
NFO)); //получаем JSON из intent-а
                        JSONArray gisArray = json.getJSONArray("gis"); //получаем JSON-масс
ив
                        tempText.setText(gisArray.toString()); //выводим  JSON-массив в тек
стовое поле
            } catch (JSONException e) {
                        Toast.makeText(MainActivity.this, "Wrong JSON format", Toast.LENGTH
_LONG).show();
            }
        }
}
```

В методе onCreate() регистрируем receiver с помощью метода registerReceiver и запускаем наш сервис, а в методе onDestroy() останавливаем сервис. Полностью код класса MainActivity будет выглядеть следующим образом:

```
 package ru.samsung.itschool.ebook.serviceexample;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.widget.TextView;
import android.widget.Toast;

import com.example.myapplication.R;

import org.json.JSONArray;
import org.json.JSONException;
import org.json.JSONObject;

public class MainActivity extends AppCompatActivity {

        private TextView tempText;

        @Override
        protected void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                setContentView(R.layout.activity_main);
                tempText = (TextView) findViewById(R.id.temp_txt);
                registerReceiver(receiver, new IntentFilter(GisService.CHANNEL));
                Intent intent = new Intent(this, GisService.class);
                startService(intent);
        }

        @Override
        protected void onDestroy() {
                super.onDestroy();
                Intent intent = new Intent(this, GisService.class);
                stopService(intent);
        }


        protected BroadcastReceiver receiver = new BroadcastReceiver() {
                @Override
                public void onReceive(Context context, Intent intent) {
                        try {
                                JSONObject json = new JSONObject(intent.getStringExtra(GisS
ervice.INFO));
                                JSONArray gisArray = json.getJSONArray("gis");
                                tempText.setText(gisArray.toString());
                        } catch (JSONException e) {
                                Toast.makeText(MainActivity.this, "Wring JSON format", Toas
t.LENGTH_LONG).show();
                        }
                }
```

```
        };
}
```

В целом, все происходящие внутри главной активности не должно вызвать у вас вопросов, за исключением, может быть, одного. Для обработки полученных результатов здесь были использованы стандартные классы для работы с JSON. Для подробного ознакомления можно воспользоваться ссылкой.

После запуска приложения получим следующий результат (рис. 3.13).

Рис. 3.13.

В результате данное приложение позволяет нам пролистывать JSON-массив с данным погоды в выбранном вами городе на две недели. Конечно, такой формат представления данных неприемлем для конечного пользователя. Следовательно, для завершения данного приложения необходимо «распарсить» уже полученный JSON-массив и представить все данные в понятной пользователю структуре. Для работы с JSON используйте функции, описанные по ссылке.

3.5.5. IntentService

Класс IntentService является наследником класса Service. Его использование целесообразно, если надо выполнять какие-то тяжелые задачи с намерениями, которые могут выполняться асинхронно. Принцип работы этого вида сервиса достаточно прост. Он сам создает новый рабочий поток, затем следит за всеми переданными намерениями и отправляет их на обработку в этот поток. Далее в коде необходимо определить, как обработать Intent. При этом несомненным плюсом такого подхода является тот факт, что вам не нужно запускать AsyncTask и управлять тяжелыми задачами, сервис сам со всем справится. Вы можете отправить данные обратно в приложение через широковещательное сообщение и принять сообщение через широковещательный приемник.

Иными словами, приложение посылает в сервис вызовы метода startService(), в которых передает намерения. IntentService принимает эти вызовы в методе onStartCommand(), берет последовательно намерения и направляет их в очередь на обработку. Далее они поочередно обрабатываются в отдельном потоке с помощью метода onHandleIntent(). Когда последний Intent из очереди обработан, сервис сам завершит работу. Во многих случаях использование IntentService проще, чем AsyncTask или Thread.

Приведем небольшой пример. Пусть в приложении происходит три запуска сервиса через метод startService() (например, в главной активности).

```
 Intent intent = new Intent(this, MyService.class);
startService(intent.putExtra("time", 1).putExtra("label", "IT") );
startService(intent.putExtra("time", 2).putExtra("label", "Samsung") );
startService(intent.putExtra("time", 3).putExtra("label", "School") );
```

Переменная time — это время паузы, которую будем делать в сервисе, label — метка, для того чтобы отличать вызовы сервиса. Далее создадим класс MyService и сделаем его наследником класса IntentService.

```
 public class MyService extends MyService{

        final String LOG_TAG = "IntentServiceLogs";

        public MyService() {
                super("name");
        }

        public void onCreate() {
                super.onCreate();
                Log.d(LOG_TAG, "Intent service created");
        }

        @Override
        protected void onHandleIntent(Intent intent) {
                int time = intent.getIntExtra("time", 0);
                String label = intent.getStringExtra("label");

                Log.d(LOG_TAG, "onHandleIntent start " + label);
                try {
                        TimeUnit.SECONDS.sleep(time);
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
                Log.d(LOG_TAG, "onHandleIntent end " + label);
        }

        public void onDestroy() {
                super.onDestroy();
                Log.d(LOG_TAG, "Intent service destroyed");
        }
}
```

В классе нужно определить конструктор. В нем вызовем конструктор суперкласса и укажем строковое имя, которое будет использовано для инициализации потока.

В методе onHandleIntent() обрабатываются намерения. Получаем из них переменные time и label, запускаем паузу на time секунд и выводим в логи значения label в начале и конце метода.

Если запустить приложение, то в логах можно будет увидеть примерно следующее:

```
12:14:37.880: D/IntentServiceLogs(4137): Intent service created
12:14:37.880: D/IntentServiceLogs(4137): onHandleIntent start IT
12:14:40.880: D/IntentServiceLogs(4137): onHandleIntent end IT
12:14:40.880: D/IntentServiceLogs(4137): onHandleIntent start Samsung
12:14:41.880: D/IntentServiceLogs(4137): onHandleIntent end Samsung
12:14:41.880: D/IntentServiceLogs(4137): onHandleIntent start School
12:14:45.890: D/IntentServiceLogs(4137): onHandleIntent end School
12:14:45.890: D/IntentServiceLogs(4137): Intent service destroyed
```

Сервис создался, необходимые вызовы выполнились по очереди и сервис завершил свою работу.

3.5.6. Сенсоры и их типы

Android-устройства имеют множество сенсоров, к которым программист может получить доступ, причем с появлением новых моделей их разнообразие постоянно растет. Android поддерживает несколько типов сенсоров:

- акселерометр;
- барометр;
- гироскоп;
- датчик освещения;
- датчик магнитных полей;
- датчик поднесения телефона к голове;
- датчик температуры аппарата;
- датчик температуры окружающей среды;
- измеритель относительной влажности и др.

Конкретный набор сенсоров зависит от Android-устройства, но в большинстве смартфонов и планшетов присутствуют самые основные — акселерометр и гироскоп. Для обработки событий, связанными с датчиками, в классе активности нужно реализовать интерфейс SensorEventListener:

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

В активности должны появиться следующие методы:

```
 @Override
 // Изменение точности показаний датчика
public void onAccuracyChanged(Sensor sensor, int accuracy) {
}

@Override
protected void onResume() {
}

@Override
protected void onPause() {
}

@Override
// Изменение показаний датчиков
public void onSensorChanged(SensorEvent event) {
}
```

В методе OnCreate получим объект типа SensorManager — менеджер сенсоров устройства. С помощью этого объекта можно получить доступ к нужному сенсору. Далее показан фрагмент класса активности с получением сенсора — акселерометра:

```
 public class MainActivity extends AppCompatActivity implements SensorEventListener {
        private final SensorManager mSensorManager;
        private final Sensor mAccelerometer;

        public SensorActivity() {
                mSensorManager = (SensorManager)getSystemService(SENSOR_SERVICE);
                mAccelerometer = mSensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETE
R);
         }

        protected void onResume() {
                super.onResume();
                mSensorManager.registerListener(this, mAccelerometer, SensorManager.SENSOR_
DELAY_NORMAL);
        }

        protected void onPause() {
                super.onPause();
                mSensorManager.unregisterListener(this);
        }

        public void onAccuracyChanged(Sensor sensor, int accuracy) {
        }

        public void onSensorChanged(SensorEvent event) {
        }
 }
```

Получение значений с датчика осуществляется с помощью объекта типа SensorEvent, который передается внутрь метода onSensorChanged.

За подробностями работы с SensorEvent можно обратиться к официальной справочной системе.

Пример 3.4*

Реализуем приложение, отображающие на экране информацию об углах наклона аппарата в трех плоскостях. Приложение будет состоять из одной активности с текстовыми полями с необходимой информацией. Создадим разметку для главной активности с шестью текстовыми полями:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent"
   android:orientation="vertical" >

   <TextView
       android:id="@+id/xyAngle"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:text="Угол XY:"
       android:textSize="25dp"
       android:layout_alignParentTop="true"
       android:layout_toLeftOf="@+id/xyValue"
       android:layout_alignParentLeft="true"
       android:layout_alignParentStart="true" />


   <TextView
       android:id="@+id/xyValue"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:text="0"
       android:textSize="25dp"
       android:layout_alignParentTop="true"
       android:layout_toRightOf="@+id/xzAngle"
       android:layout_alignParentRight="true"
       android:layout_alignParentEnd="true" />

   <TextView
       android:id="@+id/xzAngle"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:text="Угол XZ:   "
       android:textSize="25dp"
       android:layout_below="@+id/xyAngle"
       android:layout_alignParentLeft="true"
       android:layout_alignParentStart="true" />

   <TextView
       android:id="@+id/xzValue"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:text="0"
       android:textSize="25dp"
       android:layout_alignBaseline="@+id/xzAngle"
       android:layout_alignBottom="@+id/xzAngle"
       android:layout_toRightOf="@+id/zyAngle"
       android:layout_alignParentRight="true"
       android:layout_alignParentEnd="true" />

   <TextView
       android:id="@+id/zyAngle"
       android:layout_width="wrap_content"
```

```
        android:layout_height="wrap_content"
        android:text="Угол ZY:   "
        android:textSize="25dp"
        android:layout_alignBaseline="@+id/zyValue"
        android:layout_alignBottom="@+id/zyValue"
        android:layout_alignParentLeft="true"
        android:layout_alignParentStart="true" />

    <TextView
        android:id="@+id/zyValue"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="25dp"
        android:text="0"
        android:layout_below="@+id/xzValue"
        android:layout_toRightOf="@+id/zyAngle"
        android:layout_alignParentRight="true"
        android:layout_alignParentEnd="true" />

</RelativeLayout>
```

Для получения значений угла поворота устройства можно использовать метод getOrientation(float[] R, float[] values). В этот метод передается два аргумента:

- R — матрица поворота устройства;
- values — массив из трех элементов типа float, в который запишутся углы наклона аппарата в радианах.

Для начала нам необходимо получить матрицу поворота. Сделать это можно с помощью метода:

```
 getRotationMatrix (float[] R, float[] I, float[] gravity, float[] geomagnetic);
```

Подробное описание работы этого метода можно найти по ссылке. Данный метод в первые два массива помещает матрицу поворота и матрицу отклонения аппарата от магнитного полюса Земли. Для этого ему нужно также передать данные с датчика акселерометра и геомагнитного датчика.

В файле MainActivity.java объявим необходимые поля:

```
 private SensorManager sensorManager; //менеджер сенсоров

private float[] rotationMatrix; //матрица поворота

private float[] accelerometer;  //данные с акселерометра
private float[] geomagnetism;   //данные геомагнитного датчика
private float[] orientation;    //матрица положения в пространстве

//текстовые поля для вывода информации
private TextView xyAngle;
private TextView xzAngle;
private TextView zyAngle;
```

Первая переменная — менеджер сенсоров устройства. Именно она предоставляет доступ к интересующему нас датчику. Для работы с сенсорами класс MainActivity.java должен реализовывать методы интерфейса SensorEventListener.

```
 public class MainActivity extends AppCompatActivity implements SensorEventListener{
        ...
}
```

При этом в класс главной активности добавятся два обязательных метода onSensorChanged() и onAccuracyChanged(). Первый метод вызывается, когда значения датчика изменились. Второй вызывается, когда точность зарегистрированного датчика изменилась. В отличие от onSensorChanged() вызывается только при изменении значения точности. В нашем приложении можно оставить этот метод пустым.

Методе onCreate() может выглядеть следующим образом:

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

        sensorManager = (SensorManager)getSystemService(Context.SENSOR_SERVICE); //получаем
объект менеджера датчиков

        rotationMatrix = new float[16];
        accelerometer = new float[3];
        geomagnetism = new float[3];
        orientation = new float[3];

        // поля для вывода показаний
        xyAngle = (TextView) findViewById(R.id.xyValue);
        xzAngle= (TextView) findViewById(R.id.xzValue);
        zyAngle = (TextView) findViewById(R.id.zyValue);

        setContentView(R.layout.activity_main);
}
```

Теперь нужно создать метод onResume(), в котором уточняем данные необходимых датчиков:

```
 @Override
protected void onResume() {
        super.onResume();
        sensorManager.registerListener(this, sensorManager.getDefaultSensor(Sensor.TYPE_ACC
ELEROMETER), SensorManager.SENSOR_DELAY_UI );
        sensorManager.registerListener(this, sensorManager.getDefaultSensor(Sensor.TYPE_MAG
NETIC_FIELD), SensorManager.SENSOR_DELAY_UI );
}
```

В методе onResume() мы передаем в registerListener() тип нужного нам датчика (полный список датчиков) и частоту обновления данных (SENSOR_DELAY_NORMAL, SENSOR_DELAY_UI, SENSOR_DELAY_GAME или SENSOR_DELAY_FASTEST в порядке увеличения частоты обновления).

Чтобы наша программа не использовала ресурсы устройства в режиме паузы, в методе onPause() останавливаем получение данных:

```
 @Override
protected void onPause() {
        super.onPause();
        sensorManager.unregisterListener(this);
}
```

Теперь создадим метод loadSensorData(), в котором будем собирать данные с датчиков и заносить в соответствующий датчику массив:

```
 private void loadSensorData(SensorEvent event) {
       final int type = event.sensor.getType(); //определяем тип датчика
       if (type == Sensor.TYPE_ACCELEROMETER) { //если акселерометр
               accelerometer = event.values.clone();
       }

       if (type == Sensor.TYPE_MAGNETIC_FIELD) { //если геомагнитный датчик
               geomagnetism = event.values.clone();
       }
}
```

Наконец, осталось написать обработчик события onSensorChanged.

```
 @Override
public void onSensorChanged(SensorEvent event) {
       loadSensorData(event); // получаем данные с датчика
       SensorManager.getRotationMatrix(rotationMatrix, null, accelerometer, geomagnetism);
//получаем матрицу поворота
       SensorManager.getOrientation(rotationMatrix, orientation); //получаем данные ориент
ации устройства в пространстве

       if((xyAngle ==null)||(xzAngle==null)||(zyAngle ==null)){
               xyAngle = (TextView) findViewById(R.id.xyValue);
               xzAngle = (TextView) findViewById(R.id.xzValue);
               zyAngle = (TextView) findViewById(R.id.zyValue);
       }

       //вывод результата
       xyAngle.setText(String.valueOf(Math.round(Math.toDegrees(orientation[0]))));
       xzAngle.setText(String.valueOf(Math.round(Math.toDegrees(orientation[1]))));
       zyAngle.setText(String.valueOf(Math.round(Math.toDegrees(orientation[2]))));
}
```

В итоге после запуска приложения на экране устройства получим значения углов поворота в трех плоскостях (см. рис. 3.14).

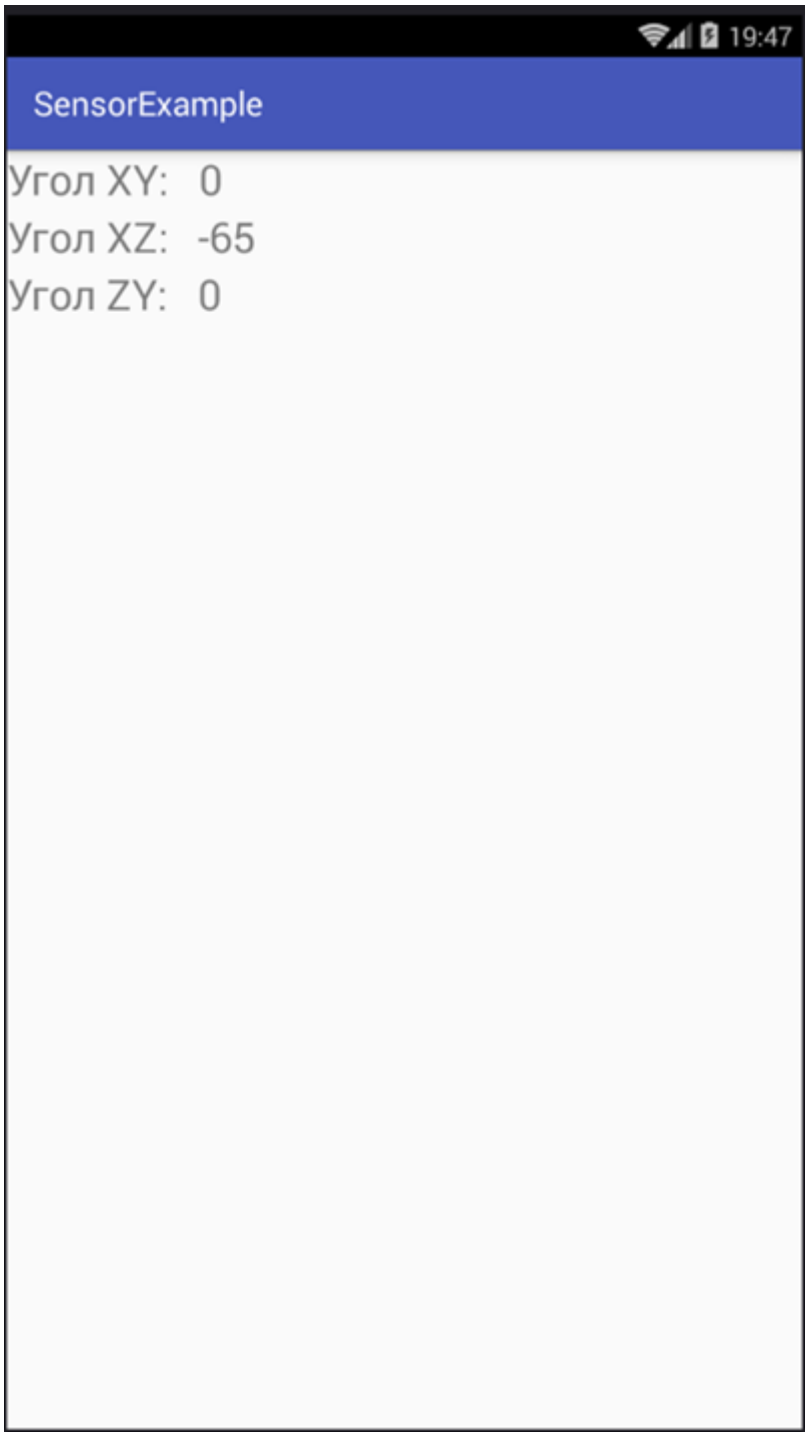


Рис. 3.14.

При изменении положения устройства значения в текстовых полях будут меняться.

3.6. Фрагменты в Android

Сайт: IT Академия SAMSUNG
Курс: MDev @ IT Академия Samsung
Книга: 3.6. Фрагменты в Android
Напечатано:: Егор Беляев
Дата: Суббота, 18 Апрель 2020, 19:24

Оглавление

3.6.1. Введение в фрагменты

Фрагменты — одно из главных нововведений Android 3 (начиная с API 11). Их главное назначение — обеспечение большей динамичности и гибкости пользовательских интерфейсов на больших экранах, например, у планшетов. Можно рассматривать их как встроенные в активности элементы, которые располагаются в основном Activity и имеют свой жизненный цикл, немного отличающийся от обычного, уже знакомого нам жизненного цикла Activity.

Фрагмент (класс Fragment) — это часть пользовательского интерфейса в Activity. Разработчик может размещать в одной активности несколько фрагментов для построения удобного пользовательского интерфейса. Возможно повторное использование фрагментов в нескольких активностях. Фрагменты можно добавлять или удалять непосредственно во время выполнения активностей. Это нечто вроде вложенной активности, которую можно многократно использовать в различных активностях.

Рассмотрим приложение, которое отображает список, при этом пользователь при выборе элемента списка получает дополнительную информацию в виде детализации к элементу списка.

Одним из вариантов реализации интерфейса такого приложения является использование двух активностей: одна активность — для управления списком, другая — для детализации.

Однако, если мы возьмем планшет в альбомной ориентации, то представление одного лишь списка на экране будет неэстетично и нерационально, потому что образуется много нефункционального пространства.

Второй вариант построения интерфейса более логичен: список и его детализацию выводить на один экран в следующем виде (рис. 3.15).

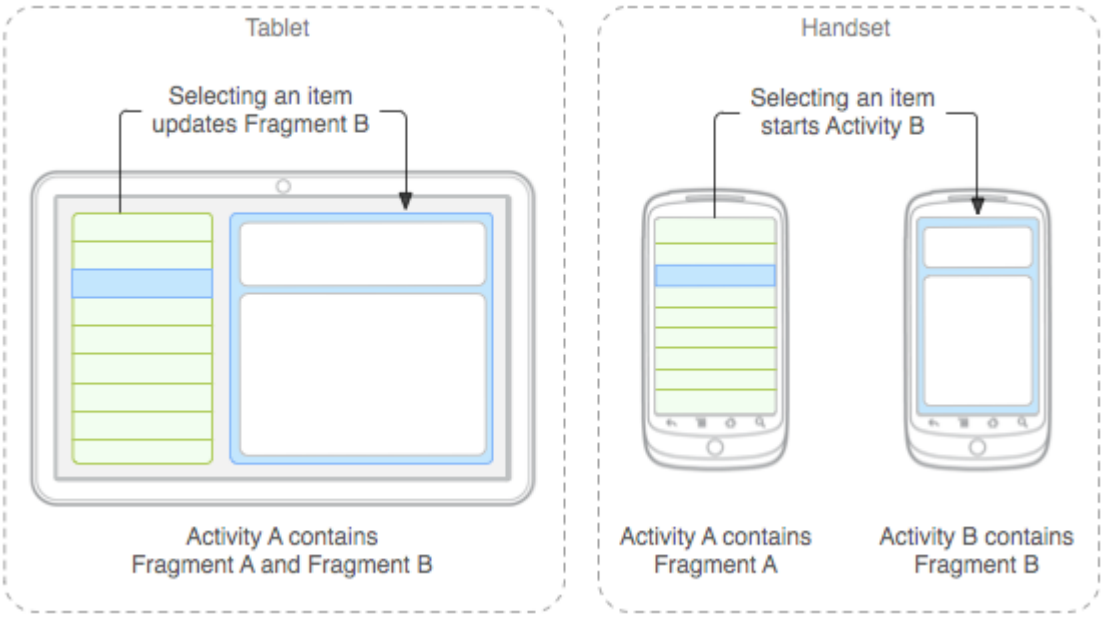


Рис. 3.15.

Чтобы построить подобный дизайн, используют фрагменты. Они были придуманы именно для того, чтобы строить интерфейс, адаптированный под смартфоны и планшеты разных моделей и размеров.

В следующем модуле мы научимся создавать похожие списки, используя так называемые адаптеры. После изучения адаптеров вы сможете создать и усовершенствовать подобный интерфейс. В данной теме подробнее остановимся на некоторых возможностях, которые нам предоставляет использование фрагментов.

3.6.2. Создание фрагментов

Фрагменты всегда являются частью какой-либо активности, при этом на жизненный цикл фрагментов влияют события, происходящие с активностью. Например, при остановке или уничтожении активности будут остановлены или уничтожены входящие в нее фрагменты.

При добавлении фрагмента в активность он помещается в ViewGroup в иерархии, представлений (View) активности, при этом каждый фрагмент имеет внутри себя свою собственную разметку.

Существует два способа создания фрагментов:

- добавление фрагмента в макет активности (как элемент <fragment>) — такой способ прост, но недостаточно гибок. Но иногда такие фрагменты полезны. Данный метод позволяет жестко привязать фрагмент к представлению активности, но в данном случае не получится переключить фрагмент на протяжении жизненного цикла активности;
- добавление в код активности (добавить его в существующий объект ViewGroup) — такой метод сложен, но позволяет управлять фрагментами во время выполнения. При таком способе разработчик сам определяет, когда фрагмент добавляется в активность и что с ним происходит потом. Он может удалить фрагмент, заменить его другим фрагментом, а потом вернуть первый.

Создание фрагмента практически ничем не отличается от создания активности. Для создания фрагмента требуется класс, унаследованный от класса Fragment.

