

## **4.1. Массивы. Класс ArrayList**

Сайт: IT Академия SAMSUNG

Курс: MDev @ IT Академия Samsung

Книга: 4.1. Массивы. Класс ArrayList

Напечатано:: Егор Беляев

Дата: Суббота, 18 Апрель 2020, 19:28

# **Оглавление**

- 4.1.1. Структуры данных. Сложность алгоритмов
- 4.1.2. Массив — базовая структура данных
- 4.1.3. Операции с массивами. Класс Arrays
- 4.1.4. Сравнение объектов. Компараторы
- 4.1.5. Расширяемый массив. Класс ArrayList
- 4.1.6. Алгоритмы поиска элементов по значению

## 4.1.1. Структуры данных. Сложность алгоритмов

В этом модуле рассматриваются структуры данных и алгоритмы.

Теоретически для программирования вполне достаточно одних массивов. Но использование специальных классов и объектов делает программирование удобнее, позволяет не задумываться о внутреннем устройстве объектов благодаря инкапсуляции. Хранение и обработка больших объемов данных в специализированных классах становится более простыми и удобными.

Массивы обладают одним существенным недостатком. После создания в них сложно или невозможно производить изменения. Изменить количество ячеек массива в принципе невозможно. Если мы хотим добавить элементы в массив, то мы можем создать новый массив и переместить все старые элементы в начало нового. Все это можно «поручить» объекту специального класса, который будет содержать массив у себя внутри. Так устроен класс `ArrayList`.

Другая проблема использования массивов напрямую — скорость работы некоторых операций. Изменить значение элемента массива легко, а, например, вставить элемент в начала массива сложно, в смысле долго: для этого нужно переместить все элементы на одну ячейку вправо. Оказывается, можно хранить данные определенным образом так, чтобы вставка элемента в начало последовательности проходило практически так же быстро, как и изменение значения элемента. Мы обсудим внутреннее строение такой структуры данных, а в стандартной библиотеке языка Java такая структура реализована классом `ListedList`. Другие способы организации данных, тоже реализованные в Java в виде классов, позволяют выполнять быстро другие операции, например, поиска элемента.

Вопрос скорости обработки достаточно важный. В этом модуле мы будем постоянно оценивать алгоритмы обработки данных по времени работы. Обычно она измеряется в количестве операций. При этом не вычисляют конкретное количество операций, а говорят о порядке этой величины.

Чаще всего нам будут встречаться алгоритмы, выполняющиеся за:

- **фиксированное количество действий, не зависящее от количества элементов** в контейнере. Например, чтобы поменять первый элемент массива с последним делают три действия, в этом случае говорят — сложность алгоритма  $O(1)$ ;
- **время, пропорциональное размеру контейнера**. Например, нахождение максимума и минимума в массиве (для этого нужно сделать порядка  $2N$  сравнений, где  $N$  — это количество элементов в нем). Сложность таких алгоритмов —  $O(n)$ ;
- **время, пропорциональное размеру контейнера в квадрате**. Например, сортировка массива пузырьком делает  $\frac{N \cdot (N + 1)}{2}$  сравнений. Сложность такого алгоритма сортировки —  $O(n^2)$ ;
- **логарифму от размера контейнера**, например, для того чтобы найти требуемое число в отсортированном массиве, можно действовать так: проверять средний элемент, если искомый меньше, то искать в левой половине, деля ее уже на четверти, если нет — в правой. Тогда для 16 элементов нужно четыре проверки. А для миллиарда — всего тридцать! Сложность подобных алгоритмов записывают  $O(\log(n))$ .

Таким образом, в стандартной библиотеке Java структуры данных делятся на **массивы и коллекции** (кроме ассоциативных массивов, которые реализуют интерфейс `Map`).

Зачем нужны массивы, если коллекции во многом удобнее и решают некоторые задачи по работе с данными значительно быстрее? Причина этого та же, по которой в Java существуют примитивные типы и классы. Использование объектов значительно удобнее,

но примитивные типы — то, чем оперирует программа на уровне машинных операций. При выполнении программы все данные представлены в памяти именно в виде массивов, непрерывной последовательности ячеек. Поэтому работа с массивами в определенных случаях проходит гораздо эффективнее, чем использование специальных классов. Поэтому во многих случаях методы принимают в качестве параметров именно массивы. Более того, практически все коллекции содержат в себе именно массивы как основу для хранения данных. Поэтому нужно обязательно уметь конвертировать коллекции в массивы и обратно.