```
public class MyFragment extends Fragment {


}
```

В отличии от активности, при создании нового фрагмента не нужно его декларировать в файле Manifest.xml. Его фрагмент содержит методы, аналогичные (но не полностью) методам класса Activity, такие как onCreate(), onStart(), onPause() и onStop().

Обычно требуется обеспечить реализацию следующих методов:

- onCreate(): вызывается при создании фрагмента; здесь инициализируются компоненты, которые нужно восстановить при возобновлении работы фрагмента после остановки;
- onCreateView(): вызывается, когда нужно «нарисовать» фрагмент на экране в первый раз; этот метод должен возвращать представление (View), которое является корневым для вашего фрагмента;
- onPause(): вызывается, когда пользователь покидает фрагмент и он (фрагмент) может быть уничтожен; здесь следует сохранить несохраненные данные, поскольку обратно к фрагменту пользователь может не вернуться.

Кроме класса Fragment, разработчику доступны несколько его подклассов, которые могут быть более удобны в разных ситуациях. Например:

- DialogFragment: используется для показа «плавающего» диалога;
- ListFragment: фрагмент, являющийся функциональным аналогом ListActivity;
- PreferenceFragment: функциональный аналог PreferenceActivity.

Обычно фрагменты используются как часть UI активности со своей собственной разметкой. Объект класса View, отображающий эту разметку, является возвращаемым значением метода onCreateView() фрагмента, вызываемым перед отображением фрагмента. Как и в случае с ListActivity, для класса ListFragment не требуется явно указывать разметку, по умолчанию

используется ListView, так что метод onCreateView() реализовывать нет необходимости. Пример метода класса с методом onCreateView(), использующего для фрагмента разметку из файла:

```
 public class SampleFragmentA extends Fragment {
    @Override
    public View onCreateView(LayoutInflater inflater,
        ViewGroup container, Bundle savedInstanceState) {
        return inflater.inflate(R.layout.fragment_a, container, false);
    }
}
```

При таких параметрах метода inflate второй параметр (container) обычно используется только для передачи свойств генерируемой иерархии объектов ViewGroup/View, а третий параметр указывает на то, что эту иерархию не нужно привязывать к параметру container.

Для указания фрагментов в файле разметки, используемом активностью, применяется тег <fragment> (обратите внимание на значение атрибута android:name, это полное имя класса, реализующего данный фрагмент):

```
 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:baselineAligned="false"
    android:orientation="horizontal" >

    <fragment
        android:id="@+id/fragment_a"
        android:name="com.example.fragmentssample.SampleFragmentA"
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="2" />

    <FrameLayout
       android:id="@+id/place_holder"
       android:layout_width="0dp"
       android:layout_height="match_parent"
       android:layout_weight="3" />

</LinearLayout>
```

Для каждого фрагмента используется собственный файл разметки, ниже показан самый простой вариант:

```
 <?xml version="1.0" encoding="utf-8"?>
<TextView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="#100F"
    android:gravity="center"
    android:text="FRAGMENT A" />
```

При необходимости фрагменты можно создавать и добавлять к активности и без указания специальных тегов, а используя любой объект View. Ниже показан вариант метода onCreate() активности с фрагментами, в которой первый фрагмент вставляется «автоматически» благодаря указанию в разметке активности, а второй — с помощью FragmentManager:

```
public class MainActivity extends FragmentActivity {
   @Override
   protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.activity_main);
       FragmentManager fm = getSupportFragmentManager();
       FragmentTransaction ft = fm.beginTransaction();
       SampleFragmentB fb = new SampleFragmentB();
       ft.add(R.id.place_holder, fb);
       ft.commit();
   }
}
```

Как показано в примере выше, при использовании библиотек совместимости (support libraries) требуется наследовать активность от класса FragmentActivity и вызывать ее метод getSupportFragmentManager() вместо метода getFragmentManager() обычной Activity (поддержка фрагментов появилась в Android API-11, поэтому для их использования в более ранних версиях применяются support libraries). В остальном код выглядит точно так же.

3.6.3. Класс Fragment и его методы

Класс Fragment представляет поведение или часть пользовательского интерфейса в классе Activity. Жизненный цикл фрагмента схож с жизненным циклом активности и зависит от него.

Методы жизненного цикла фрагмента

Для того чтобы удачно манипулировать методами жизненного цикла фрагмента, необходимо понимать, какие циклы после каких происходят. Переопределять данные методы (по аналогии с жизненным циклом активности) необходимо, только если нужно модифицировать логику фрагментов.

Ниже представлены основные методы жизненного цикла фрагмента (см. рис. 3.16).

- onAttach(Activity) — вызывается после того, как фрагмент связывается с активностью.
- onCreate(Bundle) — вызывается при создании фрагмента. В своей реализации разработчик должен инициализировать ключевые компоненты фрагмента, которые требуется сохранить, когда фрагмент находится в состоянии паузы или возобновлен после остановки.
- onCreateView (LayoutInflater, ViewGroup, Bundle) — вызывается при первом отображении пользовательского интерфейса фрагмента на дисплее. Позволяет инициализировать компоненты с layout. На данном этапе мы уже имеем всю иерархию компонентов.
- onActivityCreated(Bundle) — вызывается после того, как активность завершила свою обработку метода Activity.onCreate. Тут можно выполнять заключительные настройки интерфейса до того, как пользователь увидит фрагмент.

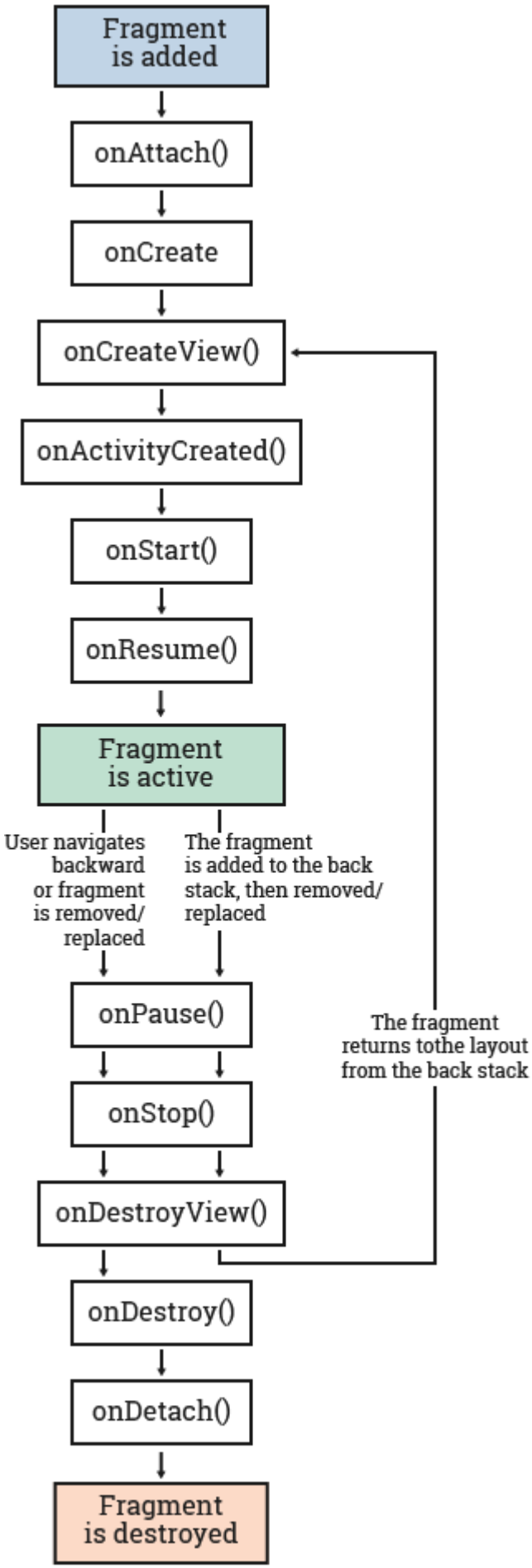


Рис. 3.16

- onStart() — похож на Activity.onStart(). Пользователь в момент вызова данного метода уже видит фрагмент, но еще не может взаимодействовать с ним.
- onResume() — вызывается после возвращения к нашему фрагменту (аналогично как в активности). После этого пользователь видит наш фрагмент и может выполнить какие-то действия.
- onPause() — если пользователь уходит на какую-то другую активность, то вызывается данный метод. Работа такая же, как в Activity.onPause() — останавливает наш фрагмент. Можно уходя с фрагмента, например, останавливать видео или звук.
- onSaveInstanceState() — позволяет сохранять состояние фрагмента и восстановить его во время выполнения onCreate(), onCreateView() или onActivityCreated().
- onStop() — связан с методом Activity, по логике выполняет то же, что и в Activity.onStop(). Останавливает работу нашего фрагмента. Но на этом жизненный цикл не заканчивается. После этого выполняется onDestroyView().
- onDestroyView() — если фрагмент находится на пути уничтожения или сохранения, то следующим будет вызван именно этот метод — в данном методе мы можем возродить наш фрагмент.
- onDestroy() — можно почистить ресурсы. Удалить те объекты, которые нам не нужно использовать.
- onDetach() — отвязывает фрагмент от активности.

Фрагмент всегда должен быть встроен в Activity, и на его жизненный цикл напрямую влияет жизненный цикл Activity.

Например, когда активность приостановлена, в том же состоянии находятся и все фрагменты внутри нее. А когда активность уничтожается, уничтожаются и все фрагменты. Однако пока активность выполняется (это соответствует состоянию Resume жизненного цикла), можно манипулировать каждым фрагментом независимо, например, добавлять или удалять их.

Ниже приведена схема влияния жизненного цикла активности на жизненный цикл фрагмента (рис. 3.17).

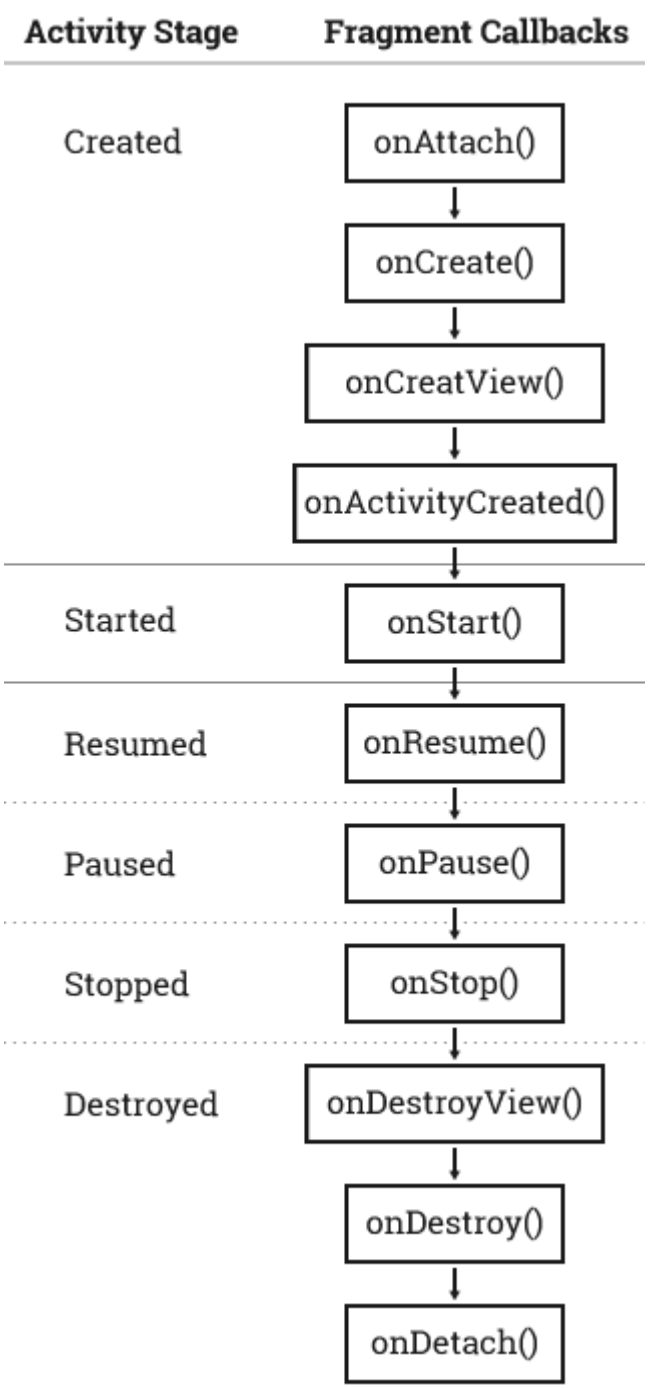


Рис. 3.17.

Пример 3.5. Указание фрагментов в файле разметки

Создадим проект, в котором рассмотрим создание фрагментов и их использование. На главной активности (activity_main.xml) будем отображать: две кнопки, растянутые по всей длине, ImageView и два элемента CheckBox. В сам проект необходимо добавить папку drawable, которую нужно привязать к ImageView.

Чтобы показать преимущества использования фрагментов, добавим в проект еще три новые разметки для фрагментов (*File -> New -> XML -> LayoutXML file*). Создание разметки для фрагмента ничем не отличается от создания разметки для активности.

Разметки могут выглядеть следующим образом: первый фрагмент содержит две кнопки. Файл разметки *button_fragment.xml*:

```
 <?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:orientation="vertical" >

    <Button
        android:id="@+id/button"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="20dp"
        android:text="Button 1" />
    <Button
        android:id="@+id/button2"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Button 2" />

</LinearLayout>
```

Второй фрагмент будет содержать один элемент ImageView. Файл разметки image_fragment.xml:

```
 <?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:orientation="vertical" >

    <ImageView
        android:id="@+id/imageView"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_gravity="center_horizontal"
        android:layout_marginTop="20dp"
        android:src="@drawable/img" />

</LinearLayout>
```

С помощью атрибута android:src=«@drawable/img» в ImageView добавляется картинка с именем img из папки drawable.

Третий фрагмент содержит элементы checkbox. Файл разметки checkbox_fragment.xml:

```
 <?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:orientation="vertical" >

    <CheckBox
        android:id="@+id/checkBox"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal"
        android:layout_marginTop="20dp"
        android:checked="false"
        android:text="New component 1" />
    <CheckBox
        android:id="@+id/checkBox2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal"
        android:checked="false"
        android:text="New component 2" />

</LinearLayout>
```

Чтобы обрабатывать элементы, которые содержаться во фрагментах, необходимо создать дополнительные java-классы для каждого фрагмента. При этом они должны наследоваться от класса Fragment или его подклассов.

При этом необходимо переопределить метод onCreateView. Метод возвращает то, что необходимо отображать внутри фрагмента. Мы сообщаем системе, что хотим видеть во фрагменте содержимое соответствующего layout-файла. Для этого мы сами создаем View с помощью объекта inflater и передаем его системе. По смыслу это аналог метода setContentView, который мы вызываем в активностях. Отличие в том, что здесь нам приходится вручную создавать View, а не просто передавать идентификатор layout-файла.

Класс ButtonFragment может выглядеть следующим образом:

```
public class ButtonFragment extends Fragment {

        @Override
        // Переопределяем метод onCreateView
        public View onCreateView(LayoutInflater inflater, ViewGroup container,
                                 Bundle savedInstanceState) {

                // менеджер компоновки, который позволяет получать доступ к layout с наших
 ресурсов
                View view = inflater.inflate(R.layout.button_fragment, container, false);

                // теперь можем получить наши элементы, расположенные во фрагменте
                Button button = (Button) view.findViewById(R.id.button);
                button.setOnClickListener(new View.OnClickListener() {
                        @Override
                        public void onClick(View v) {
                                Toast.makeText(getActivity(), "Message from fragment", Toas
t.LENGTH_LONG).show();
                        }
                });
                return view;
        }
}
```

Тоже самое по аналогии сделаем для классов CheckBoxFragment и ImageFragment. Классы CheckBoxFragment и ImageFragment могут выглядеть следующим образом:

```
public class CheckBoxFragment extends Fragment {

        @Override
        // Переопределяем метод onCreateView
        public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle saved
InstanceState) {
                View view = inflater.inflate(R.layout.checkbox_fragment, container, false);
                return view;
        }
}
public class ImageFragment extends Fragment {

        @Override
        // Переопределяем метод onCreateView
        public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle saved
InstanceState) {
                View view = inflater.inflate(R.layout.image_fragment, container, false);
                return view;
        }
}
```

Теперь в разметке основной активности (activity_main.xml) можем подключить наши фрагменты:

```
 <?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >

    <fragment
        android:id="@+id/button_fragment"
        android:name="ru.samsung.itschool.book.staticfragment.ButtonFragment"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        tools:layout="@layout/button_fragment" />
    <fragment
        android:id="@+id/checkbox_fragment"
        android:name="ru.samsung.itschool.book.staticfragment.CheckBoxFragment"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        tools:layout="@layout/checkbox_fragment" />
    <fragment
        android:id="@+id/image_fragment"
        android:name="ru.samsung.itschool.book.staticfragment.ImageFragment"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1"
        tools:layout="@layout/image_fragment" />

</LinearLayout>
```

Обратите внимание, что в атрибутах android:name ru.samsung.itschool.book.staticfragment — это название пакета, и, возможно, у вас оно будет другое.

Такими образом, мы разнесли три блока функциональности (кнопки, картинки и чекбоксы) по разным лайотам: по сути разбили нашу активность на несколько частей. Эти части и есть наши фрагменты.

Чтобы понять преимущества использования фрагментов, создадим альбомную отдельную разметку для ландшафтной ориентации экрана устройства. Для этого необходимо создать в ресурсах папку *layout-land* и определить там разметку или нажать кнопку и выбрать пункт *Create Landscape Variation* (см. рис. 3.18).

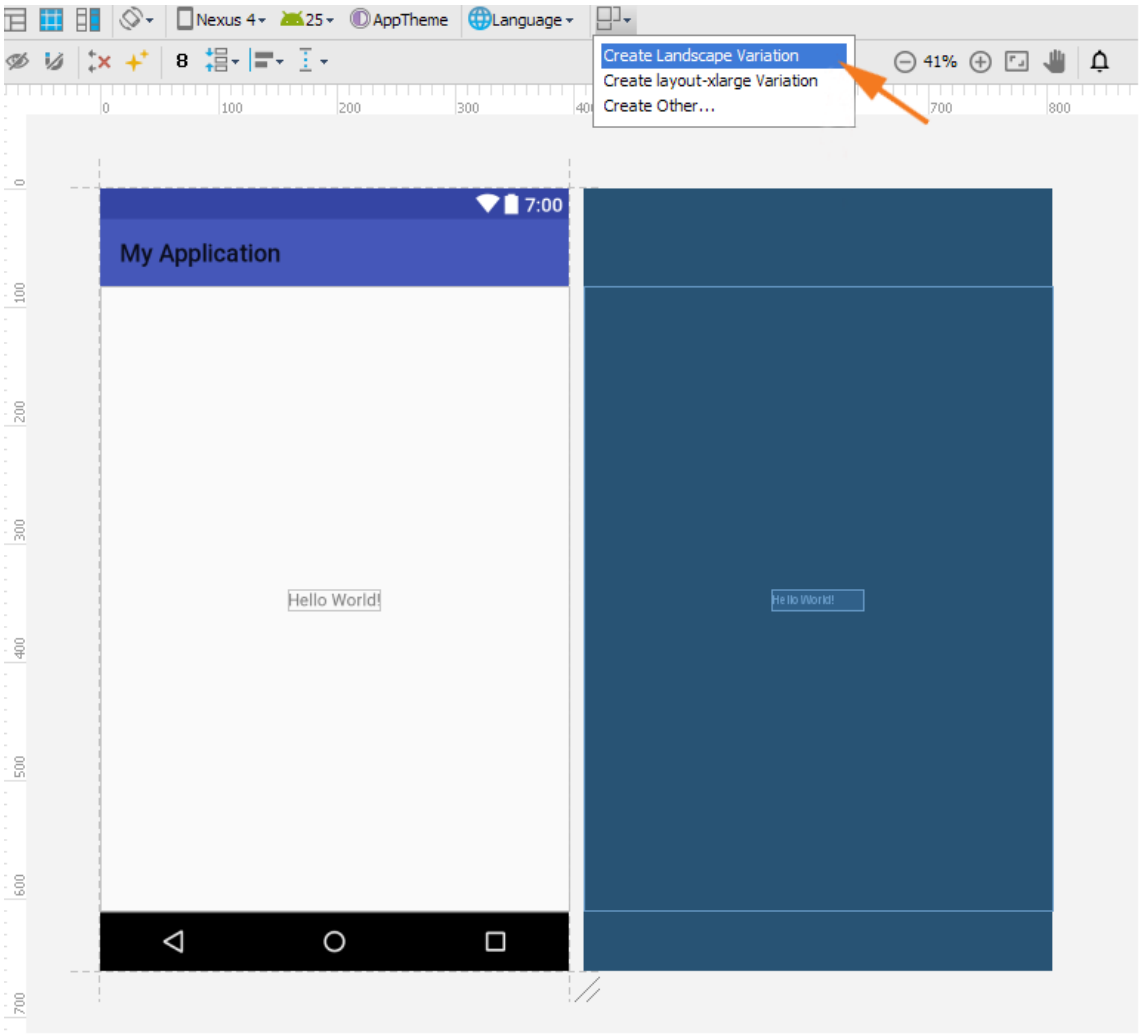


Рис. 3.18.

В результате появится файл разметки для альбомной ориентации (рис. 3.19).

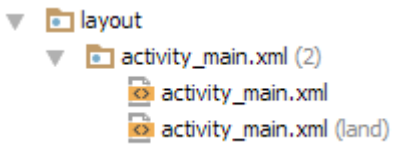


Рис. 3.19.

Файл layout-land/activity_main.xml может выглядеть следующим образом:

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

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="horizontal" >

    <RelativeLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_weight="2"
        android:orientation="vertical">

        <fragment
            android:id="@+id/button_fragment"
            android:name="ru.samsung.itschool.book.staticfragment.ButtonFragment"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_alignParentLeft="true"
            android:layout_alignParentTop="true"
            android:layout_marginLeft="15dp"
            tools:layout="@layout/button_fragment"
            android:layout_alignParentRight="true"
            android:layout_alignParentEnd="true" />

        <fragment
            android:id="@+id/checkbox_fragment"
            android:name="ru.samsung.itschool.book.staticfragment.CheckBoxFragment"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            tools:layout="@layout/checkbox_fragment"
            android:layout_below="@+id/button_fragment"
            android:layout_alignParentRight="true"
            android:layout_alignParentEnd="true"
            android:layout_alignParentBottom="true" />
    </RelativeLayout>

    <RelativeLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_weight="1">
        <fragment
            android:id="@+id/image_fragment"
            android:name="ru.samsung.itschool.book.staticfragment.ImageFragment"
            android:layout_width="358dp"
            android:layout_height="wrap_content"
            tools:layout="@layout/image_fragment"
            android:layout_alignParentTop="true"
            android:layout_alignParentLeft="true"
            android:layout_alignParentStart="true"
            android:layout_alignParentBottom="true"
            android:layout_alignParentRight="true"
```

```
            android:layout_alignParentEnd="true" />
    </RelativeLayout>

</LinearLayout>
```

Такой подход позволяет, используя те же фрагменты, создавать различный интерфейс для разной ориентации устройства.

Наконец, для того чтобы фрагменты заработали на основной активности, необходимо наследовать наш класс от FragmentActivity.

Файл MainActivity.java выглядит следующим образом:

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

import android.support.v4.app.FragmentActivity;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;

public class MainActivity extends FragmentActivity {

        @Override
        protected void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                setContentView(R.layout.activity_main);
        }
}
```

В результате у нас получилось приложение с простым функционалом: при нажатии на кнопку появляется всплывающее сообщение (см. рис. 3.20).

Рис. 3.20.

При повороте экрана, у нас загружается разметка из папки *layout-land* (рис. 3.21).

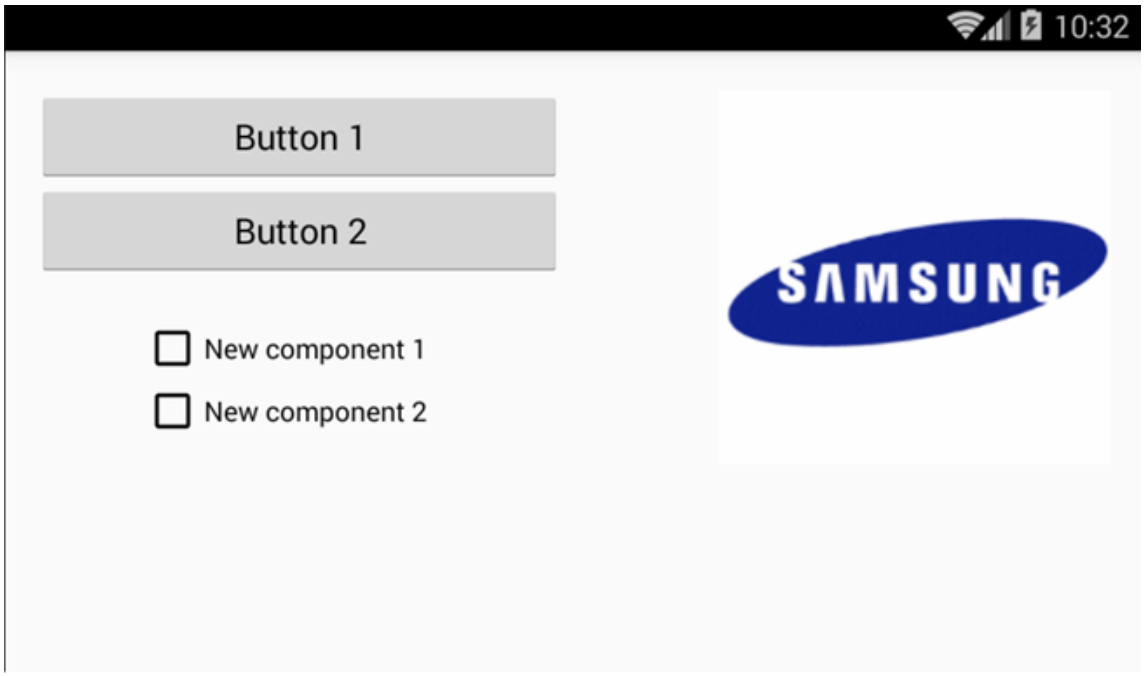


Рис. 3.21.

Полный код проекта можно посмотреть здесь.

3.6.4. Управление фрагментами

Объекты FragmentManager

Для управления фрагментами в активности необходим экземпляр класса FragmentManager.

Чтобы получить его, следует вызвать метод getFragmentManager() из кода активности.

```
FragmentManager fragmentManager = getFragmentManager();
```

Класс FragmentManager имеет два метода, позволяющих найти фрагмент, связанный с активностью:

- findFragmentById(int id) — находит фрагмент по идентификатору;
- findFragmentByTag(String tag) — находит фрагмент по заданному тегу.

Можно использовать класс FragmentManager для открытия FragmentTransaction, что позволяет выполнять транзакции с фрагментами, например, добавление и удаление.

Объекты FragmentTransaction

Большим достоинством использования фрагментов в активности является возможность добавлять, удалять, заменять их и выполнять другие действия с ними в ответ на действия пользователя. Любой набор изменений, вносимых в активность, называется ***транзакцией***. Каждую транзакцию можно сохранять в стеке переходов назад, которым управляет активность. Это позволяет перемещаться назад по изменениям во фрагментах (аналогично перемещению назад по активностям).

Экземпляр класса FragmentTransaction можно получить от класса FragmentManager, например, так:

```
 FragmentManager fragmentManager = getFragmentManager();
FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();
```

Каждая транзакция является набором изменений, выполняемых одновременно. Вы можете указать все изменения, которые нужно выполнить в данной транзакции, вызывая следующие методы:

- add() — добавляет фрагмент в активность;
- remove() — удаляет фрагмент из активности;
- replace() — заменяет один фрагмент на другой;
- hide() — делает фрагмент невидимым на экране;
- show() — отображает скрытый фрагмент на экран;
- detach() (API 13) — отсоединяет фрагмент от графического интерфейса, при этом экземпляр класса сохраняется;
- attach() (API 13) — присоединяет фрагмент, который был отсоединен методом detach().

Перед началом транзакции необходимо создать объект класса FragmentTransaction, используя метод FragmentManager.beginTransaction(). Далее необходимо вызвать различные методы для управления фрагментами.

В завершении любой транзакции, которая состоит из цепочки вышеперечисленных методов, необходимо вызвать метод commit(). До вызова метода commit() у разработчика может возникнуть необходимость вызвать метод addToBackStack(), чтобы добавить транзакцию в стек переходов назад по транзакциям фрагмента. Этим стеком переходов назад управляет активность, что позволяет пользователю вернуться к предыдущему состоянию фрагмента, нажав кнопку «Назад».

```
 // Создание нового фрагмента и транзакции
Fragment newFragment = new ExampleFragment();
FragmentTransaction transaction = getFragmentManager().beginTransaction();

// Замена контейнер в разметке на фрагмент
// и добавляем транзакцию в стек обратного вызова
transaction.replace(R.id.fragment_container, newFragment);
transaction.addToBackStack(null);

// выполнение транзакции
transaction.commit();
```

В приведенном коде объект newFragment замещает фрагмент (если таковой имеется), находящийся в контейнере макета, на который указывает идентификатор R.id.fragment_container. В результате вызова метода addToBackStack() транзакция замены сохраняется в стеке переходов назад, чтобы пользователь мог обратить транзакцию и вернуть предыдущий фрагмент, нажав кнопку «Назад».

Если в транзакцию добавить несколько методов (например, еще раз вызвать add() или remove()), а затем вызвать addToBackStack(), то все изменения, примененные до вызова метода commit(), будут добавлены в стек переходов назад как одна транзакция, и кнопка «Назад» обратит их все вместе.

Пример 3.6. Динамическое создание фрагментов

Разработаем приложение, в котором происходит динамическое добавление фрагментов в активность. При этом для разных ориентаций экрана приложение будет иметь различный вид.

В итоге должно получиться следующее (рис. 3.22).

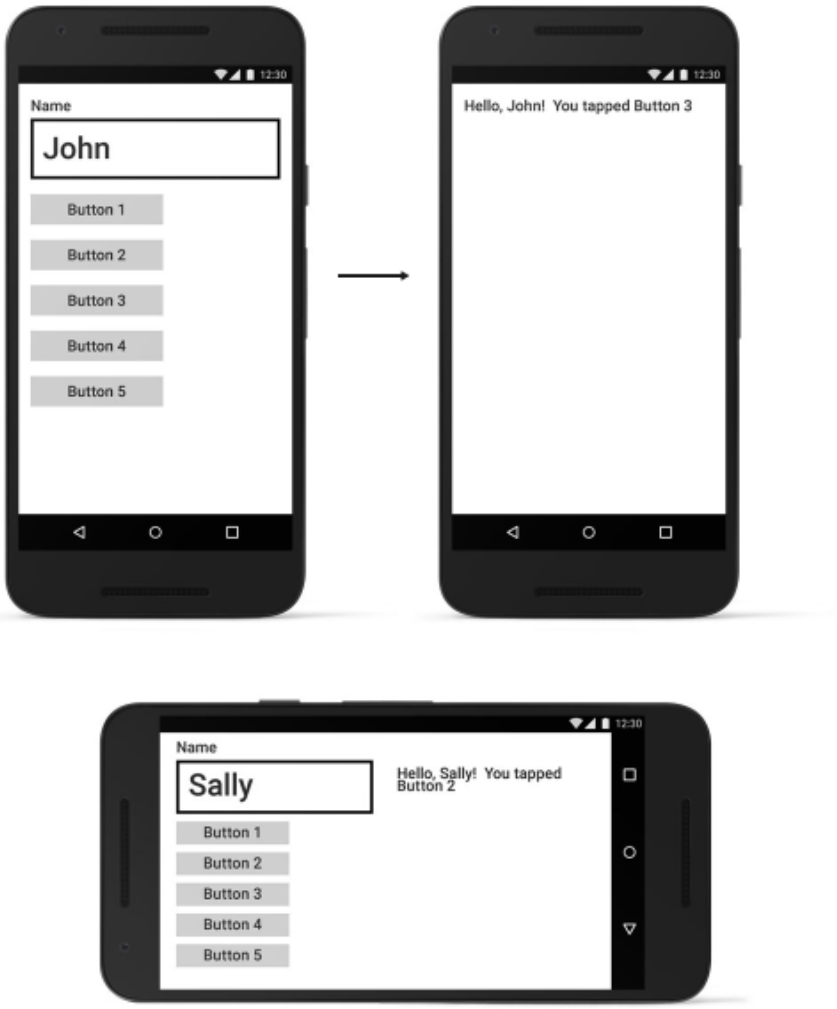


Рис. 3.22.

По аналогии с предыдущим примером создадим разметку для фрагментов InputFragment (фрагмент с кнопками и полем ввода данных) и MessageFragment (фрагмент с полем вывода сообщения).

Файл input_fragment.xml.

```
 <?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical" android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:id="@+id/textView"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Input your name:" />

    <EditText
        android:id="@+id/name"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:ems="10"
        android:inputType="textPersonName" />

    <Button
        android:id="@+id/button1"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Button 1" />

    <Button
        android:id="@+id/button2"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Button 2" />

    <Button
        android:id="@+id/button3"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Button 3" />

    <Button
        android:id="@+id/button4"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Button 4" />

    <Button
        android:id="@+id/button5"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Button 5" />
</LinearLayout>
```

Файл message_fragment.xml:

```
 <?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical" android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:id="@+id/message"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Mesage"
        android:textAppearance="@style/TextAppearance.AppCompat.Display3" />
</LinearLayout>
```

Класс MessageFragment может выглядеть следующим образом:

```
 public class MessageFragment extends Fragment{
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.message_fragment, container, false);
        return view;
    }

    public void setMessage(String item){
        TextView message = (TextView) getView().findViewById(R.id.message);
        message.setText(item);
    }
}
```

Класс содержит методы onCreateView для создания компонентов внутри фрагмента и метод setMessage для установки сообщения в TextView.

Класс InputFragment будет выглядеть следующим образом:

```
public class InputFragment extends Fragment implements View.OnClickListener{
   public View onCreateView(LayoutInflater inflater, ViewGroup container,
                            Bundle savedInstanceState) {
       View view = inflater.inflate(R.layout.input_fragment, container, false);
       return view;
   }

   @Override
   public void onActivityCreated(Bundle savedInstanceState) {
       super.onActivityCreated(savedInstanceState);
       Button button1 = (Button) getView().findViewById(R.id.button1);
       button1.setOnClickListener(this);

       Button button2 = (Button) getView().findViewById(R.id.button2);
       button2.setOnClickListener(this);

       Button button3 = (Button) getView().findViewById(R.id.button3);
       button3.setOnClickListener(this);

       Button button4 = (Button) getView().findViewById(R.id.button4);
       button4.setOnClickListener(this);

       Button button5 = (Button) getView().findViewById(R.id.button5);
       button5.setOnClickListener(this);

   }

   @Override
   public void onClick(View v) {

       EditText name = (EditText) getView().findViewById(R.id.name);
       String nameValue = name.getText().toString();

       String buttonNum = "";
       switch (v.getId()){
           case R.id.button1:
               buttonNum = "1";
               break;
           case R.id.button2:
               buttonNum = "2";
               break;
           case R.id.button3:
               buttonNum = "3";
               break;
           case R.id.button4:
               buttonNum = "4";
               break;
           case R.id.button5:
               buttonNum = "5";
               break;

       }
       String message = "Hello, " + nameValue + "! Your taped button №" + buttonNum;
```

```
        MessageFragment fragment = (MessageFragment)getFragmentManager().findFragmentById
(R.id.fragment_detail);
        if (fragment != null && fragment.isInLayout()) {
            fragment.setMessage(message);
        } else {
            Intent intent = new Intent(getActivity().getApplicationContext(), MessageActivi
ty.class);
            intent.putExtra("value", message);
            startActivity(intent);
        }
    }
}
```

В методе onActivityCreated создаем кнопки и назначаем им слушателей.

В методе onClick происходит следующее. Сначала создается поле ввода, и в переменную nameValue записывается значение из этого поля. Затем в зависимости от нажатой кнопки переменной buttonNum присваивается соответствующее значение. Далее в переменной message формируется передаваемое сообщение.

Дальше происходит самое интересное. С помощью метода findFragmentById создается объект fragment класса MessageFragment. Далее происходит проверка наличия фрагмента на экране. Если фрагмент на экране (ландшафтная ориентация), то с помощью метода setMessage помещаем сообщение в нужное поле. Иначе (в случае портретной ориентации) создаем объект класса Intent, с помощью метода putExtra передаем туда наше сообщение и запускаем MessageActivity.

Для запуска MessageActivity необходимо ее создать и назначить ей разметку (не забудьте продекларировать ее в манифесте приложения). Создадим файл activity_message.xml и установим в разметку соответствующий фрагмент.

```
 <?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"
    tools:context="ru.samsung.itschool.book.fragmentexample.MessageActivity">
    <fragment class="ru.samsung.itschool.book.fragmentexample.MessageFragment"
        android:id="@+id/fragment_detail"
        android:layout_width="match_parent"
        android:layout_height="match_parent" >
    </fragment>
</LinearLayout>
```

Код класса может выглядеть следующим образом:

```
public class MessageActivity extends AppCompatActivity {
   String message;
   @Override
   protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.activity_message);
       Bundle extras = getIntent().getExtras();
       message = extras.getString("value");

       TextView textView = (TextView) findViewById(R.id.message);
       textView.setText(message);
   }

   @Override
   public void onConfigurationChanged(Configuration newConfig) {
       super.onConfigurationChanged(newConfig);
       if (newConfig.orientation == Configuration.ORIENTATION_LANDSCAPE) {
           Intent intent = new Intent(this, FragmentActivity.class);
           intent.putExtra("value", message);
           startActivity(intent);
       }
   }
}
```

В методе onCreate получаем данные из намерения и устанавливаем их в textView.