Удивительно, но благодаря объектно-ориентированному программированию можно даже отделить данные от алгоритмов, их обрабатывающих. Действительно, для того чтобы отсортировать любую последовательность, выстроить ее элементы в порядке по неубыванию, нужно уметь две вещи: получать элементы этой последовательности и уметь сравнивать их по величине (для чисел это вполне естественный порядок, а для сортировки точек на плоскости нужно «ввести отношение порядка», записать алгоритм сравнения двух точек, который будет выдавать вердикт, какая из двух точек «больше» другой). Если так, то достаточно всего одной функции `sort()`, которая будет принимать любой контейнер (коллекцию или массив) и объект-компаратор («сравнитель»). В стандартной библиотеке эта функция сделана статической в классах `Arrays` и `Collections`. Эти два класса как раз и содержат функции для обработки массивов и коллекций.

## 4.1.2. Массив — базовая структура данных

Массивы изучались в первом модуле, в разделе 1.8. В этом разделе мы возвращаемся к этой теме, потому что массив — это базовая структура данных. Многие структуры данных реализованы в Java на базе массивов.

Самое главное, что нужно помнить по работе с массивами объектов, — при создании массива объектов сами объекты не создаются. Например, код

```
String[] array = new String[10];
System.out.println(Arrays.toString(array));
```

Выведет

```
[null, null, null, null, null, null, null, null, null, null]
```

То есть строки не были созданы даже пустыми и для работы нужно их создать или присвоить им значения других объектов, например, так:

```
for (int i = 0; i < array.length; i++)
    array[i] = "";
```

### 4.1.3. Операции с массивами. Класс Arrays

В Java массивы — это объекты. Поэтому их нельзя сравнивать при помощи `==` и при помощи `equals()` и копировать сами объекты присваиванием, если мы хотим работать с их элементами.

Для копирования массивов можно использовать метод `clone()` или использовать метод `arraycopy()` класса `System`. Например:

```
int[] a = {1, 2, 3, 4, 5};  
int[] b = a.clone();  
int[] c = new int[a.length];  
System.arraycopy(a, 0, c, 0, c.length);
```

- Массив-источник;
- индекс начала копирования из источника;
- массив-приемник;
- индекс начала копирования в приемнике;
- количество элементов, которые будут скопированы.

Например, код:

```
int[] a = {1, 2, 3, 4, 5};  
int[] b = {6, 7, 8, 9};  
System.arraycopy(a, 2, b, 1, 3);
```

Скопирует три элемента из `a`, начиная с третьего по счету элемента, в `b`, начиная со второго по счету. Таким образом, массив `b` станет таким:

```
[6, 3, 4, 5]
```

Обратите внимание, массив-приемник должен быть создан заранее и иметь подходящую длину, чтобы вместить все элементы при копировании.

Метод `arraycopy()` реализован на низком уровне, поэтому работает гораздо быстрее копирования при помощи цикла. Особенно это заметно при большом объеме копируемых данных (см. рис. 4.1).

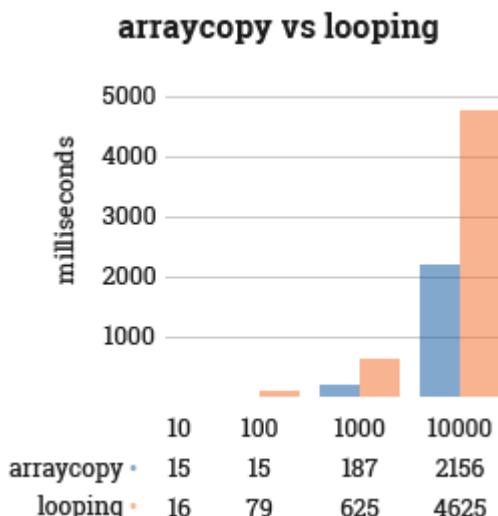


Рис. 4.1.

## Класс Arrays

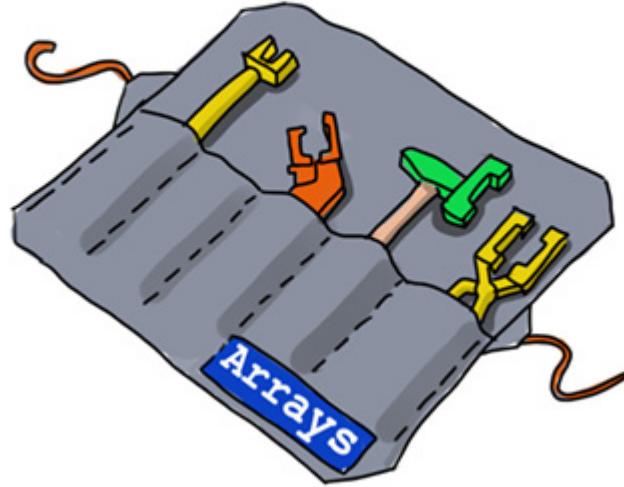
Для стандартных операций с массивами (поиск, сортировка и пр.) удобно использовать класс `ArrayList`.

Методы класса `Arrays` можно разделить на группы:

- заполнение;
- вывод (сериализация);
- сортировка;
- поиск;
- сравнение.