```
 Bundle extras = getIntent().getExtras();
message = extras.getString("value");

TextView textView = (TextView) findViewById(R.id.message);
textView.setText(message);
```

Функция onConfigurationChanged вызывается при смене ориентации экрана. В ней проверяем, если мы попали с вертикального режима в горизонтальный, то запускаем FragmentActivity и передаем в него текущее сообщение.

Осталось создать разметку для главной активности приложения. Как и в предыдущем примере создадим разметку для разной ориентации экрана. В портретной ориентации на экране будет отображаться только InputFragment, в ландшафтной и InputFragment, и MessageFragment, как это показано на рисунке 3.22 в начале примера.

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"
    tools:context="ru.samsung.itschool.book.fragmentexample.MainActivity">

    <fragment class="ru.samsung.itschool.book.fragmentexample.InputFragment"
        android:id="@+id/fragment_list"
        android:layout_width="368dp"
        android:layout_height="495dp"
        tools:layout_editor_absoluteY="8dp"
        tools:layout_editor_absoluteX="8dp">
    </fragment>
</LinearLayout>
```

Файл layout-land/activity_main.xml:

```
 <?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:orientation="horizontal" >

        <fragment class="ru.samsung.itschool.book.fragmentexample.InputFragment"
            android:id="@+id/fragment_list"
            android:layout_weight="1"
            android:layout_width="match_parent"
            android:layout_height="match_parent" >
        </fragment>

        <fragment class="ru.samsung.itschool.book.fragmentexample.MessageFragment"
            android:id="@+id/fragment_detail"
            android:layout_weight="1"
            android:layout_width="match_parent"
            android:layout_height="match_parent" >
        </fragment>
 </LinearLayout>
```

Код класса главной активности приложения выглядит следующим образом:

```
 public class MainActivity extends FragmentActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        MessageFragment fragment =  (MessageFragment)getFragmentManager().findFragmentById
(R.id.fragment_detail);
        Bundle extras = getIntent().getExtras();
        String value = extras.getString("value");
        if(value !=null && fragment != null && fragment.isInLayout()){
            fragment.setMessage(value);
        }
    }
```

В методе onCreate создается объект fragment класса MessageFragment. С помощью строк

```
 Bundle extras = getIntent().getExtras();
String value = extras.getString("value");
```

получаем нужную строку из намерения. И если fragment находится на экране (ландшафтная ориентация), вставляем строку сообщение:

```
 if(value !=null && fragment != null && fragment.isInLayout()){
            fragment.setMessage(value);
}
```

В результате запуска приложения получим следующее.

Для портретной ориентации (рис. 3.23).

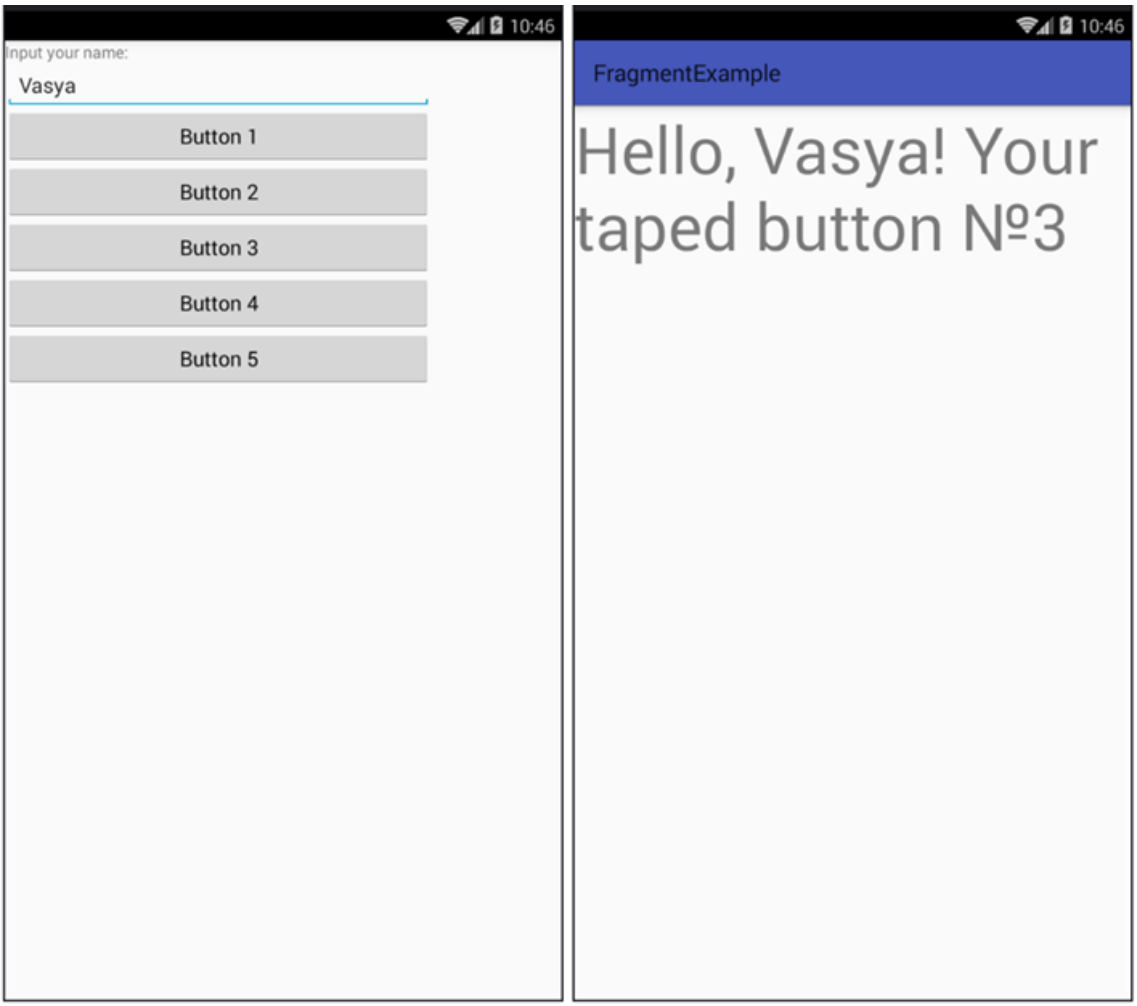


Рис. 3.23.

Для ландшафтной ориентации (рис. 3.24).

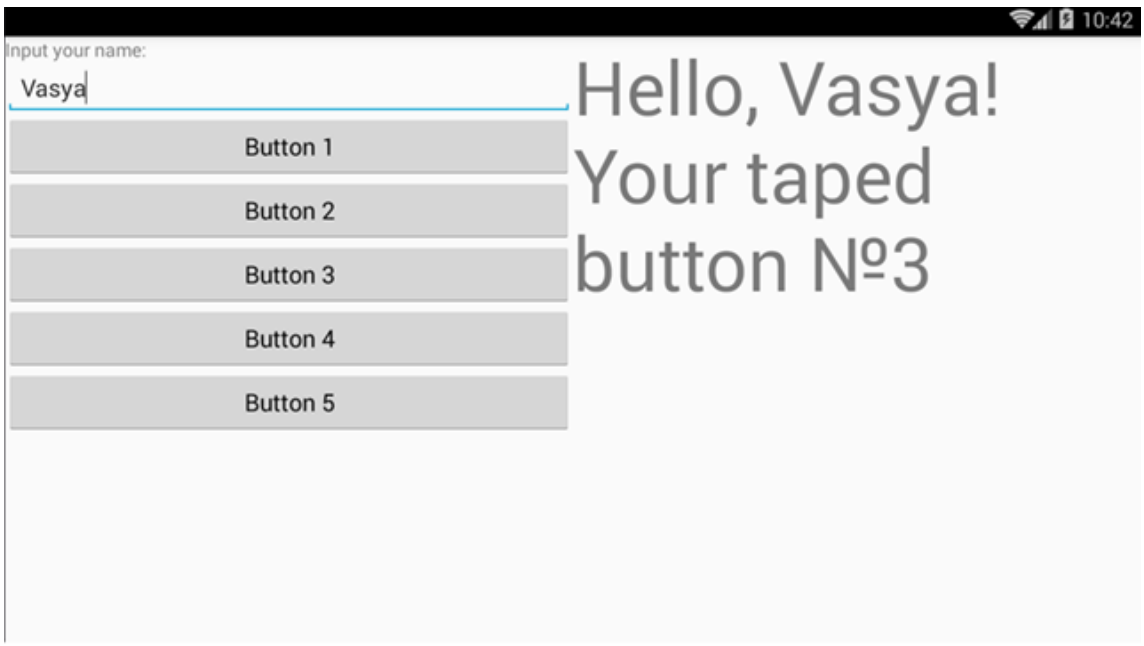


Рис. 3.24.

Полный код проекта можно посмотреть здесь.

3.6.5. Взаимодействия между фрагментами и активностями

Хотя Fragment реализован как объект, независимый от класса Activity, и может быть использован внутри нескольких активностей, конкретный экземпляр фрагмента напрямую связан с содержащей его активности.

В частности, фрагмент может обратиться к экземпляру Activity с помощью метода getActivity()и без труда выполнить такие задачи, как поиск экземпляра класса View внутри активности:

```
View listView = getActivity().findViewById(R.id.list);
```

Аналогичным образом активность может вызывать методы фрагмента, получив ссылку на объект Fragment от FragmentManager с помощью метода findFragmentById() или findFragmentByTag().

Например,

```
ExampleFragment fragment = (ExampleFragment) getFragmentManager().findFragmentById(R.id.ex
ample_fragment);
```

3.7. Двумерная графика в Android-приложениях

Сайт: IT Академия SAMSUNG
Курс: MDev @ IT Академия Samsung
Книга: 3.7. Двумерная графика в Android-приложениях
Напечатано:: Егор Беляев
Дата: Суббота, 18 Апрель 2020, 19:25

Оглавление

3.7.1. Основы графики в Android

Изучение техники рисования под Android начнем с создания нового проекта. Создадим проект и назовем его AndroidPaint.

Работая в операционной системе Android с различными программами, пользователь взаимодействует с ними с помощью кнопок, выпадающих списков и других элементов интерфейса. Эти элементы интерфейса являются объектами, созданными на основе разновидностей класса View. Для проведения графических экспериментов в нашем проекте мы создадим новый элемент интерфейса, который будет наследоваться от класса View.

Создадим новый класс MyDraw и унаследуем его от View.

Для того чтобы можно было рисовать на поверхности нашего элемента, переопределим метод onDraw в созданном классе. Вызывать его вручную не нужно. Этот метод относится к методам обратного вызова, его будет вызывать операционная система, когда необходимо. Метод onDraw вызывается операционной системой при старте активности, чтобы нарисовать элемент интерфейса на экране. Нам нужно описать только действия, которые должен выполнять onDraw, то есть что и в какой последовательности рисовать на поверхности элемента.

Код получившейся заготовки класса MyDraw должен выглядеть так:

```
public class MyDraw extends View{
        public MyDraw (Context context) {
                super(context);
        }

        @Override
        protected void onDraw(Canvas canvas){
                super.onDraw(canvas);
        }
}
```

Мы унаследовали класс MyDraw от класса android.view. View и переопределили метод onDraw(Canvas canvas). Все строчки, которые будут выделяться волнистой линией, нужно проверить, а также импортировать необходимые пакеты самостоятельно, если необходимо. Импорт удобно проводить с помощью сочетания клавиш Ctrl + Shift + O.

Далее создадим экземпляр класса MyDraw и разместим его на активности. Откроем файл с описанием класса активности и немного изменим код внутри метода onCreate:

```
public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        requestWindowFeature(Window.FEATURE_NO_TITLE);
        setContentView(new MyDraw(this));
        //setContentView(R.layout.activity_main);
}
```

Методы обратного вызова часто используются в программировании под Android. Например, чтобы разместить наш виджет на активности (экране приложения), в методе обратного вызова onCreate мы разместили код:

```
setContentView(new MyDraw(this));
```

Сам метод onCreate вызывать нигде не нужно. Система его вызовет сама при создании активности.

Обратите внимание, что строчка с установкой макета activity_main закомментирована. Для того чтобы избавиться от рамки с заголовком в активности, мы также добавили вызов метода requestWindowFeature(Window. FEATURE_NO_TITLE), но это делать вовсе не обязательно.

На этом подготовительные работы завершены. Дальнейший код мы будем писать в созданном классе MyDraw. Посмотрите на класс MyDraw. При вызове в метод onDraw передается объект класса Canvas — это холст, на котором мы будет рисовать. На объекте Canvas введена декартова система координат, левый верхний угол соответствует точке (0;0), ось **Y** направлена вниз, ось **X** — вправо. Размер области рисования можно узнать вызовом методов getWidth() и getHeight(). С помощью различных методов класса мы можем рисовать линии, окружности, дуги и так далее.

Прежде чем что-то рисовать, нужно определить некоторые параметры рисования, такие как цвет, толщина и способ рисования (рисование контуров или заливка фигуры). Сам процесс рисования во многом аналогичен рисованию на бумаге — здесь в нашем распоряжении есть программные аналоги цветных карандашей, кисти, линейки, циркуля и т. п. Например, мы можем взять толстый зеленый карандаш и нарисовать им линию, затем взять циркуль с тонким красным карандашом и нарисовать окружность. Эти действия в правильном порядке необходимо прописать в коде при рисовании.

Параметры рисования определяются в объекте класса Paint. Он содержит стили, цвета и другую графическую информацию для рисования графических объектов. С его помощью можно выбирать способ отображения графических примитивов, которые можно рисовать на объекте Canvas. Например, чтобы установить сплошной цвет для рисования линии, нужно вызвать метод setColor(). Но для этого нужно сначала создать объект класса Paint.

```
 Paint paint = new Paint();
paint.setColor(Color.RED); // Выбираем инструмент красного цвета
```

Объект Paint может вести себя как карандаш, рисуя контуры фигуры, или как кисть в графических редакторах, закрашивая фигуры цветом. За это поведение отвечает метод setStyle().

Очертания графического примитива можно рисовать, используя вызов setStyle() с параметрами (табл. 3.11).

Стиль	Описание
Paint.Style.FILL	Рисование с заливкой
Paint.Style.STROKE	Рисование только контура
Paint.Style.FILL_AND_STROKE	Рисование контура и заливки

Табл. 3.11.

Для рисования качественной графики можно вызывать методы, обеспечивающие сглаживание пикселей:

```
 Paint paint = new Paint();
paint.setSubpixelText(true); // Субпиксельное сглаживание для текста
paint.setAntiAlias(true); // Антиальясинг, сглаживание  линий
```

Для представления цвета в Android обычно используют 32-битное целое число, а не экземпляры класса Color. Для задания цвета можно использовать статические константы класса Color, например:

```
 int blue = Color.BLUE;
Paint paint = new Paint();
paint.setColor(blue);
```

Такие константы описаны только для самых основных цветов. Цвет также можно описать с помощью четырех восьмибитных чисел в формате ARGB, по одному для каждого канала (Alpha, Red, Green, Blue). Напомним, что каждое восьмибитное целое беззнаковое число может принимать значения от 0 до 255. Получить нужный цвет из набора цветовых компонентов можно через методы rgb() или argb(), которые возвращают целые значения (код цвета).

```
 // Полупрозрачный синий
int myTransparentBlue = Color.argb(127, 0, 0, 255);
```

Метод parseColor() позволяет получить целочисленное значение из шестнадцатеричной формы:

```
 int pColor = Color.parseColor("#FF0000FF");
```

Для большей гибкости приложения цвета помещают в ресурсы. В этом случае их можно будет менять без вмешательства в код программы:

```
 <?xml version="1.0" encoding="utf-8"?>
<resources>

    <color name="ballColor" >#FF00FFFF
    </color>

</resources>
```

В коде программы можно обратиться к цвету следующим образом:

```
 int ballColor = getResources().getColor(R.color.ballColor);
```

Имя для цвета в файле ресурсов желательно давать описательное, соответствующее тому, для чего этот цвет применяется. Например, backgroundColor. Имена с названиями цветов вроде red или blue использовать не рекомендуется, в этом случае при изменении кода цвета можно потерять смысл в его имени.

Прежде чем приступить к экспериментам, приведем основные функции класса Canvas для рисования графических примитивов. В таблице 3.12 приведены основные методы для рисования с помощью Canvas.

Метод	Назначение
drawARGB()/drawRGB()/drawColor()	Заполняет холст сплошным цветом

Метод	Назначение
drawArc()	Рисует дугу между двумя углами внутри заданной прямоугольной области
drawBitmap()	Рисует растровое изображение на холсте
drawCircle()	Рисует окружность с определенным радиусом вокруг заданной точки
drawLine(s)()	Рисует линию (или последовательность линий) между двумя точками
drawOval()	Рисует овал на основе прямоугольной области
drawPaint()	Закрашивает весь холст с помощью заданного объекта Paint
drawPath()	Рисует указанный контур, используется для хранения набора графических примитивов в виде единого объекта
drawPicture()	Рисует объект Picture внутри заданного прямоугольника
drawPoint()	Рисует точку в заданном месте
drawRect()	Рисует прямоугольник
drawText()	Рисует текстовую строку на холсте
rotate() и restore()	Вращение холста
scale() и translate()	Изменение и перемещение координатной системы

Табл. 3.12.

Более подробно с инструментами класса Canvas можно ознакомиться в официальной документации.

Пример 3.7

Как было сказано выше, вся работа с графикой происходит в методе onDraw() класса. Для начала установим цвет холста, на котором будем рисовать, пусть это будет белый цвет. При желании можно установить любой другой.

Для этого сразу после строчки super.onDraw(canvas) напишем код:

```
 Paint paint = new Paint();
// Выбираем кисть
paint.setStyle(Paint.Style.FILL);
// Белый цвет кисти
paint.setColor(Color.WHITE);
// Закрашиваем холст
canvas.drawPaint(paint);
```

Далее мы будем рисовать на этом «подготовленном» холсте. Следуя описанному выше принципу, перед каждым рисованием будем указывать настройки инструмента.

Например, для того чтобы нарисовать сглаженный красный круг в центре экрана, напишем:

```
 // Включаем антиальясинг
paint.setAntiAlias(true);
paint.setColor(Color.argb(127,0,0,255));
// Полупрозрачный синий круг радиусом 100 пикселей в центре экрана
canvas.drawCircle(getWidth() / 2, getHeight() / 2, 100, paint);
```

Для рисования прямоугольника аналогично нужно указать координаты и инструмент:

```
 // Синий прямоугольник вверху экрана
paint.setColor(Color.BLUE);
canvas.drawRect(0, 0, getWidth(),200, paint);
```

Выведем текст на экран:

```
 paint.setColor(Color.WHITE);
paint.setStyle(Paint.Style.FILL);
paint.setTextSize(30);
canvas.drawText("Samsung IT school", 50, 100, paint);
```

Выведем текст под углом. Для этого сначала надо повернуть холст:

```
 // текст под углом
float rotate_center_x = 200; //центр поворота холста по оси X
float rotate_center_y = 200; // центр поворота холста по оси Y
float rotate_angle = 45; //угол поворота

// поворачиваем холст
canvas.rotate(-rotate_angle, rotate_center_x, rotate_center_y);



paint.setColor(Color.BLACK);
paint.setTextSize(40);

canvas.drawText("Samsung IT school",0,450,paint);

// возвращаем холст на прежний угол
canvas.rotate(rotate_angle, rotate_center_x, rotate_center_y);
```

Таким же образом можно рисовать под углом и другие фигуры.

На экране можно рисовать готовые изображения из файлов. Хоть Android и поддерживает распространенные форматы изображений, рекомендуемым является формат PNG. Поместим логотип компании Samsung с именем samsung в папку *res\drawable* как графический ресурс приложения и выведем его в правом нижнем углу (рис. 3.25).

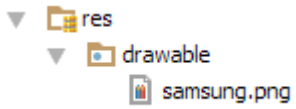


Рис. 3.25.

```
 // Выводим изображение логотипа Samsung на экран в правом нижнем углу экрана
Bitmap image = BitmapFactory.decodeResource(getResources(), R.drawable.samsung);
int xx = canvas.getWidth(), yy = canvas.getHeight();
canvas.drawBitmap(image, xx - image.getWidth(), yy - image.getHeight(), paint);
```

В результате, если мы запустим приложение, то увидим следующее (рис. 3.26).

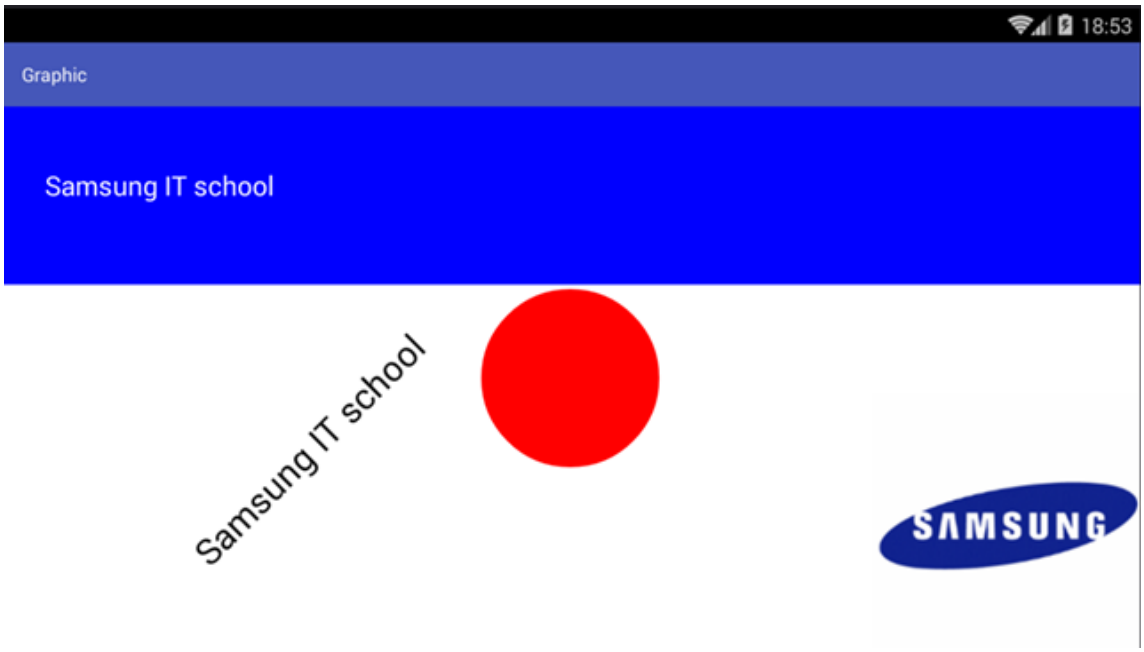


Рис. 3.26.

Полностью код класса MyDraw выглядит следующим образом:

```
public class MyDraw extends View{

   public MyDraw(Context context) {
       super(context);
   }

   @Override
   protected void onDraw(Canvas canvas) {
       super.onDraw(canvas);
       Paint paint = new Paint();
       // Выбираем кисть
       paint.setStyle(Paint.Style.FILL);
       // Белый цвет кисти
       paint.setColor(Color.WHITE);
       // Закрашиваем холст
       canvas.drawPaint(paint);

       // Включаем антиальясинг
       paint.setAntiAlias(true);
       paint.setColor(Color.argb(127,0,0,255));
       // Полупрозрачный синий круг радиусом 100 пикселей в центре экрана
       canvas.drawCircle(getWidth() / 2, getHeight() / 2, 100, paint);

       // Синий прямоугольник вверху экрана
       paint.setColor(Color.BLUE);
       canvas.drawRect(0, 0, getWidth(),200, paint);

       // Текст
       paint.setColor(Color.WHITE);
       paint.setStyle(Paint.Style.FILL);
       paint.setTextSize(30);
       canvas.drawText("Samsung IT school", 50, 100, paint);

       // текст под углом
       float rotate_center_x = 200; //центр поворота по оси X
       float rotate_center_y = 200; // центр поворота по оси Y
       float rotate_angle = 45; //угол поворота

       // поворачиваем холст
       canvas.rotate(-rotate_angle, rotate_center_x, rotate_center_y);

       paint.setColor(Color.BLACK);
       paint.setTextSize(40);

       canvas.drawText("Samsung IT school",0,450,paint);

       // возвращаем холст на прежний угол
       canvas.rotate(rotate_angle, rotate_center_x, rotate_center_y);

       // Выведем изображение логотипа Samsung на экран в правом нижнем углу экрана
       Bitmap image = BitmapFactory.decodeResource(getResources(), R.drawable.samsung);
       int xx = canvas.getWidth(), yy = canvas.getHeight();
       canvas.drawBitmap(image, xx - image.getWidth(), yy - image.getHeight(), paint);
```

```
    }
}
```

Проект разобранного примера находится здесь.

3.7.2. Игра «Забавные птички»

Смысл игры

Давайте напишем игру со следующим смыслом. Вы управляете птицей, спешащей домой к себе в гнездо. Ваша задача уворачиваться от пролетающих птиц и набирать очки.

Игровой процесс

Птица игрока может летать вверх или вниз относительно экрана. При соприкосновении со стенками птица отталкивается от них и летит в противоположную сторону. Игрок может управлять птицей, осуществляя прикосновения выше и ниже положения птицы на экране.

За облет птицы противника назначаются очки. Соприкосновение с птицей противника, а также смена направления полета приводит к уменьшению очков.

(В игре использованы графические ресурсы со страницы сайта http://www.xojo3d.com/tut015.php)

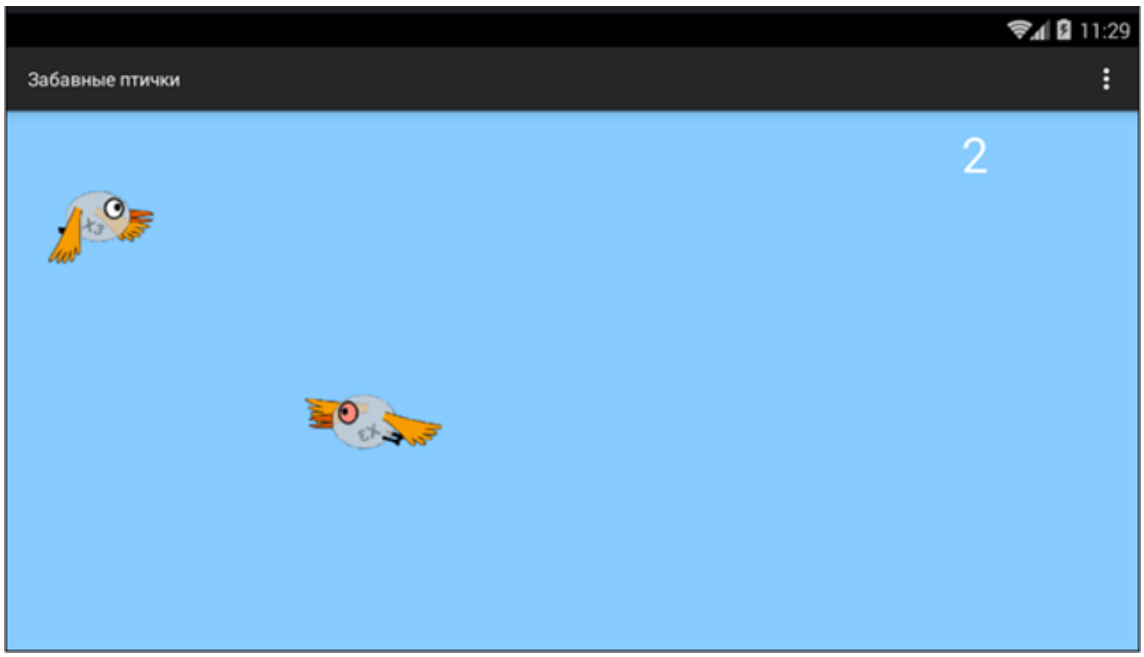


Рис. 3.27.

Создадим проект как обычно и назовем его «Забавные птички».

3.7.3. Игра «Забавные птички». Игровое поле

Создание класса GameView

Для управления игрой — реализации игровой логики, отображения игры на экране и взаимодействия с пользователем — разработаем класс GameView. Объект этого класса будет служить игровым полем для игры «Забавные птички».

За основу класса GameView возьмем стандартный класс View, который является базовым классом компонентов интерфейса пользователя в Android.

1. Создадим новый класс с именем GameView – ***File –>New –> Java Class***
2. В качестве суперкласса для созданного GameView укажем View.
   Добавим конструктор, который принимает единственный параметр — контекст.

```
public class GameView extends View{

        public GameView(Context context) {
                super(context);
        }
}
```

3. Переопределим метод onSizeChanged для определения размеров игрового поля:

```
@Override
protected void onSizeChanged(int w, int h, int oldw, int oldh) {
        super.onSizeChanged(w, h, oldw, oldh);

        viewWidth = w;
        viewHeight = h;
}
```

   Переменные viewWidth и viewHeight объявим как поля класса GameView — они будут содержать актуальные размеры игрового поля:

```
private int viewWidth;
private int viewHeight;
```

4. Создадим в GameView поле points для хранения очков, набранных игроком.

```
private int points = 0;
```

5. Чтобы можно было рисовать на поверхности компонента, переопределим метод onDraw() в созданном классе.
   Метод onDraw() является методом обратного вызова, его будет вызывать операционная система тогда, когда будет нужно перерисовать компонент (например, при старте активности). Вручную onDraw() вызывать не нужно.
   Внутрь метода onDraw() передается объект типа Canvas — холст, на котором можно рисовать, вызывая методы этого объекта. Важно будет определить правильную последовательность рисования.

```
 @Override
protected void onDraw(Canvas canvas) {
        super.onDraw(canvas);
}
```

6. Нарисуем фон и количество очков на поверхности компонента:

```
 @Override
protected void onDraw(Canvas canvas) {

        super.onDraw(canvas);

        canvas.drawARGB(250, 127, 199, 255); // заливаем цветом
        Paint p = new Paint();
        p.setAntiAlias(true);
        p.setTextSize(55.0f);
        p.setColor(Color.WHITE);
        canvas.drawText(points+"", viewWidth - 100, 70, p);
}
```

Добавление компонента GameView в активность

Активность не будет содержать разметку, полученную из xml-файла. Единственным элементом, который будет содержаться в активности, будет экземпляр класса GameView (наше игровое поле).

Откроем файл MainActivity с описанием класса активности и немного изменим код внутри метода onCreate.

Вместо существующей разметки добавим на активность разработанный компонент GameView.

```
 protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        //setContentView(R.layout.activity_main);
        setContentView(new GameView(this));
}
```

3.7.4. Игра «Забавные птички». Создание класса Sprite для управления анимацией

Класс Sprite будет являться базовым классом для всех игровых объектов. Он будет поддерживать перемещение объектов на экране, покадровую анимацию и определять столкновения объектов. Основой класса является картинка, содержащая множество кадров с различным состоянием игрового объекта (персонажа). Ниже показана картинка (рис. 3.28), которая будет использоваться для создания анимации главного героя — птички, спешащей домой.

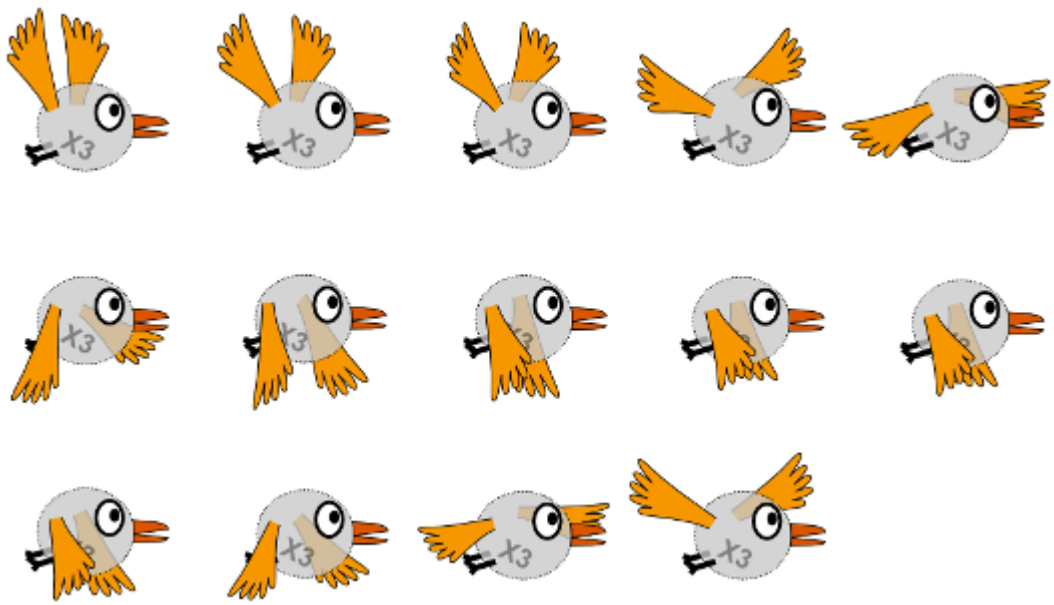


Рис. 3.28.

1. Создание нового класса Sprite. В пакете ru.samsung.itschool.book.funnybirds создайте новый класс с именем Sprite — ***File –>New –> Java Class***.
2. Добавим в класс следующие поля:

```
 private Bitmap bitmap;
private List<Rect> frames;
private int frameWidth;
private int frameHeight;
private int currentFrame;
private double frameTime;
private double timeForCurrentFrame;

private double x;
private double y;

private double velocityX;
private double velocityY;

private int padding;
```

Импортируем используемые классы Bitmap и Rect (Alt + Enter).
Назначения полей приведены в таблице 3.13.

Переменная	Назначение

Переменная	Назначение
bitmap	Ссылка на изображение с набором кадров
frames	Коллекция прямоугольных областей на изображении — кадры, которые участвуют в анимационной последовательности
currentFrame	Номер текущего кадра в коллекции frames
frameTime	Время, отведенное на отображение каждого кадра анимационной последовательности
timeForCurrentFrame	Текущее время отображения кадра. Номер текущего кадра currentFrame меняется на следующий при достижении переменной timeForCurrentFrame значения из frameTime
frameWidth и frameHeight	Ширина и высоту кадра для отображения на экране
x и y	Местоположение левого верхнего угла спрайта на экране
velocityX и velocityY	Скорости перемещения спрайта по оси X и Y соответственно
Padding	Внутренний отступ от границ спрайта, необходимый для более точного определения пересечений спрайтов

Табл. 3.13.

3. Создайте конструктор класса Sprite.

```
 public Sprite(double x, double y, double velocityX, double velocityY,  Rect initialF
rame, Bitmap bitmap){

        this.x = x;
        this.y = y;
        this.velocityX = velocityX;
        this.velocityY = velocityY;
        this.bitmap = bitmap;
        this.frames = new ArrayList<Rect>();
        this.frames.add(initialFrame);
        this.bitmap = bitmap;
        this.timeForCurrentFrame = 0.0;
        this.frameTime = 0.1;
        this.currentFrame = 0;
        this.frameWidth = initialFrame.width();
        this.frameHeight = initialFrame.height();
        this.padding = 20;
}
```

4. Создадим геттеры и сеттеры для полей класса.
   Поскольку поля класса должны быть скрыты от доступа из вне, для доступа к ним создадим методы — геттеры (для получения значений) и сеттеры (для установки значений).
   Большинство методов можно сгенерировать автоматически ***Code -> Generate… -> Getter and Setter***.

```
 public void setX(double x) {
        this.x = x;
}

public double getY() {
        return y;
}

public void setY(double y) {
        this.y = y;
}

public int getFrameWidth() {
        return frameWidth;
}

public void setFrameWidth(int frameWidth) {
        this.frameWidth = frameWidth;
}

public int getFrameHeight() {
        return frameHeight;
}

public void setFrameHeight(int frameHeight) {
        this.frameHeight = frameHeight;
}

public double getVx() {
        return velocityX;
}

public void setVx(double velocityX) {
        this.velocityX = velocityX;
}

public double getVy() {
    return velocityY;
}

public void setVy(double velocityY) {
        this.velocityY = velocityY;
}

public int getCurrentFrame() {
        return currentFrame;
}

public void setCurrentFrame(int currentFrame) {
        this.currentFrame = currentFrame%frames.size();
}

public double getFrameTime() {
```

```
    return frameTime;
}

public void setFrameTime(double frameTime) {
        this.frameTime = Math.abs(frameTime);
}

public double getTimeForCurrentFrame() {
        return timeForCurrentFrame;
}

public void setTimeForCurrentFrame(double timeForCurrentFrame) {
        this.timeForCurrentFrame = Math.abs(timeForCurrentFrame);
}

public int getFramesCount () {
        return frames.size();
}
```

5. Напишем метод добавления кадров в анимационную последовательность.

```
 public void addFrame (Rect frame) {
        frames.add(frame);
}
```

Добавление кадров в последовательность анимации заключается в добавлении соответствующей кадру прямоугольной области на изображении, заданной с помощью объекта Rect (прямоугольник).

6. Добавим метод update() для обновления внутреннего состояния спрайта.

```
 public void update (int ms) {

        timeForCurrentFrame += ms;

        if (timeForCurrentFrame >= frameTime) {
                currentFrame = (currentFrame + 1) % frames.size();
                timeForCurrentFrame = timeForCurrentFrame - frameTime;
        }

        x = x + velocityX * ms/1000.0;
        y = y + velocityY * ms/1000.0;
}
```

Метод update() вызывается таймером с определенной периодичностью. Внутрь метода передается время в миллисекундах, прошедшее с момента последнего вызова этого метода. Время используется для изменения состояния спрайта — его перемещения в пространстве за это время, а также рассчитывается необходимость перехода к следующему кадру.
В update() реализуется механика перебора кадров изображения на основании текущего времени воспроизведения кадра и его сравнения с необходимым временем воспроизведения одного кадра. В зависимости от времени и скорости по осям

X и Y осуществляется изменение координат спрайта на экране.
Переход к следующему кадру и зацикливание (переход от последнего к нулевому) осуществляется здесь:

```
currentFrame = (currentFrame + 1) % frames.size();
```

7. Создайте метод draw() для рисования спрайта на холсте.

```
public void draw (Canvas canvas) {
       Paint p = new Paint();
       Rect destination = new Rect((int)x, (int)y, (int)(x + frameWidth), (int)(y +
frameHeight));
       canvas.drawBitmap(bitmap, frames.get(currentFrame), destination,  p);
}
```

Метод draw() рисует на переданном холсте текущий кадр frames.get(currentFrame) в области экрана, заданной в прямоугольнике destination.

8. Добавим метод определения пересечения спрайтов.
Когда два объекта нашей игры взаимодействуют друг с другом, нам нужно знать, сталкиваются ли они, и в зависимости от этого предпринимать определенные действия. Поскольку спрайты могут содержать небольшую пустую область вокруг себя, для определения столкновений лучше использовать прямоугольные области, отступающие внутрь от границ спрайта на величину padding.
Метод, возвращающий прямоугольную область, участвующую в определении столкновений:

```
public Rect getBoundingBoxRect () {
   return new Rect((int)x+padding, (int)y+padding, (int)(x + frameWidth - 2 *paddin
g),
                                    (int)(y + frameHeight - 2* padding));
}
```

Метод определения пересечения спрайтов:

```
public boolean intersect (Sprite s) {
       return getBoundingBoxRect().intersect(s.getBoundingBoxRect());
}
```

Нет наложения спрайтов, нет пересечения (рис. 3.29).

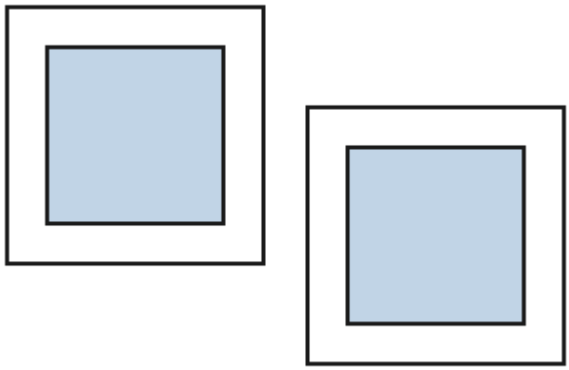


Рис. 3.29.

Есть наложения спрайтов, но нет пересечения (рис. 3.30).

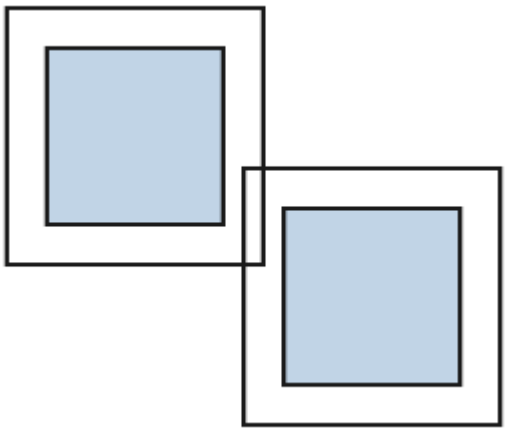


Рис. 3.30.

Есть наложение спрайтов, есть пересечение (рис. 3.31).

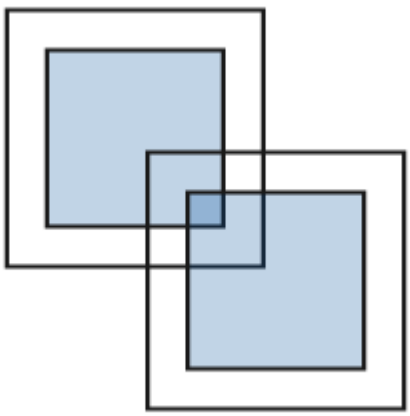


Рис. 3.31.