Все эти методы являются статическими, а, значит, должны быть вызваны по имени класса `Arrays`.

Массив, к которому применяют эти методы, является одним из параметров указанных методов и должен быть описан как `type_[]`, где `type_` может быть одним из примитивных типов `byte`, `short`, `int`, `long`, `char`, `float`, `double`, или тип `Object`, что, в свою очередь, означает возможность использования любого типа, являющегося наследником типа `Object`.



Таким образом, на каждую операцию реализовано по восемь функций. Мы будем использовать символы `type_[]`, обозначая всю группу.

## Заполнение массива

Первая группа методов — `fill()` — заполняет указанным значением `value`:

Заполнение массива целиком:

```
static void fill(type_[] a, type_ value)
```

Это метод удобно использовать, когда необходимо задать значение по умолчанию для всех элементов массива:

```
import java.util.Arrays;  
//...  
int[] a = new int[10];  
Arrays.fill(a, -1); //заполнение всех элементов массива значением -1
```

Заполнение части массива от индекса `from` включительно до индекса `to` исключительно (полуинтервала `[from; to)`):

```
static void fill(type_[] a, int from, int to, type_ value)
```

Например:

```
// заполнение единицами первой половины массива  
Arrays.fill(a, 0, a.length / 2, 1);
```

## Вывод массива

В классе Arrays имеется группа методов `toString()`, представляющих массив в виде строки. Этот процесс называют «*сериализацией*».

```
String toString(type_[] a)
```

При помощи этих методов, например, удобно выводить массивы на консоль. Например, код:

```
int[] a = new int[5];
Arrays.fill(a, 1);
```

Выведет

```
[1, 1, 1, 1, 1]
```

## Сортировка массива

Следующая группа методов — `sort()` — сортирует массив в порядке возрастания числовых значений:

```
Arrays.sort(b);
```

(О сортировке объектов см. следующий параграф.)

Аналогично методам заполнения существуют методы, которые принимают границы сортируемого диапазона.

## Поиск в массиве

В классе Arrays отсутствует метод поиска в произвольном массиве. Для поиска элементов в произвольном массиве Группа методов `binarySearch()` организует бинарный поиск, при котором метод возвращает индекс найденного элемента массива.

Отметим, что бинарный поиск работает корректно только на отсортированном массиве.

Подробно бинарный поиск рассматривается далее.

В классе Arrays отсутствует метод поиска в произвольном массиве. Для этого можно использовать простой перебор элементов при помощи цикла.

## Сравнение массивов

Еще одна группа методов — `equals()` — сравнивает массивы:

```
static boolean equals (type_[] a1, type_[] a2)
```

Массивы считаются равными, если они имеют одинаковую длину и равные элементы с одинаковыми индексами. В этом случае метод возвращает `true`.

Код

```
int[] a = {1, 2, 3, 4, 5};
int[] b = a.clone();
System.out.println( (a == b) + " +" + (Arrays.equals(a, b)));
```

выведет

```
false true
```

Потому что массивы `a` и `b` — это разные объекты, но они оба массивы целых чисел и поэлементно равны.

Если элементы класса не относятся ни к одному из примитивных типов, то в классе необходимо реализовать метод `equals()`, который будет сравнивать отдельные элементы массивов на равенство.

## 4.1.4. Сравнение объектов. Компараторы

Для того чтобы отсортировать контейнер, нужно уметь сравнивать элементы между собой. Для примитивных числовых типов все естественно: числа можно сравнивать «как в математике», а у объектов нужно указать способ, ввести порядок, упорядочить их. Например, как сортировать список сотрудников? Возможно, по фамилиям, а, возможно, по зарплате. При этом какого сотрудника поставить в начало списка с наибольшей или, наоборот, с наименьшей зарплатой?

Для упорядочивания объектов в языке Java есть два способа: «научить» сами объекты класса сравниваться между собой (реализовать в классе интерфейс Comparable и создать еще один класс, реализующий интерфейс Comparator — «сравнитель»), такие классы называют **компараторами**.

Рассмотрим подробно второй вариант.

Интерфейс Comparator обязывает определить метод сравнения:

```
compare(Object obj1, Object obj2)
```

Метод compare(Object obj1, object obj2) должен возвращать:

- отрицательное число, если obj1 считается меньше, чем obj2;
- 0, если они считаются равными;
- положительное число, если obj1 считается больше, чем obj2.

У классов Arrays и Collections есть соответствующие методы sort(), которые принимают объект компаратора последним параметром.

Параметром метода сортировки с использованием компаратора обязательно должен быть наследник класса Object, а не примитивный тип.

Рассмотрим использование компаратора на примере.

### Пример 4.1

Отсортировать массив по возрастанию значений младшей цифры элементов массива при помощи функции sort() класса Arrays.