3.7.5. Игра «Забавные птички». Создание птицы

Создание птицы, управляемой пользователем

1. Добавим в папку **res/drawable** изображение птицы, состоящей из нескольких кадров player.png
   Изображение можно добавить, используя буфер обмена: скопировать файл в проводнике и вставить в конечную папку (рис. 3.32).

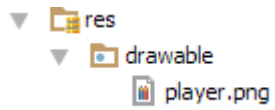


Рис. 3.32.

2. Объявим переменную playerBird типа Sprite как поле класса GameView.

```
private Sprite playerBird;
```

3. Создадим объект класса Sprite в конструкторе GameView.
   Для создания объекта конструктору класса Sprite нужно передать исходные координаты объекта на экране (10,0), проекции скорости на ось **X** и **Y** — 0 и 200 соответственно, а также изображение с кадрами и прямоугольник, описывающий границы первого кадра.
   Содержимое конструктора GameView:

```
public GameView(Context context) {

        super(context);
        Bitmap b = BitmapFactory.decodeResource(getResources(), R.drawable.player);

        int w = b.getWidth()/5;
        int h = b.getHeight()/3;

        Rect firstFrame = new Rect(0, 0, w, h);

        playerBird = new Sprite(10, 0, 0, 100, firstFrame, b);
}
```

4. Нарисуем птицу на экране. Для отрисовки спрайта птицы на компоненте GameView, вызовите у спрайта метод draw:

```
protected void onDraw(Canvas canvas) {
        super.onDraw(canvas);
        canvas.drawARGB(250, 127, 199, 255); // цвет фона
        playerBird.draw(canvas);
}
```

Добавление анимации у Птицы

1. Обновление игровой модели.
   В классе GameView определим метод update(), в котором будет происходить изменение игрового состояния (модели игры). Например, в этом методе будут происходить обновления всех спрайтов в игре — вызов их метода update().
   Вызов данного метода удобно осуществлять автоматически через определенные промежутки времени с помощью таймера.

Определим промежуток времени, за который должно происходить изменение игровой модели. Добавьте константу timerInterval как поле класса GameView.

```
 private final int timerInterval = 30;
```

В методе update() обновим состояние спрайта с птицей и перерисуем GameView:

```
 protected void update () {
        playerBird.update(timerInterval);
        invalidate();
}
```

2. Добавление класса таймера в класс GameView.
   Таймер под Android в Java можно реализовать при помощи класса-наследника CountDownTimer.
   В реализуемом классе необходимо переопределить абстрактные методы onTick() и onFinish(). В методе onTick() указываются действия, которые нужно делать периодически, например, вызов update() у спрайта и перерисовки GameView — invalidate(). В методе onFinish() — действия, когда таймер заканчивает свою работу.
   Определите класс Timer непосредственно внутри GameView для удобного вызова методов класса GameView. В конструкторе класса укажите общее время работы таймера — Integer. MAX_VALUE и время периодического срабатывания — timerInterval. При срабатывании таймера onTick() вызовите метод update() класса GameView.

```
 class Timer extends CountDownTimer {
        public Timer() {
                super(Integer.MAX_VALUE, timerInterval);
        }

        @Override
        public void onTick(long millisUntilFinished) {
                update ();
        }

        @Override
        public void onFinish() {
        }
}
```

3. Создадим и запустим таймер в самом конце конструктора GameView после создания всех спрайтов.

```
 Timer t = new Timer();
t.start();
```

Фрагмент экрана с игрой после запуска — птица падает вниз (рис. 3.33).

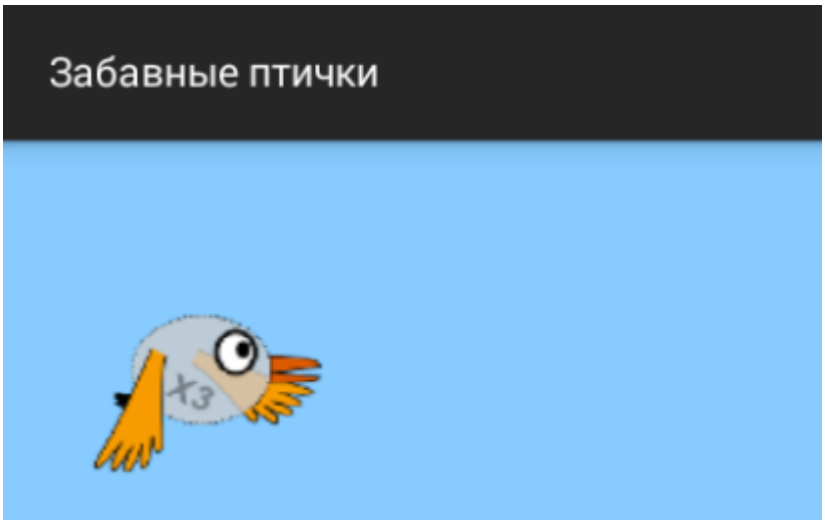


Рис. 3.33.

4. Добавим больше кадров с изображением птицы. После создания спрайта птицы в конструкторе GameView добавьте следующий фрагмент:

```
for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 4; j++) {
                if (i == 0 && j == 0) {
                        continue;
                }
                if (i == 2 && j == 3) {
                        continue;
                }
                playerBird.addFrame(new Rect(j * w, i * h, j * w + w, i * w + w));
        }
}
```

Теперь птица сможет махать крыльями.

3.7.6. Игра «Забавные птички». Добавление противника и контроль столкновений

Добавление противника

Во многом процесс добавления противника — птицы, летящей на встречу игроку, аналогичен уже рассмотренному.

1. Добавим в папку **res/drawable** изображение летящей на встречу птицы enemy.png (см. рис. 3.34).

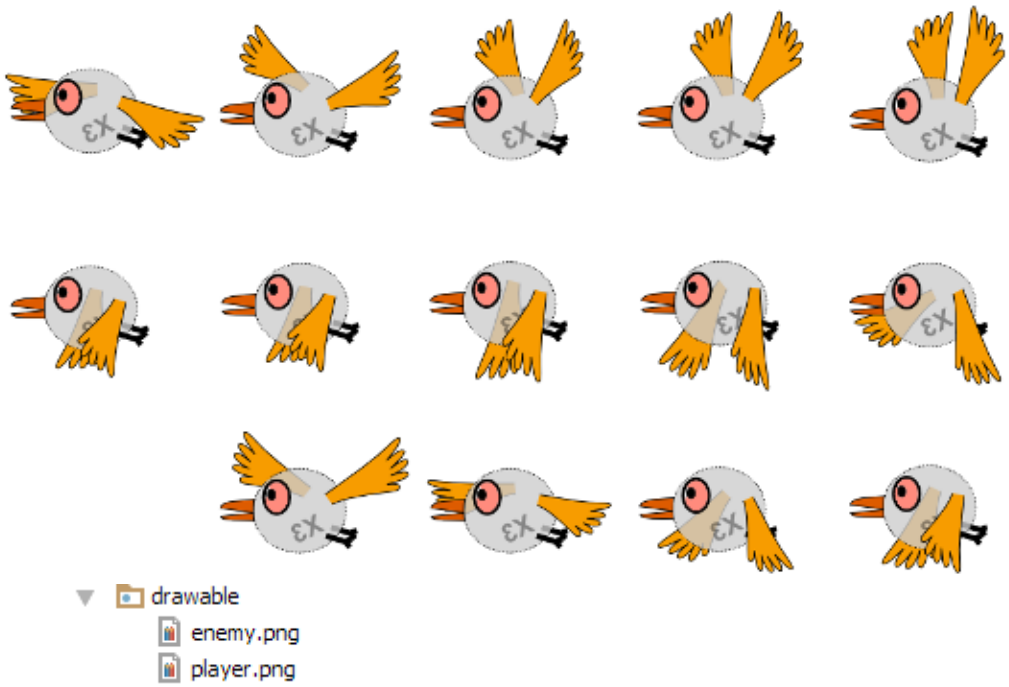


Рис. 3.34.

2. Создадим поле enemyBird в классе GameView.

```
private Sprite enemyBird;
```

3. Напишем в конструкторе GameView создание спрайта и добавление в него кадров: b = BitmapFactory.decodeResource(getResources(), R.drawable.enemy;

```
 w = b.getWidth()/5;
h = b.getHeight()/3;

firstFrame = new Rect(4*w, 0, 5*w, h);

enemyBird = new Sprite(2000, 250, -300, 0, firstFrame, b);

for (int i = 0; i < 3; i++) {
        for (int j = 4; j >= 0; j--) {
                if (i ==0 && j == 4) {
                        continue;
                }
                if (i ==2 && j == 0) {
                        continue;
                }
                enemyBird.addFrame(new Rect(j*w, i*h, j*w+w, i*w+w));
        }
}
```

4. Нарисуем спрайт в методе onDraw() класса GameView:

```
 enemyBird.draw(canvas);
```

5. Изменим состояние спрайта в методе onUpdate() класса GameView:

```
 enemyBird.update(timerInterval);
```

Управление птицей и контроль столкновений

Пользователь может направлять движение птицы. Прикосновением пальца выше спрайта птицы — подняться выше, прикосновением ниже спрайта — опуститься ниже.

1. Переопределим метод onTouchEvent () класса GameView:

```
 @Override
public boolean onTouchEvent(MotionEvent event) {

        int eventAction = event.getAction();

        if (eventAction == MotionEvent.ACTION_DOWN)  {
                // Движение вверх
                if (event.getY() < playerBird.getBoundingBoxRect().top) {
                        playerBird.setVy(-100);
                        points--;
                }
                else if (event.getY() > (playerBird.getBoundingBoxRect().bottom)) {
                        playerBird.setVy(100);
                        points--;
                }
        }
        return true;
}
```

При смене птицей направления полета расходуются очки:

```
 points--;
```

2. Напишем код, который не позволит птице игрока вылететь за пределы экрана. Птица должна менять направление полета при соударении с верхней и нижней горизонтальной линией экрана. Внутри метода update() после обновления состояния спрайтов:

```
 if (playerBird.getY() + playerBird.getFrameHeight() > viewHeight) {
        playerBird.setY(viewHeight - playerBird.getFrameHeight());
        playerBird.setVy(-playerBird.getVy());
        points--;
}
else if (playerBird.getY() < 0) {
        playerBird.setY(0);
        playerBird.setVy(-playerBird.getVy());
        points--;
}
```

3. Добавим метод возвращения птицы противника после пролета:

```
 private void teleportEnemy () {
        enemyBird.setX(viewWidth + Math.random() * 500);
        enemyBird.setY(Math.random() * (viewHeight - enemyBird.getFrameHeight()));
}
```

4. Возвращение птицы противника в начальное положение осуществляется после пролета игрока. Внутри метода update() напишем:

```
 if (enemyBird.getX() < - enemyBird.getFrameWidth()) {
        teleportEnemy ();
        points +=10;
}
```

За облет птицы игроку начисляются очки.

5. Возвращение птицы противника в начальное положение осуществляется также после столкновения с птицей игрока. Внутри метода update()напишем:

```
 // Проверка столкновений
if (enemyBird.intersect(playerBird)) {
        teleportEnemy ();
        points -= 40;
}
```

За столкновения с птицей у игрока снимаются очки. Полный код получившегося проекта можно посмотреть здесь.

3.8. Разработка игровых приложений. SurfaceView

Сайт: IT Академия SAMSUNG
Курс: MDev @ IT Академия Samsung
Книга: 3.8. Разработка игровых приложений. SurfaceView
Напечатано:: Егор Беляев
Дата: Суббота, 18 Апрель 2020, 19:25

Оглавление

3.8.1. Общие подходы для реализации игровых приложений

Компьютерные игры — один из самых популярных способов использования компьютеров, смартфонов и планшетов.

Первые игры отличались простотой графики и программной логики. Но со временем игры становились все сложнее. Над их созданием работал уже не один программист, а целый коллектив разработчиков — программисты, дизайнеры, художники, аниматоры, специалисты по звуковым эффектам и люди других профессий. Для обеспечения эффективности разработки, ее ускорения и повышения качества игрового продукта в середине 90-х годов появился новый вид программного обеспечения для разработчиков — игровой движок.

Последовательные этапы проектирования и реализации игрового приложения

В проектировании игрового приложения можно выделить некоторые этапы.

- ***Идея.*** Прежде всего, разработка игры начинается с идеи. Она может возникнуть случайно или в процессе долгого перебора. Можно отталкиваться от любимого жанра или от жанра, который популярен на данный момент. Возможно, ваша идея будет заимствована у другой игры или, напротив, представляет нечто совершенно новое.
- ***Исследование.*** Иногда для разработки игры нужно выяснить некоторые делали. Если вы разрабатываете симулятор спортивной игры, вам нужно подробно ознакомиться с ее правилами. Решили делать игру про пиратов или рыцарей? Не поленитесь подробно изучить тему. Скорее всего, в процессе найдется что-то новое. Это поможет привнести в игру оригинальные находки в сюжет (если он присутствует) или игровой процесс. Если в вашей игре вымышленный мир, то самое время подумать, по каким правилам он существует.
- ***Разработка геймплея.*** На этом этапе начинается проработка игрового процесса. Если на этапе идеи хватило формулировки «по реке плывет бегемот и уворачивается от препятствий», то сейчас нужно знать, что за препятствия встречаются бегемоту (камни, другие животные), как игрок будет им управлять (гироскоп, swipe-движения или касания по экрану), что будет с ним, если он столкнется с препятствием (игра начнется сначала или у бегемота есть несколько жизней), ну и, конечно, неплохо было бы знать, куда он плывет. Как можно увидеть, геймплей охватывает множество мелочей. Важно все хорошо продумать. Проработанный геймплей заинтересовывает игрока. Но нужно не перестараться, например, введя пол сотни видов врагов. После тщательной проработки геймлея, когда вы четко представляете, как ваша игра будет выглядеть для пользователя, можно приступать к следующему этапу.
- ***Проектирование архитектуры.*** Подходы к проектированию игровых приложений в целом не отличаются от остальных. Данная тема рассматривалась в главе 3.1. Главное на этом этапе — определиться с необходимыми классами, рассмотреть возможность применения наследования, абстракции. К примеру, если мир в игре случайно генерируемый, нужно проработать алгоритм генерации, если в игре много уровней, которые создаются вручную, возможно, стоит задуматься о написании редактора уровней. Нет уникального рецепта проектирования архитектуры, многое зависит от конкретного приложения.
- ***Реализация, тестирование.*** Этот этап включает в себя написание кода. Также сюда входят рисование графики, подбор звукового сопровождения, проектирование уровней. Другими словами, подготовка или создание игровых ресурсов и контента.

Существуют различные модели разработки программного обеспечения и, в частности, игровых приложений. Выделяют такие модели, как каскадную, итеративную, спиральную, V-модель разработки. К примеру, каскадная модель подразумевает, что разработка состоит из нескольких этапов и переход к следующему этапу осуществляется только после успешного выполнения предыдущего. То есть, до тех пор, пока проектирование полностью не закончено, нельзя приступать к написанию кода. (Нужно упомянуть, что данный подход устаревший и совершенно не гибкий).

Конечно, данные модели применяются в серьезных проектах и нет необходимости применять их в учебных приложениях.

- ***Распространение, эксплуатация.*** Для игровых мобильных приложений единственный путь распространения — это магазины приложений. Впервые эту идею в современном виде предложила компания Apple в 2008 году и с тех пор ни одна мобильная платформа не обходится без них.
  Например, до распространения интернета для десктопных приложений путь до клиента был дорогим и длинным: нужна была реклама, заводы наносили программы на дискеты, CD и DVD, а затем они рассылались по почте или продавались в магазинах. Сейчас разработчик выкладывает свою программу в соответствующий операционной системе магазин, и она становится доступной пользователям с любого уголка планеты, где есть доступ в интернет.
- ***Поддержка.*** Зачастую поддержку называют продолжающейся разработкой. Почему? Посмотрите, например, отзывы о программах на Google Play. Пользователи оставляют свои замечания, авторы объясняют работу программы, исправляют ошибки, выходят новые версии и т. д. Хотя, надо заметить, бывают случаи, когда приложение не предполагает поддержку, то есть разработчики по ряду причин могут прекратить работу над проектом.

Профессии в мире индустрии игр

Индустрия игр породила множество профессий. Во времена простых игр, когда игровая сфера только развивалась, в команде разработчиков вполне могло хватить одного геймдизайнера и программиста. Но современные проекты очень масштабны и поэтому требуют разделения труда. Программист не сможет заниматься всем сразу, разрабатывать искусственный интеллект и интерфейс, а также заниматься серверной частью. Для этого выделены специальности AI programmer, GUI programmer, Server programmer. Среди множества профессий игровой индустрии выделяются несколько основных.

- ***Геймдизайнер.*** Круг обязанностей геймдизайнера очень широк. Можно сказать, что он должен определить внешний вид и возможности игрового процесса.
- ***Программист.*** Среди программистов выделяют такие специализации как Core programmer, gui programmer, physics programmer, sound programmer и др.
- ***Дизайнер, художник.*** Профессии, связанные с разработкой интерфейса, графики для игры.
- ***Тестировщик*** ищет ошибки и следит за качеством продукта. Тестировщик моделирует различные ситуации, которые могут возникнуть в ходе игрового процесса и проверяет, все ли работает правильно. Это очень важный и трудоемкий этап. По статистике тестирование требует в 4 раза больше ресурсов, чем разработка.

3.8.2. Понятие игрового движка и его использование при разработке игры

Профессиональные разработчики игр с графикой, обрабатываемой в реальном времени, не пишут игры «с нуля». Они используют готовые игровые движки. ***Игровой движок*** — это центральный программный компонент приложения, который обеспечивает основные технологии и, зачастую, кроссплатформенность.

Изначально под игровым движком понимали подсистему обеспечения визуализации двумерной и, затем, трехмерной графики, создания анимации и визуальных эффектов. Позже движки стали обеспечивать поддержку физики игрового мира (физический движок), звука, сетевого взаимодействия, искусственного интеллекта.

Какие преимущества дает игровой движок разработчикам?

- Прежде всего, игровой движок создает каркас игрового приложения, делая код более организованным и управляемым. Не секрет, что современные игры содержат миллионы строчек кода. Продуманная архитектура игры, которая во многом обеспечивается движком, упрощает командную разработку, управление и поддержку кода из миллионов строк.
- Движок сокращает рутинные технические моменты реализации игровых процессов. Например, вместо изучения технических аспектов, обеспечивающих создание быстрой и плавной анимации, вызова множества библиотечных функций, можно использовать высокоуровневые функции движка, компактно решающие эту задачу.
- Движок позволяет сконцентрироваться на разработке игровой логики, позволяет мыслить более высокоуровневыми категориями.
- Особая ценность движка — переносимость, кроссплатформенность. Это, конечно, справедливо не для всех игровых движков. Но хорошо спроектированный игровой движок упрощает перенос игры на другие платформы.

Игровые движки — это сложные программы, которые невозможно освоить «с ходу». Поэтому и говорят, что это инструмент для профессионалов.

3.8.3. Класс SurfaceView

В предыдущей главе был подробно рассмотрен способ рисования на View с помощью класса Canvas. Получить доступ к объекту Canvas можно в переопределенном методе onDraw класса View. Отметим некоторые особенности рисования с помощью класса View:

- способ легок в реализации. Нужно только переопределить метод onDraw;
- данный способ подходит для отрисовки небольшого количества кадров, так как рисование происходит в основном потоке.

Таким образом, рисование на View больше подходит для создания собственного элемента интерфейса, который выполняет специальные задачи (например, изображает график, небольшую анимацию). Также можно применять для создания игр, которые не требуют частой перерисовки, а рисование кадра не требует сложных вычислений (такие как шахматы, шашки, викторины).

Для более сложных задач рисования используется класс SurfaceView, который является наследником класса View. SurfaceView, представляет собой выделенную поверхность для рисования внутри иерархии View. Главное его отличие от View в том, что рисование на этой поверхности возможно предоставить дополнительному потоку приложения. Соответственно, приложение не должно ждать, пока иерархия View будет готова к перерисовке.

Доступ к поверхности рисования в параллельном потоке происходит не напрямую, он может быть получен с помощью объекта SurfaceHolder. Получить этот объект можно вызовом метода getHolder(), после инициализации SurfaceView. Этот объект нужно передать в дополнительный поток, чтобы в нем получить доступ к Canvas вашего SurfaceView. Создание и удаление потока нужно синхронизировать с созданием и удалением SurfaceView. Для этого нужно использовать интерфейс SerfaceHolder. Callback, который содержит методы surfaceCreate(SurfaceHolder) и surfaceDestroy(SurfaceHolder), чтобы отслеживать события создания и уничтожения SurfaceView. Нужно сделать так, чтобы доступ к поверхности осуществлялся только между этими событиями. Также интерфейс SerfaceHolder.Callback содержит метод, который реагирует на изменения SurfaceView (см. табл. 3.14).