Для такой сортировки создадим компаратор. Будем использовать объекты класса Integer.

```
class LastDigitComp implements Comparator<Integer> {
    @Override
    public int compare(Integer obj1, Integer obj2) {
        // получаем последние цифры чисел
        int m1 = obj1 % 10;
        int m2 = obj2 % 10;
        // и сравниваем их
        if (m1 < m2)
            return -1;
        else if (m1 > m2)
            return 1;
        else
            return 0;
    }
}
```

Теперь создадим массив и заполним его случайными числами.

```
final int MAXRANGE = 100;
Random rnd = new Random();
Integer[] a = new Integer[10];
for (int i = 0; i < a.length; ++i) {
    a[i] = rnd.nextInt(MAXRANGE);
}
```

Отсортируем массив (и выведем его до и после сортировки).

```
System.out.println("Random: " + Arrays.toString(a));
Arrays.sort(a, new LastDigitComp());
System.out.println(Arrays.toString(a));
```

На экране появится:

```
Random: [56, 36, 92, 38, 76, 34, 52, 62, 86, 93]
Sorted:[92, 52, 62, 93, 34, 56, 36, 76, 86, 38]
```

Изменяя условия в компараторе, мы можем менять порядок сортировки массива (например, поменяв местами команды `return -1` и `return 1`, мы изменим сортировку по возрастанию на сортировку по убыванию значений младшей цифры элементов массива).

Важный вопрос: как сортируются элементы, если у них последняя цифра одинакова?  
На выводе мы видим 92, 52, 62. Почему числа выстроились именно в таком порядке? Дело в том, что с точки зрения компаратора они «равны», а алгоритм `sort()` реализует стабильную сортировку. Это значит, что равные элементы будут располагаться в контейнере в том же порядке, что и до сортировки.

## 4.1.5. Расширяемый массив. Класс ArrayList

Основной недостаток массивов в том, что обычный массив имеет фиксированную длину. После того, как он создан, изменение его свойства `length` невозможно (это свойство объявлено как `final`). Это неудобно, потому что часто мы заранее не знаем, какой объем данных получит программа.

Но фиксированный размер — это принципиальная необходимость. При резервировании памяти нам выделяется участок памяти. И расширить его потом не всегда возможно («продолжение» может быть уже занято под другие объекты). Единственное, что можно сделать в любом случае, это создать массив большего размера, скопировать туда элементы из старого массива, а потом старый удалить. В Java последнее действие будет выполнено автоматически при помощи сборщика мусора.

### Простой расширяемый массив

Разработаем «резиновый», автоматически расширяемый массив. Для этого создадим новый класс, в котором разместим массив.

```
class MyArrayList{  
    private Object[] data;  
    // длина используемой части  
    private int size;
```

Стоит делать его массивом `Object`, благодаря этому наш класс станет универсальным: в нем можно будет хранить объекты любых типов. И примитивные типы тоже, благодаря автоупаковке.

Обычно хранят массив большего размера, чем реально необходимо `size < length`. Это дает возможность не пересоздавать его при добавлении каждого элемента. И хорошо иметь возможность сразу инициализировать предполагаемым числом элементов. Поэтому конструкторы могут быть такие:

```
public MyArrayList(int n) {  
    data = new Object[n];  
}  
public MyArrayList() {  
    this(10);  
}
```

Для работы с элементами нужно написать функции `get()` и `set()`:

```
// обращение к элементам  
public Object get(int i) {  
    return data[i];  
}  
  
public void set(int i, Object obj) {  
    data[i] = obj;  
}
```



В языке Си можно перегружать операторы. Таким образом, работу с «резиновым» массивом можно сделать прозрачной — индексацию организовать обычными квадратными скобками. В Java для этого необходимо использовать методы.

Здесь мы не проверяем границы индекса *i*. Это не очень хорошо, например, возможна ситуация, что мы возьмем пятый элемент из массива длиной в три элемента. Причем, так как внутренний массив обычно «длиннее», сообщение об ошибке мы тоже не получим.

Наконец, самая важная функция: добавления элементов. Она должна просто положить элемент, если есть свободное место или пересоздать массив.

На сколько элементов увеличивать массив в случае нехватки места? Можно доказать, что эффективно увеличивать размер массива в какое-то количество раз. Часто увеличивают размер вдвое. Стандартный *ArrayList* увеличивает его на две трети.

```
public void add (Object obj)
{
    if (size == data.length){
        Object[] nData = new Object[size * 2];
        System.arraycopy(data, 0, nData, 0, size);
        data = nData;
    }
    data[size] = obj;
    size++;
}
```

Удаление элементов — длительный процесс. Если удаляются последние элементы, можно просто уменьшить *size*, если из середины, нужно перемещать элементы:

```
public void remove(int i) {
    if (i != size - 1)
        System.arraycopy(data, i + 1, data, i, size - i);
    // очищаем последний элемент
    data[size - 1] = null;
    size--;
}
```

Добавим еще функцию:

```
public int getSize() {
    return size;
}
```

Испытаем класс *MyArrayList*:

```
MyArrayList myArrayList = new MyArrayList();
myArrayList.add("one");
myArrayList.add("two");
myArrayList.add("three");
myArrayList.remove(1);
myArrayList.add(4);
for (int i = 0; i < myArrayList.getSize(); i++)
    System.out.print(myArrayList.get(i)+ " ");
```

Программа выводит

```
one three 4
```

То есть в нашем «резиновом» массиве хранятся и строки, и целые числа (в виде объектов Integer).

Отметим два недостатка построенного класса. Во-первых, не работает код:

```
String test = myArrayList.get(0); // НЕКОРРЕКТНО!
```

Правильно будет, если приводить тип в явном виде:

```
String test = (String)(myArrayList.get(0));
```

Ситуацию можно исправить, если сделать класс параметризованным при помощи дженериков, но тогда в нем нельзя будет хранить объекты разных типов.

Второй существенный недостаток: нельзя для обхода использовать цикл foreach.

```
for (Object obj : myArrayList) // НЕКОРРЕКТНО!
```

Это происходит потому, что нужно реализовывать интерфейс Iterable, научить класс обходить свои элементы.

## Итераторы

Интерфейс Iterator (итератор) содержит обобщенную схему доступа ко всем элементам контейнера (коллекции объектов), которая не зависит от особенностей его организации. Итератор последовательности возвращает элементы в соответствии с линейным порядком их следования.

Для реализации интерфейса Iterator необходимо определить методы:

- hasNext() — возвращает true, если следующий элемент существует в коллекции;
- next() — возвращает следующий элемент коллекции, при этом итератор переходит на следующий элемент;
- remove() — удаляет текущий элемент.

Напишем для класса MyArrayList свой итератор:

```

private class Itr implements Iterator {
    int cursor;

    @Override
    public boolean hasNext() {
        if (cursor < size)
            return true;
        else
            return false;
    }

    @Override
    public Object next() {
        cursor++;
        return data[cursor - 1];
    }

    @Override
    public void remove() {
        MyArrayList.this.remove(cursor);
    }
}

```

Реализуем в классе MyArrayList интерфейс Iterable. Это всего один метод, возвращающий итератор.

```

@Override
public Iterator iterator() {

    return new Itr();
}

```

И теперь можно обходить MyArrayList при помощи итератора:

```

Iterator iterator = myArrayList.iterator();
while(iterator.hasNext())
{
    Object obj = iterator.next();
    System.out.print(obj + " ");
}

```

или проще:

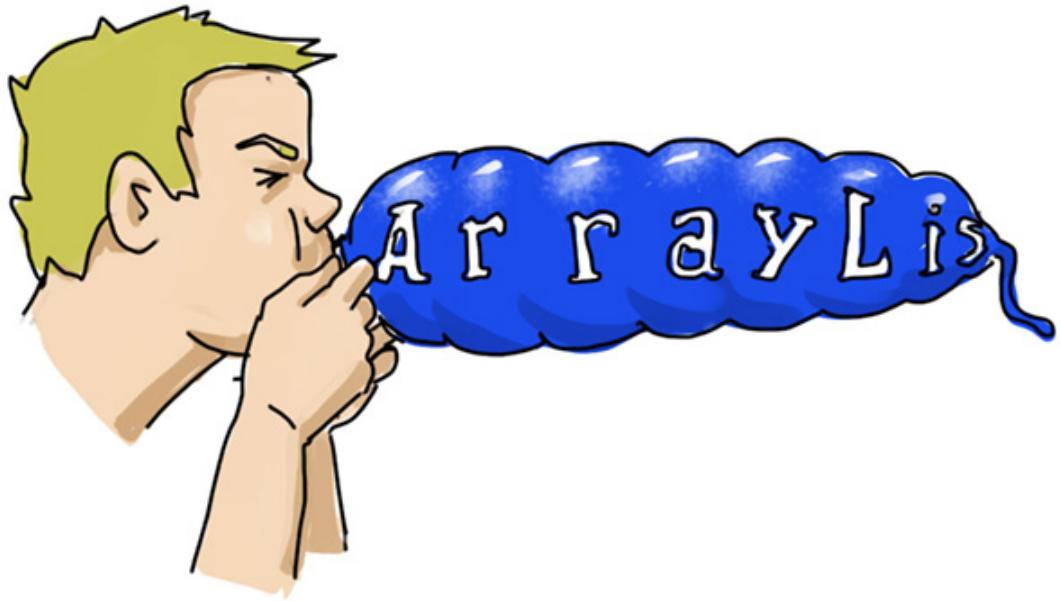
```

for (Object obj : myArrayList){
    System.out.print(obj + " ");
}

```

## Библиотечный класс ArrayList

В Java автоматически расширяемый массив представлен классом стандартной библиотеки ArrayList.