Метод	Описание
surfaceCreated (SurfaceHolder holder)	Метод вызывается сразу после создания поверхности
surfaceChanged (SurfaceHolder holder, int format, int width, int height)	Метод вызывается после структурных изменений (формата или размера). Как минимум вызывается один раз после вызова метода surfaceCreated
surfaceDestroyed (SurfaceHolder holder)	Метод вызывается перед уничтожением поверхности

Табл. 3.14.

3.8.4. Создание приложений с помощью SurfaceView

Перейдем к рассмотрению простого примера рисования с использованием класса SurfaceView.

Создание класса — наследника SurfaceView

Для начала нужно создать свой класс, унаследованный от класса SurfaceView с интерфейсом SurfaceHolder. Callback и переопределить один из конструкторов. Создадим класс DrawView. Объект SurfaceHolder осуществляет доступ к поверхности для рисования. Для него нужно вызвать метод addCallback(SurfaceHolder. Callback). Это говорит ему, что он должен получать обратные вызовы от методов объекта Callback. Другими словами, должен быть в курсе, когда создается, изменяется и уничтожается поверхность. Для того чтобы получить к нему доступ, нужно вызвать метод getHolder.

```
 import android.content.Context;
import android.view.SurfaceHolder;
import android.view.SurfaceView;

public class DrawView extends SurfaceView implements SurfaceHolder.Callback {
        public DrawView(Context context) {
                super(context);
                getHolder().addCallback(this);
        }

        @Override
        public void surfaceCreated(SurfaceHolder holder) {
                // создание SurfaceView
        }

        @Override
        public void surfaceChanged(SurfaceHolder holder, int format, int width, int height)
{
                // изменение размеров SurfaceView
        }

        @Override
        public void surfaceDestroyed(SurfaceHolder holder) {
                // уничтожение SurfaceView
        }
}
```

Создание отдельного потока для рисования

Для рисования нужно создать отдельный поток. Сделать это можно, создав класс, и унаследовав его от Thread. Рисование будет происходить в методе run(), который нужно переопределить. Поток должен постоянно работать до тех пор, пока SurfaceView будет показываться на экране. Если этого делать больше не нужно, поток должен как-то узнать о завершении своей работы. Как уже было сказано, доступ к поверхности для рисования хранит объект SurfaceHolder. Данный объект можно передать в класс DrawThread через его конструктор.

Итак, чтобы получить поверхность для рисования в дополнительном потоке, нужно вызвать метод lockCanvas() для SurfaceHolder. После этого на поверхности можно рисовать. После того, как необходимые методы рисования для Canvas были вызваны, нужно вызвать метод unlockCanvasAndPost() для объекта SurfaceHolder. После этого SurfaceView отрисует Canvas. При необходимости перерисовать кадр, нужно каждый раз вызывать эти методы lockCanvas() и unlockCanvasAndPost().

Метод requestStop() необходим для выхода из цикла, который находится внутри метода run(). Его вызов позволит в нужное время завершить поток.

```
public class DrawThread extends Thread {

        private SurfaceHolder surfaceHolder;

        private volatile boolean running = true;//флаг для остановки потока

        public DrawThread(Context context, SurfaceHolder surfaceHolder) {
                this.surfaceHolder = surfaceHolder;
        }

        public void requestStop() {
                running = false;
        }

        @Override
        public void run() {
                while (running) {
                        Canvas canvas = surfaceHolder.lockCanvas();
                        if (canvas != null) {
                                try {
                                        // рисование на canvas
                                } finally {
                                        surfaceHolder.unlockCanvasAndPost(canvas);
                                }
                        }
                }
        }
}
```

В классе DrawView нужно создать новое поле для дополнительного потока:

```
private DrawThread drawThread;
```

В переопределенный метод surfaceCreate() нужно добавить создание и вызов дополнительного пока, а также добавить его объявление в поле класса DrawView.

```
@Override
public void surfaceCreated(SurfaceHolder holder) {
        drawThread = new DrawThread(getContext(),getHolder());
        drawThread.start();
}
```

Перед удалением SurfaceView нужно позаботиться о завершении дополнительного потока, так как перерисовывать уже будет нечего. Сделать это нужно в методе SurfaceDestroy().

```
 @Override
public void surfaceDestroyed(SurfaceHolder holder) {
       drawThread.requestStop();
       boolean retry = true;
       while (retry) {
              try {
                     drawThread.join();
                     retry = false;
              } catch (InterruptedException e) {
                     //
              }
       }
}
```

В классе MainActivity в качестве параметра методу setContentView() нужно передать новый экземпляр класса DrawView.

```
 public class MainActivity extends AppCompatActivity {
        @Override
        protected void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                setContentView(new DrawView(this));
        }
}
```

Затем в класс DrawThread добавляем методы рисования на Canvas.

Пример 3.8

Из полученной заготовки создадим приложение, в котором смайлик будет передвигаться по экрану по направлению к месту касания.

Итак, у нас уже реализован класс MainActivity, он остается без изменений.

Поместим графический файл со смайликом *smile.png* (см. рис. 3.35) в папку с ресурсами (в нашем проекте Android Studio — папка *drawable*):

Рис. 3.35.

В уже реализованном ранее классе DrawThread добавляем поля для хранения картинки bitmap, координат касания (towardPointX, towardPointY), устанавливаем цвет фона, определяем конструктор:

```
 public class DrawThread extends Thread {

        private SurfaceHolder surfaceHolder;

        private volatile boolean running = true;
        private Paint backgroundPaint = new Paint();
        private Bitmap bitmap;
        private int towardPointX;
        private int towardPointY;
        {
                backgroundPaint.setColor(Color.BLUE);
                backgroundPaint.setStyle(Paint.Style.FILL);
        }

        public DrawThread(Context context, SurfaceHolder surfaceHolder) {
                bitmap = BitmapFactory.decodeResource(context.getResources(), R.drawable.sm
ile);
                this.surfaceHolder = surfaceHolder;
        }
...
```

Сеттер для координат:

```
 public void setTowardPoint(int x, int y) {
        towardPointX = x;
        towardPointY = y;
}
```

И переопределим метод run():

```
 @Override
public void run() {
        int smileX = 0;
        int smileY = 0;
        while (running) {
                Canvas canvas = surfaceHolder.lockCanvas();
                if (canvas != null) {
                        try {
                                canvas.drawRect(0, 0, canvas.getWidth(), canvas.getHeight
(), backgroundPaint);
                                canvas.drawBitmap(bitmap, smileX, smileY, backgroundPaint);

                                if (smileX + bitmap.getWidth() / 2 < towardPointX) smileX+=
10;
                                if (smileX + bitmap.getWidth() / 2 > towardPointX) smileX-=
10;
                                if (smileY + bitmap.getHeight() / 2 < towardPointY) smileY+
=10;
                                if (smileY + bitmap.getHeight() / 2 > towardPointY) smileY-
=10;
                        } finally {
                                surfaceHolder.unlockCanvasAndPost(canvas);
                        }
                }
        }
}
```

Здесь переменные smileX и smileY задают изменение позиции смайлика при перерисовке поверхности в зависимости от координат касания.

Наконец, в классе DrawView переопределим унаследованный метод onTouchEvent(), который будет передавать координаты касания экрана классу drawThread:

```
 @Override
public boolean onTouchEvent(MotionEvent event) {
    drawThread.setTowardPoint((int)event.getX(),(int)event.getY());
    return false;
}
```

Запускаем проект и получим требуемое приложение (полностью проект приведен здесь).

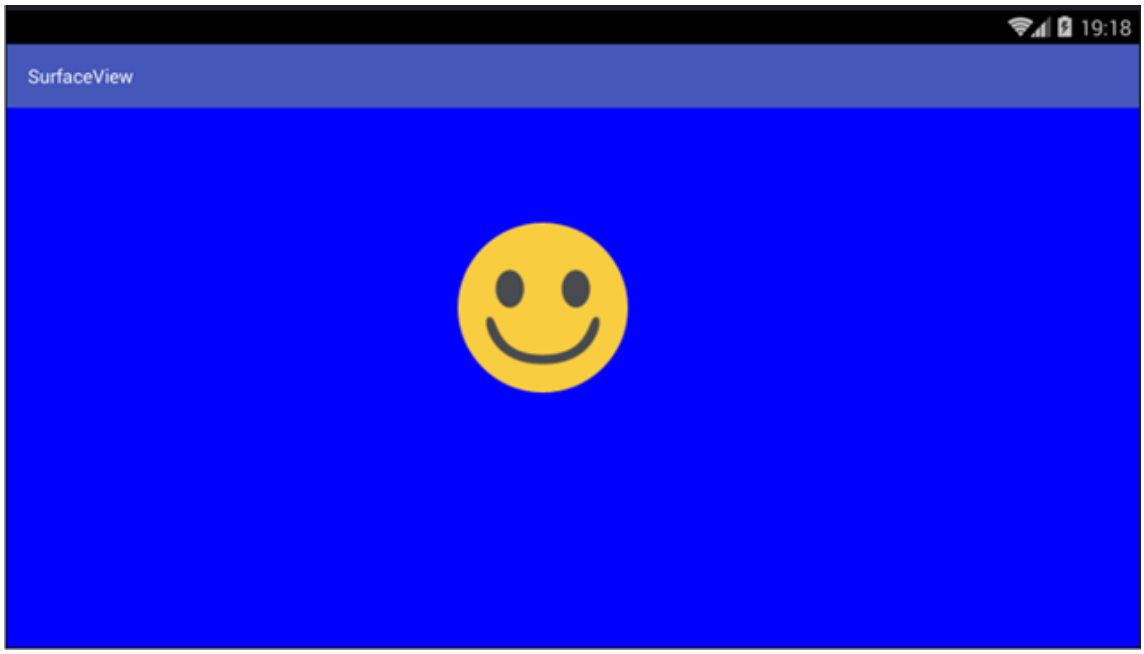


Рис. 3.36.

3.9.* Разработка 3D-игр с использованием фреймворка libGDX

Сайт: IT Академия SAMSUNG
Курс: MDev @ IT Академия Samsung
Книга: 3.9.* Разработка 3D-игр с использованием фреймворка libGDX
Напечатано:: Егор Беляев
Дата: Суббота, 18 Апрель 2020, 19:25

Оглавление

Введение в libGDX

Изучив основы создания 2D-графики с использованием простейших приемов работы на канвасе, становится понятно, что написать более-менее сложную 2D/3D-игру таким способом будет затруднительно. Основная проблема в том, что для создания игры программисту потребуется описать в программном коде множество типовых для игр алгоритмов. А именно:

- отрисовка игровых объектов в зависимости от расположения зрителя (камеры);
- изменение внешнего вида объектов в зависимости от освещения, которое определяется геймплеем;
- распознавание столкновения/пересечения игровых объектов;
- математические и физические расчеты;
- реализация взаимодействия с пользователем;
- передвижение игровых объектов;
- использование существующих 3D-моделей/2D-спрайтов.

При создании игры нужно будет написать функции для этих и многих других промежуточных задач, прежде чем приступать к реализации непосредственно самого геймплея. И это займет очень много времени. Однако можно взять готовую библиотеку, в которой уже имеются все необходимые функции для написания 3D-игр. Таких библиотек, а точнее фреймворков, для Android сегодня существует несколько. Однако предназначенный для разработки на Java всего один. Это libGDX.

LibGDX — это бесплатная платформа с открытым исходным кодом. Цель проекта — помочь в создании игр/приложений и развертывании на настольных и мобильных платформах.

LibGDX является свободным продуктом, лицензируется с лицензией на ПО с открытым исходным кодом Apache 2.

Пример 3.9

Рассмотрим основы использования libGDX под Android. В качестве примера разработаем 3D-игру под Android подобного вида (рис. 3.37).

Рис. 3.37.

Идея игры очень проста: игрок — второй пилот истребителя, отвечающий за системы вооружения. Игроку нужно успеть вовремя выстрелить лазерным оружием, когда двигатель вражеского корабля находится в перекрестье прицела, пока первый пилот старается не дать сбросить себя с хвоста. То есть, по сути, геймплей описывается фразой «кликни вовремя».

В ходе этого урока понадобится только Android Studio 1.5 или выше и мастер создания проекта от libGDX, который значительно облегчает задачу первоначальной настройки проекта (скачать можно по ссылке в инструкции на wiki проекта libGDX).

Сначала необходимо запустить мастер создания проекта командой:

```
java -jar gdx-setup.jar
```

В открывшемся окне мастера создания проекта необходимо указать все необходимые настройки. После нажатия кнопки Generate будет создан проект приложения в папке, указанной в поле Destination.

Рис. 3.38.

В созданном проекте уже сделаны все необходимые настройки для использования фреймворка libGDX и его можно просто открывать/импортировать в среде разработки — Android Studio, Idea или Eclipse.

Открыв получившийся проект в Android Studio, можно начинать собственно работу с кодом. Основной код игры находится в файле *SimpleGame.java* (это задается в мастере создания проекта). Удалим шаблонный код и начнем писать свой код:

```
public class SimpleGame extends ApplicationAdapter {
    public PerspectiveCamera cam;
    final float[] startPos = {150f, -9f, 0f};

    @Override
    public void create() {
        cam = new PerspectiveCamera(67, Gdx.graphics.getWidth(),
                                            Gdx.graphics.getHeight());
        cam.position.set(startPos[0], startPos[1], startPos[2]);
        cam.lookAt(0, 0, 0);
        cam.near = 1f;
        cam.far = 300f;
        cam.update();
    }
}
```

В этом небольшом кусочке кода создается новая камера с углом обзора в 67 градусов (это значение довольно часто используется). Тут же задается соотношение сторон ширины и высоты экрана. Затем позиция камеры устанавливается в точку (150, -9, 0) и разворачивается на центр координат (поскольку именно там будет размещена пирамидка, вокруг которой строится геймплей). В конце вызывается служебный метод update(), для того чтобы изменения применились к камере.

Теперь нужно отобразить на экране что-то, на что будет смотреть камера (игрок). Конечно, можно было бы использовать какую-то 3D-модель, но пока, в целях упрощения примера, создадим только простую пирамиду:

```
public class SimpleGame extends ApplicationAdapter {
    //...
    public Model model;
    public ModelInstance instance;

    @Override
    public void create() {
        //...
        ModelBuilder modelBuilder = new ModelBuilder();
        model = modelBuilder.createCone(20f, 120f, 20f, 3,
             new Material(ColorAttribute.createDiffuse(Color.GREEN)),
             VertexAttributes.Usage.Position | VertexAttributes.Usage.Normal);
        instance = new ModelInstance(model);
        instance.transform.setToRotation(Vector3.Z, 120);
    }

    @Override
    public void dispose() {
        model.dispose();
    }
}
```

Здесь сначала создается экземпляр ModelBuilder, который предназначен для создания моделей в коде. Затем с его помощью создается простая модель конуса с размерами 20×120×20 и количеством граней 3 (что в итоге и дает пирамиду) и материалом зеленого цвета. При создании модели необходимо задать как минимум Usage. Position. Usage. Normal добавляет к модели нормали, так что освещение сможет работать правильно.

Теперь модель содержит все необходимое для отрисовки и управления собственными ресурсами. Однако она не содержит информации о позиционировании. Для задания этих параметров нужно создать ModelInstance. Он содержит данные о расположении, параметрах вращения и масштаба для отрисовки модели. По умолчанию экземпляр модели будет помещен в точку (0, 0, 0), так что просто создадим ModelInstance. Кроме того, далее при помощи вызова метода transform.setToRotation() повернем пирамидку на 120 градусов по оси Z, для более удачного ракурса с позиции камеры.
Модель необходимо освобождать после использования, так что необходимо добавить вызов метода Dispose(), что и делается в методе onDispose() (см. жизненный цикл приложения libGDX).

После создания экземпляр модели ее необходимо визуализировать:

```
public class SimpleGame extends ApplicationAdapter {
    //...
    public ModelBatch modelBatch;

    @Override
    public void create() {
        modelBatch = new ModelBatch();
        //...
    }

    @Override
    public void render() {
        Gdx.gl.glViewport(0, 0, Gdx.graphics.getWidth(), Gdx.graphics.getHeight());
        Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT | GL20.GL_DEPTH_BUFFER_BIT);

        modelBatch.begin(cam);
        modelBatch.render(instance);
        modelBatch.end();
    }

    @Override
    public void dispose() {
        model.dispose();
        modelBatch.dispose();
    }
}
```

Добавляем в метод create() создание ModelBatch, который отвечает за отрисовку и инициализацию модели. В программе, разработанной для libGDX, метод render() используется для прорисовки объектов игры. Его необходимо переопределить и внести в него свой код визуализации. В приведенном методе render выполняется следующая последовательность действий:

- очищаем экран;
- запускаем камеру — modelBatch.begin(cam);
- рисуем ModelInstance;

- вызов modelBatch.end() завершает процесс отрисовки;
- наконец, нужно освободить modelBatch, чтобы гарантировать, что все ресурсы (например, шейдеры, которые он использует) корректно освобождены.

Рис. 3.39.

Модель выглядит неплохо (рис. 3.39), но наличие освещения сделало бы визуализацию лучше. Добавим его:

```
 public class SympleGame extends ApplicationAdapter {
    //...
    public Environment environment;

    @Override
    public void create() {
        environment = new Environment();
        environment.set(new ColorAttribute(ColorAttribute.AmbientLight, 0.4f, 0.4f, 0.4f, 1
f));
        environment.add(new DirectionalLight().set(0.8f, 0.8f, 0.8f, 10f, 10f, 20f));
        //...
    }

    @Override
    public void render() {
        //...
        modelBatch.begin(cam);
        modelBatch.render(instance, environment);
        modelBatch.end();
    }
}
```

В этом участке кода добавляем окружающую среду — класс Environment. Создаем его экземпляр и задаем окружающий (рассеянный) свет (0.4, 0.4, 0.4) (обратите внимание, что значение прозрачности игнорируется). Затем создаем источник направленного света — DirectionalLight с цветом (0.8, 0.8, 0.8) и направлением (10, 10, 20). И наконец, во время отрисовки передадим созданный environment (окружение) в обработчик моделей.

Рис. 3.40.

Теперь модель выглядит значительно лучше (рис. 3.40). Однако для игры важно иметь динамическую картинку. Добавим функционал, чтобы камера немножко сдвигалась при каждой прорисовке. Для реализации этого следует вспомнить жизненный цикле libGDX-приложения (рис. 3.41).

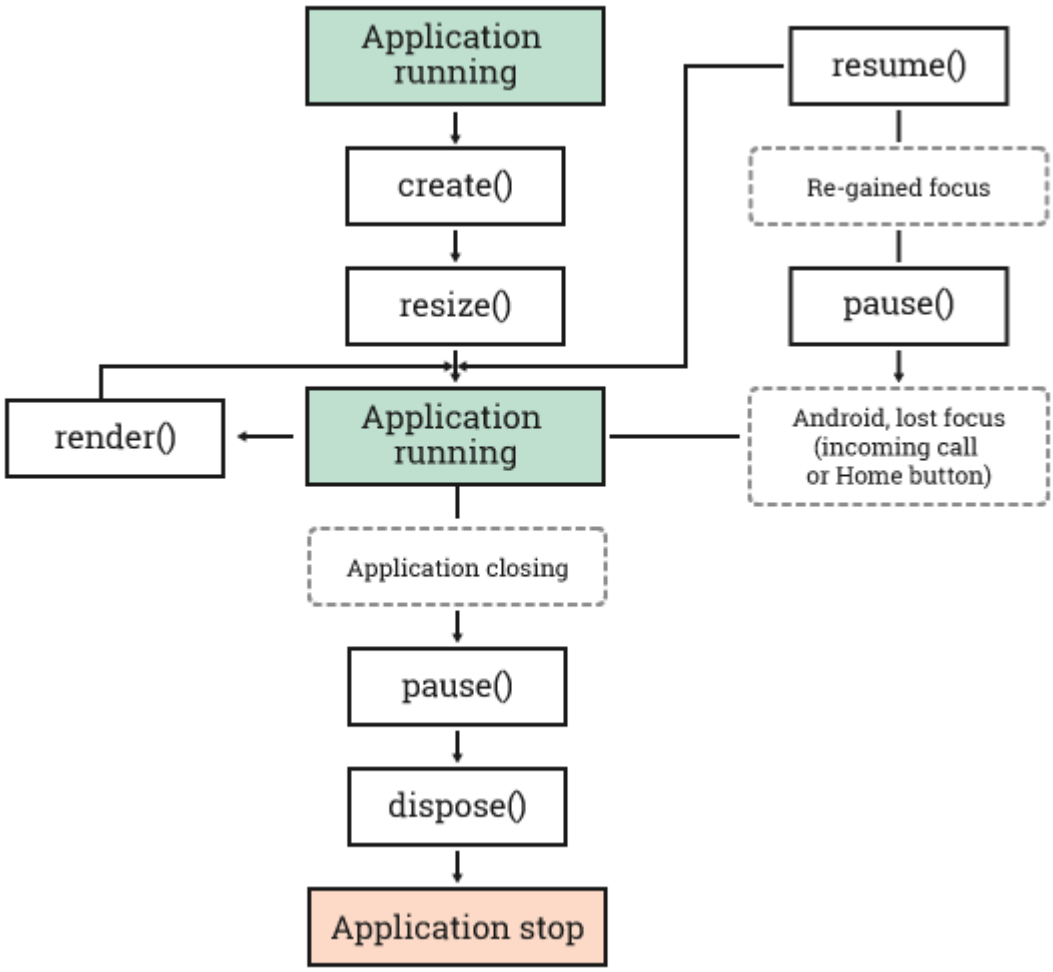


Рис. 3.41.