В отличие от обычных массивов класс `java.util.ArrayList` из пакета `java.util` является автоматически расширяемым массивом. При создании объекта типа `ArrayList` необязательно указывать его размерность. При работе с таким массивом используются специальные методы.



Подробно `ArrayList` обсуждается в статье <http://habrahabr.ru/post/128269/>.

#### Достоинства `ArrayList`

- Быстрый доступ к элементам по индексу за константное время  $O(1)$ ;
- доступ к элементам по значению за линейное время  $O(n)$ ;
- позволяет хранить любые значения в том числе и `null`.

#### Недостатки `ArrayList`

- Вставка/удаление элемента в середине списка вызывает перезапись всех элементов, размещенных «правее» в списке на одну позицию влево, то есть работает за линейное время  $O(n)$ ;
- при удалении элементов размер массива не уменьшается, до явного вызова метода `trimToSize()`
- не синхронизирован (использовать в многопоточной программе нужно с осторожностью);
- поиск элемента по значению работает за линейное время  $O(n)$ . (Как и в обычном массиве, для того чтобы найти элемент, нужно просмотреть все его элементы.)

### **Создание массива и добавление в него элементов**

Выполняется с помощью метода `add()`:

```
import java.util.ArrayList;  
//...  
  
ArrayList friendsList = new ArrayList();  
friendsList.add("Антонов Сергей");  
friendsList.add("Сергеев Антон");
```

При необходимости можно добавить новый элемент в указанное место (по индексу в диапазоне от 0 до size()-1 (см. ниже)):

```
friendsList.add(1, "Андреев Игорь");
```

При добавлении и удалении элементов ArrayList меняет свой размер, который можно узнать с помощью метода size() (а не поля length).

```
System.out.println(friendsList.size());
```

Обратиться к элементу можно по его индексу с помощью метода get(). Индексация в ArrayList, как и в статических массивах, начинается с 0.

```
System.out.println(friendsList.get(0));
```

Соответственно, обращение к последнему элементу массива будет выглядеть так:

```
System.out.println(friendsList.get(friendsList.size()-1));
```

В friendsList можно добавить и целое число (а не строку с именем друга), при этом компилятор java не выдаст никаких сообщений об ошибках.

```
friendsList.add(5);
```

Но в дальнейшем, например, при попытке распечатать содержимое списка, возникнут проблемы, так как объекты из ArrayList будут иметь разный тип.

Для корректного использования объектов типа ArrayList удобно его параметризовать (в угловых скобках указать тип элементов этого контейнера).

```
ArrayList <String> friendsList = new ArrayList <>();
```

## Замена и удаление элементов

Для замены элемента в массиве нужно использовать метод set() с указанием индекса и новым значением:

```
friendsList.set(1, "Викторов Андрей");
```

Для удаления элемента из массива пользуются методом remove(). При этом элемент можно удалять как по индексу, так и по объекту:

```
friendsList.remove("Антонов Сергей");
friendsList.remove(0);
```

Очистка списка (удаление всех элементов) — это метод clear().

```
friendsList.clear();
```

При поиске элемента в массиве применяют indexOf(), который возвращает номер элемента в диапазоне от 0 до size()-1, или отрицательное число, если элемент не найден.

```
int ind = friendsList.indexOf("Сергеев Антон");
System.out.println("Индекс элемента = " + ind);
```

## Просмотр и поиск элементов

Просмотр всего списка можно осуществить несколькими способами.

1. через цикл for

```
for (int i = 0; i < friendsList.size(); i++) {
    System.out.println(friendsList.get(i));
}
```

2. через цикл for each

```
for (String friend : friendsList) {
    System.out.println(friend);
}
```

3. через Iterator

```
import java.util.Iterator;
...
Iterator iterator = friendsList.iterator();
while (iterator.hasNext()) {
    System.out.println(iterator.next());
}
```

Чтобы узнать, есть в массиве какой-либо элемент, можно воспользоваться методом `contains()`, который вернет true или false:

```
if (friendsList.contains("John Johnson")) {
    System.out.println("Есть такой друг!");
}
```

В массиве `ArrayList` значения вполне могут совпадать (как и в обычном статическом массиве). Например, среди друзей попадаются тезки, и мы их можем добавить в `ArrayList`. Узнать, сколько раз повторяются одинаковые элементы, можно, если обратиться к методу `frequency()` класса `Collections`.

```
import java.util.Collections;
//...

ArrayList<String> friendsList = new ArrayList<String>();
friendsList.add("Иванов");
friendsList.add("Петров");
friendsList.add("Сидоров");
friendsList.add("Иванов");
int count = Collections.frequency(friendsList, "Иванов");
System.out.println(count + " ");
```

## Конвертация в массив

Также можно конвертировать список `ArrayList` в обычный статический массив. Конвертация в массив может понадобится, например, для ускорения некоторых операций или передачи массива в качестве параметра методам, которые требуют именно массив.

Это можно сделать при помощи метода `toArray()`. Этот метод принимает массив, который заполняется элементами `ArrayList`.

Перед вызовом метода массив должен быть создан и количество элементов в нем должно быть не меньше, чем в конвертируемом `ArrayList`.

Например:

```
ArrayList<String> arrayList = new ArrayList<String>();
arrayList.add("Антонов Сергей");
arrayList.add("Сергеев Антон");
arrayList.add("Викторов Андрей");
String[] array = new String[arrayList.size()];
// конвертируем ArrayList в массив
arrayList.toArray(array);
System.out.println(Arrays.toString(array));
```

Программа выведет полученный массив:

```
[Антонов Сергей, Сергеев Антон, Викторов Андрей]
```

## 4.1.6. Алгоритмы поиска элементов по значению

### Последовательный поиск

Предположим, что у нас есть некоторая последовательность значений и стоит задача определить, есть ли в этой последовательности некое заданное значение и на каком месте оно находится. Какой алгоритм необходим, чтобы решить такую задачу?

Конечно, первое, что приходит на ум, — это просто перебрать последовательно все значения. Такой алгоритм называется **«последовательным поиском»**.

И это единственно возможный алгоритм, если расположение элементов никак не определено. Действительно, искомый элемент может находиться в любой ячейке, и в процессе проверки одних элементов мы ничего не можем сказать о других.

Если мы имеем массив длиной  $N$ , то при последовательном поиске в зависимости от того, на каком месте расположен искомый элемент, может понадобиться от 1 до  $N$  сравнений. Значит, в худшем случае сложность этого алгоритма составляет  $O(N)$ .

Для небольших массивов это не существенно, но для последовательностей, длиной в сотни тысяч значений и более, это становится проблематичным в связи со значительным увеличением времени поиска.

Таким образом, основной недостаток последовательного поиска — низкая эффективность.

Но, тем не менее, часто приходится использовать этот способ.

Для последовательного поиска в классе Collections есть метод contains(). Он возвращает boolean в зависимости от того, есть элемент в коллекции или нет.

В классе Arrays такого класса нет. Можно воспользоваться обычным циклом. Например, код:

```
int ind = -1;
for (int i = 0; i < array.length; i++) {
    if (array[i] == k) {
        ind = k;
        break;
    }
}
```

В переменную  $ind$  попадет индекс первого элемента массива  $array$ , равного  $k$  или  $0$ , если элемент не будет обнаружен.

Таким образом, сложность последовательного поиска — линейная.

### Трудоёмкость последовательного поиска

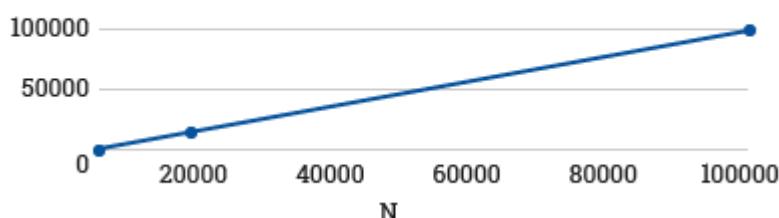


Рис. 4.2.

### Двоичный поиск

Для поиска значений в упорядоченных последовательностях (в отсортированных по возрастанию или убыванию массивах и коллекциях) наиболее эффективно использовать алгоритм двоичного (бинарного) поиска.

Идея алгоритма проста.

Сравним средний элемент с индексом  $mid$  (если количество элементов четно, то любой из двух центральных) с искомым значением.

Если этот элемент больше искомого, то продолжить поиск в левой части (с индексами от 0 до  $mid-1$ ), иначе в правой части (с индексами от  $mid$  до последнего).

Снова находим середину и делаем аналогично.

Так делаем до тех пор, пока рассматриваемая часть будет содержать один единственный элемент.

Можно сравнивать в процессе поиска элементы еще и *на равенство* среднему, но это не обязательно, метод работает очень быстро, а это добавит лишние проверки, что в среднем случае замедлит его работу.

Если он равен искомому, мы нашли его на месте  $mid$ ; если нет, то такого элемента нет, а если бы он был, он должен был бы стоять  $mid$ . Последнее утверждение очень важно, оно позволяет организовать приближенный поиск, значение, которое наиболее близко к искомому. Если элемент не найден, нужно рассмотреть соседние с  $mid$ .

В классах Arrays и Collections есть методы:

```
int binarySearch(type_[] a, int key)
```

И методы, которые принимают границы диапазонов:

```
binarySearch(type_[] a, int low, int hi, int key)
```

Для их использования массивы и коллекции должны быть предварительно отсортированы.