Для правильной визуализации игры в простейшем случае нужно использовать следующий алгоритм:

- при старте вызывается метод create(), в котором уместно размещать всю инициализацию;
- затем N раз в секунду вызывается метод render(), где N — FPS. Этот метод предназначен для рисования текущего кадра.

Таким образом, чтобы добавить в приложение динамичности, нужно в методе render() менять параметры положения игровых объектов.

```
 public class SimpleGame extends ApplicationAdapter {
    //...
    final float bound = 45f;
    float[] pos = {startPos[0], startPos[1], startPos[2]};
    float[] Vpos = new float[3];
    final float speed = 2f;

    private float getSpeed() {
        return speed * Math.signum((float) Math.random() - 0.5f) * Math.max((float) Math.ra
ndom(), 0.5f);
    }

    @Override
    public void create () {
        //...
        // initialize speed
        for (int i = 0; i < 3; i++){
            Vpos[i] = getSpeed();
        }
    }

    @Override
    public void render() {
        //...
        for (int i = 0; i < 3; i++) {
            pos[i] += Vpos[i];
            if (pos[i] <= startPos[i] - bound) {
                pos[i] = startPos[i] - bound;
                Vpos[i] = getSpeed();
            }
            if (pos[i] >= startPos[i] + bound) {
                pos[i] = startPos[i] + bound;
                Vpos[i] = getSpeed();
            }
        }
        cam.position.set(pos[0], pos[1], pos[2]);
        cam.update();

        modelBatch.begin(cam);
        modelBatch.render(instance, environment);
        modelBatch.end();
    }

}
```

В этой части программы создаем иллюзию того, что пирамидка двигается, хотя на самом деле движется камера, посредством которой игрок смотрим на нее. В начале игры в методе create() случайным образом выбирается значение приращения Vpos[i] для каждой координаты (скорость). На каждой перерисовке сцены в методе render() к координатам прибавляется значение шага изменения. Если происходит выход за установленные границы изменения

координат, то возвращаем координаты в эти границы и генерируем новые скорости, чтобы камера начала двигаться в другую сторону. cam.position.set() собственно и устанавливает камеру в новые координаты, рассчитанные по описанному выше закону, а cam.update() завершает процесс изменения параметров камеры.

На разных устройствах скорость движения пирамидки будет разной из-за разницы в FPS и, соответственно, количестве вызовов render() в секунду. Для предотвращения этого нужно использовать получаемое от libGDX значение времени, прошедшее с последнего вызова render(). Однако в рассматриваемом примере этого делать не будем, чтобы не усложнять проект.

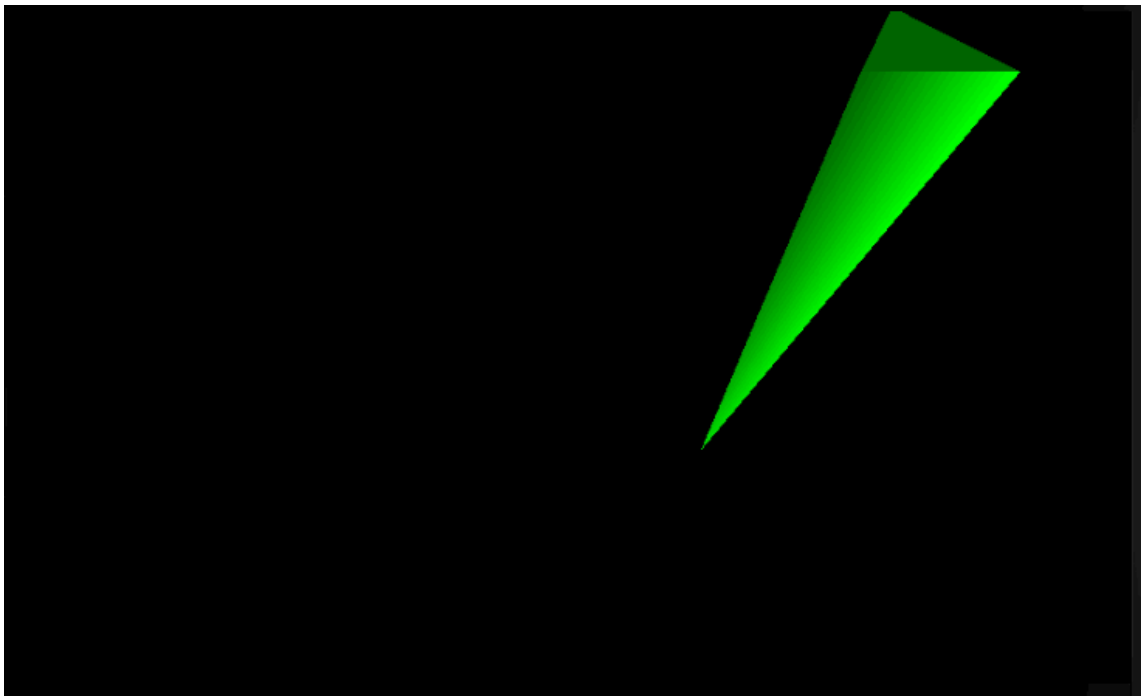


Рис. 3.42.

А теперь сделаем игровой HUD:

```
 public class SimpleGame extends ApplicationAdapter {
    //...
    protected Label label;
    protected Label crosshair;
    protected BitmapFont font;
    protected Stage stage;

    protected long startTime;
    protected long hits;

    @Override
    public void create() {
        //...
        instance.transform.setToRotation(Vector3.Z, 90).translate(-5,0,0);

        font = new BitmapFont();
        label = new Label(" ", new Label.LabelStyle(font, Color.WHITE));
        crosshair = new Label("+", new Label.LabelStyle(font, Color.RED));
        crosshair.setPosition(Gdx.graphics.getWidth() / 2 - 3, Gdx.graphics.getHeight() / 2
- 9);

        stage = new Stage();
        stage.addActor(label);
        stage.addActor(crosshair);

        startTime = System.currentTimeMillis();
    }

    @Override
    public void render() {
        //...
        StringBuilder builder = new StringBuilder();
        builder.append(" FPS: ").append(Gdx.graphics.getFramesPerSecond());
        long time = System.currentTimeMillis() - startTime;
        builder.append("| Game time: ").append(time);
        builder.append("| Hits: ").append(hits);
        builder.append("| Rating: ").append((float) hits/(float) time);
        label.setText(builder);
        stage.draw();
    }

    @Override
    public void resize(int width, int height) {
        stage.getViewport().update(width, height, true);
    }

}
```

Обратите внимание, что параметры вращения и сдвига (метод translate(x, y, z)) пирамидки изменены так, чтобы она находилась в центре экрана и была направлена туда же, куда смотрит камера. То есть на старте игры игрок находится с противником на одном курсе.

Здесь создаются 2 текстовых метки. Метка label предназначена для отображения внутриигровой информации (FPS, время игры и статистика попаданий). Метка crosshair рисуется красным цветом и содержит в себе только один символ — «+». Это показывает игроку середину экрана — его прицел. Для каждой из них в конструкторе new Label(<ТЕКСТ>, new Label. LabelStyle(font, <ЦВЕТ>)) задается стиль, включающий в себя шрифт и цвет надписи. Метки передаются в объект Stage методом addActor(), и, соответственно, отрисовываются автоматически, когда отрисовывается Stage.

Кроме того, для метки crosshair методом setPosition() задается позиция — середина экрана. Для этого получаем размеры экрана (Gdx.graphics.getWidth() иGdx.graphics.getHeight()), чтобы рассчитать, куда нужно поместить прицел, чтобы он оказался в середине. Обратите внимание, что setPosition() задает координаты левого нижнего угла надписи. Чтобы в центре экрана оказался именно центр плюсика, нужно вычесть константы 3 и 9. Однако не стоит использовать такой подход в полноценных играх. Для этого лучше использовать спрайты.

При каждой отрисовке создаем текст через StringBuilder, куда собираем все то, что хотим вывести внизу экрана: FPS, время в игре, количество попаданий и рейтинг. Используем метод setText(), чтобы выводить нужный текст в метку на каждом кадре в методе render().

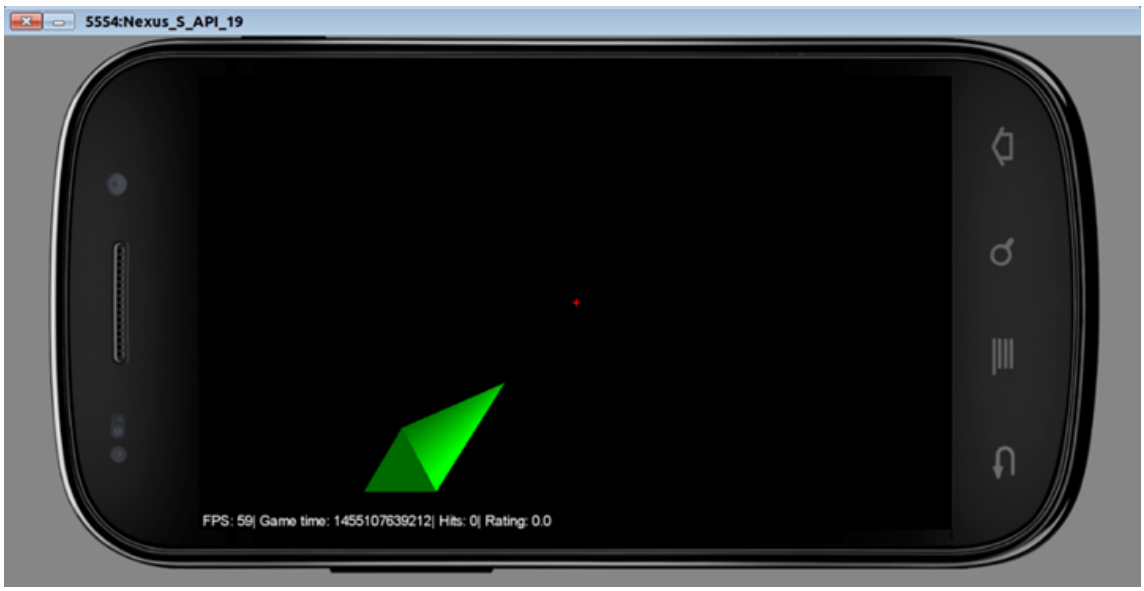


Рис. 3.43.

На данный момент полностью сделана визуализация геймплея (рис. 3.43), но нет самой важной его части. Игрок пока не может стрелять. Добавим нужную функциональность.

```
public class SimpleGame extends InputAdapter implements ApplicationListener {
    //...
    final float zone = 12f;
    boolean isUnder = false;
    long underFire;

    @Override
    public void create() {
        //...
        Gdx.input.setInputProcessor(new InputMultiplexer(this));
    }

    @Override
    public void render() {
        if (Math.abs(pos[1] - startPos[1]) < zone &&
                  Math.abs(pos[2] - startPos[2]) < zone) {
            isUnder = true;
            crosshair.setColor(Color.RED);
        } else {
            isUnder = false;
            crosshair.setColor(Color.LIME);
            underFire = 0;
        }
        //...
    }

    @Override
    public void pause() {}

    @Override
    public void resume() {}

    @Override
    public boolean touchDown(int screenX, int screenY, int pointer, int button) {
        if (isUnder) {
            underFire = System.currentTimeMillis();
        } else {
            hits /= 2;
        }
        return true;
    }

    @Override
    public boolean touchUp(int screenX, int screenY, int pointer, int button) {
        if (isUnder && underFire != 0) {
            hits += System.currentTimeMillis() - underFire;
            underFire = 0;
        } else {
            hits /= 2;
        }
        return false;
    }
}
```

Обратите внимание, что описание класса SimpleGame изменилось. Теперь он наследуется от InputAdapter и реализует интерфейс ApplicationListener. Такая структура позволит сохранить наш код в неизменном виде, и при этом дополнить его возможностью обработки пользовательского ввода. В методе create() добавлена строка, регистрирующая разрабатываемый класс как обработчик ввода. Методы pause() и resume() необходимо реализовать, поскольку у InputAdapter они абстрактные.

Вся математика расчета попадания находится в render(). В нем проверяется, находятся ли координаты камеры в той зоне, чтобы противник был в центре экрана (находятся ли координаты Y и Z в пределах start ± zone). Если игрок и цель на одном курсе, значит стрелять можно: устанавливаем isUnder = true и делаем прицел более яркого красного цвета. Опять-таки эта простота определения попадания — это некоторая хитрость, основанная на условности игрового процесса. Вообще в libGDX есть средства для определения, какие 3D-модели попали в область касания в общем случае.

Методы обработки касаний называются touchDown (палец коснулся экрана) и touchUp (палец убрали с экрана). Эти методы принимают координаты касания, но для рассматриваемого геймплея. На самом деле достаточно определить, находится ли камера сейчас в той позиции, чтобы смотреть на пирамидку прямо. Если это так (пользователь нажал вовремя), то в touchDown начинаем подсчет времени, сколько лазер поражал враждебную пирамидку. Если нет, то сокращаем очки пользователя делением надвое (штраф за промах). Когда пользователь отпускает палец, проверяем, не отпустил ли он его слишком поздно. Если отпустил поздно, то штрафуем, если вовремя (лазер еще поражал цель), то добавляем очки.

Пример 3.10

Модели

Несмотря на то что геймплей реализован, для игры не хватает красивой картинки. Пирамидка — это довольно скучно. Так что в качестве опционального дополнения к примеру можно еще реализовать 3D-модель летательного аппарата вместо пирамидки. Возьмем эту модель и добавим ее в игру.

Модель поставляется в четырех форматах разных 3D-редакторов. Однако libGDX использует свой бинарный формат моделей, в который их нужно конвертировать, чтобы использовать в игре. Для этого предусмотрена специальная утилита — fbx-conv. Скачиваем архив с исполняемыми файлами и распаковываем в какую-нибудь папку. Там есть версия под Windows, Linux и MacOS. Windows-версия запустится без дополнительных настроек, а для Linux и MacOS нужно предварительно выполнить команду:

```
export LD_LIBRARY_PATH=/folder/where/fbx-conv/extracted/
```

Этой командой указываем утилите, где искать свою разделяемую библиотеку libfbxsdk.so, которая требуется утилите для работы. Теперь запускаем утилиту:

```
./fbx-conv-lin64 -f space_frigate_6/space_frigate_6.3DS
```

Конечно, выше приведен только образец команды. При конвертации нужно будет указать путь к файлу модели и использовать исполняемый файл утилиты для текущей ОС. В итоге, в указанном примере получится файл space_frigate_6.g3db, который надо положить в папку проекта *android/assets* (папка с ресурсами приложения для платформы Android).

Вообще связка libGDX+fbx-conv очень проблемная. Было перепробовано около десятка бесплатных моделей космических кораблей с http://tf3dm.com/ и http://www.turbosquid.com/ прежде чем получилось найти эту, которая заработала. Сложности самые разные. Иногда модель в игре получается без текстур, иногда она загружается нормально, но просто не отображается, а иногда (это чаще всего) при загрузке модели игра выпадает с OutOfMemoryError. Даже та модель, которая в конечном счете была использована, доставила проблем. Из obj нормально не конвертировалась, а вот из 3ds получилось. В свете этого можно сказать, что пока еще у libGDX с поддержкой моделей все обстоит не очень хорошо. Можно использовать этот движок для простеньких игр, если старательно подбирать модели или же делать их самостоятельно с оглядкой на совместимость с libGDX. Или же использовать более продвинутые движки типа jMonkeyEngine.

Теперь подключим загруженную модель в игру:

```
public class SimpleGame extends InputAdapter implements ApplicationListener {
   //...
   public AssetManager assets;
   public boolean loading;

   @Override
   public void create() {
       //...
       assets = new AssetManager();
       assets.load("space_frigate_6.g3db", Model.class);
       loading = true;
   }

   @Override
   public void render() {
       if (loading)
           if (assets.update()) {
               model = assets.get("space_frigate_6.g3db", Model.class);
               instance = new ModelInstance(model);
               loading = false;
           } else {
               return;
           }
       //...
   }

   @Override
   public void dispose() {
       model.dispose();
       modelBatch.dispose();
   }

}
```

Здесь создаем экземпляр класса AssetManager, который отвечает за загрузку игровых ресурсов и указываем ему загрузить данную модель. На каждой отрисовке проверяем, не загрузил ли еще AssetManager модель (метод update(), возвращающий boolean. Если загрузил, то присваиваем в instance вместо пирамидки модель самолета и устанавливаем loading = false, чтобы это создание inctance не повторялось на каждом кадре, ведь assets.update() будет возвращать true и далее на протяжении всего времени работы приложения.

При запуске получим исключение java.io. FileNotFoundException: SPACE_FR.PNG. Это означает, что файл модели не включает текстуры, их нужно добавлять отдельно. Выбрав из четырех представленных понравившуюся текстуру, переименовываем ее в SPACE_FR.PNG, копируем в assets и запускаем приложение. В итоге получим готовую игру (рис. 3.44).

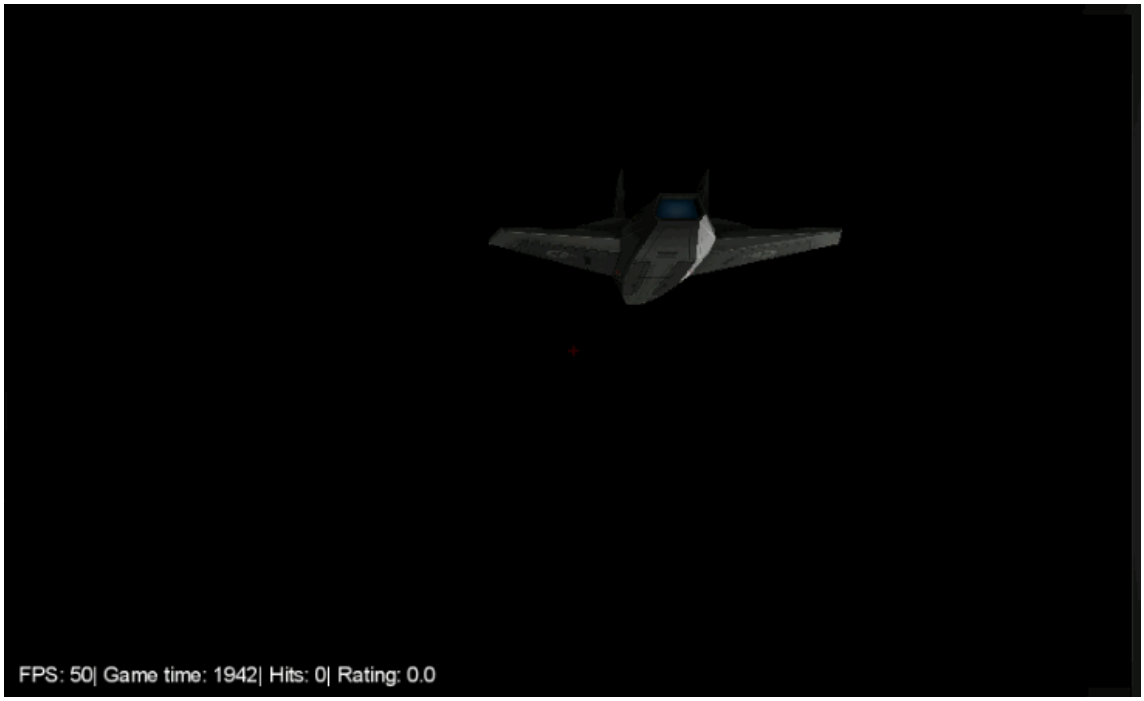


Рис. 3.44.

Итог

В примере была создана очень простая, но почти полноценная, с точки зрения использованных средств (освещение, движение, HUD, касания, модели), игра. При этом размер получившейся программы всего порядка 200 строк. Конечно, в ней есть многое, что можно улучшить: нормальный прицел, skybox (небо или космос вокруг), звуки выстрелов и полета, игровое меню, более правильное определение попадания и т. п. Тем не менее, игра уже содержит базу игрового процесса и наглядно показывает самые основные моменты разработки игр на libGDX.

Проект и установочный APK файл можно скачать по адресу — https://github.com/myitschool-book/3.9_exersises.git

Источники

- https://libgdx.badlogicgames.com/nightlies/docs/api/overview-summary.html>
- http://www.todroid.com/android-gdx-game-creation-part-i-setting-up-up-android-studio-for-creating-games/
- https://xoppa.github.io/blog/basic-3d-using-libgdx/
- https://habrahabr.ru/post/276139/