Посмотрим реализацию этого алгоритма в классе Arrays для массива int.

Фрагмент Arrays.java

```

...
311:    public static int binarySearch(int[] a, int low, int hi, int key)
312:    {
313:        if (low > hi)
314:            throw new IllegalArgumentException("The start index is higher than " +
315:                                            "the finish index.");
316:        if (low < 0 || hi > a.length)
317:            throw new ArrayIndexOutOfBoundsException("One of the indices is out " +
318:                                            "of bounds.");
319:        int mid = 0;
320:        while (low <= hi)
321:        {
322:            mid = (low + hi) >>> 1;
323:            final int d = a[mid];
324:            if (d == key)
325:                return mid;
326:            else if (d > key)
327:                hi = mid - 1;
328:            else
329:                // This gets the insertion point right on the last loop.
330:                low = ++mid;
331:        }
332:        return -mid - 1;
333:    }
...

```

Обратите внимание, что он возвращает, если элемент не найден. Возвращается  $-mid - 1$ , благодаря этому, если элемент не найден, мы получаем отрицательное значение того индекса, где элемент должен был быть. Единица вычитается для того, чтобы снять неоднозначность, когда элемент должен был стоять в начале последовательности (индекс 0).

Какова же сложность алгоритма двоичного поиска?

На каждой следующей итерации наш диапазон поиска сокращается вдвое (на самом деле это не совсем так, если рассматриваемая часть содержит нечетное количество элементов, но можно доказать, что это несущественно).

То есть мы делим  $N$  на 2, потом еще раз на 2 и так до тех пор, пока диапазон не станет пустым. Значит максимальное количество сравнений, которое можно выполнить на последовательности  $N$  — это  $\log_2 N$ , поэтому пишут, что сложность двоичного поиска равна  $O(\log_2 N)$ .

На графике (рис. 4.3) приведена зависимость количества действий двоичного поиска от длины последовательности. Сравните ее с количеством действий последовательного поиска при длине 100 тысяч!

### Трудоёмкость двоичного поиска

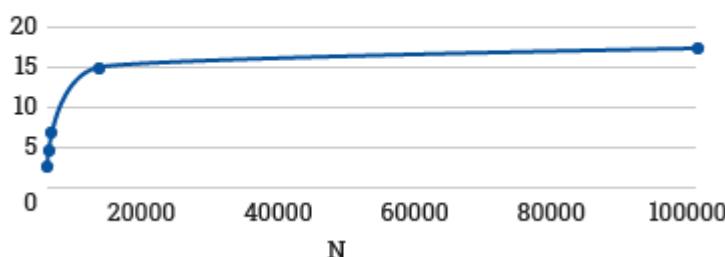


Рис. 4.3.

Таким образом:

- достоинство двоичного поиска — высокая эффективность;
- недостаток — работает только на упорядоченной последовательности, требует предварительной сортировки.

Приведем пример использования метода `binarySearch()`:

```
// инициализируем массив  
int intArr[] = {30, 20, 5, 12, 55};  
  
// сортируем его  
Arrays.sort(intArr);  
  
int retVal = Arrays.binarySearch(intArr, 12);  
System.out.println("Элемент с о значением 12 находится по индексу " + retVal);
```

Будет выведено:

```
Элемент с о значением 12 находится по индексу 1
```

## **4.2. Связные списки. Очереди, стеки, деки**

Сайт: IT Академия SAMSUNG

Курс: MDev @ IT Академия Samsung

Книга: 4.2. Связные списки. Очереди, стеки, деки

Напечатано:: Егор Беляев

Дата: Суббота, 18 Апрель 2020, 19:28

# **Оглавление**

- 4.2.1. Связные списки. Класс LinkedList
- 4.2.2. Сравнение ArrayList и LinkedList
- 4.2.3. Коллекции. Интерфейс List
- 4.2.4. Стеки, очереди, деки

## 4.2.1. Связные списки. Класс LinkedList

При создании массивов память выделяется последовательно. И это огромный плюс. Зная адрес в памяти начального элемента и тип элементов массива (что эквивалентно знанию о размере байтах на каждый элемент), программа может с легкостью найти любой элемент массива. Это делается при помощи простой формулы: к адресу начала массива прибавляется индекс элемента, умноженный на размер элемента в байтах.



Поэтому в языке Си можно даже использовать альтернативную форму обращения, вместо  $a[5]$  писать  $5[a]$ .

Дело в том, что имя массива в этом языке является фактически адресом первого, по индексу нулевого элемента. А благодаря арифметике указателей, если прибавить к указателю целое число, мы получим указатель (фактически адрес) элемента, который расположен на столько элементов правее (или левее, если число отрицательное).

Таким образом,  $a[5]$  транслируется в  $*(a + 5)$  А  $5[a]$  переводится  $*(5 + a)$ . Оба выражения, естественно, равны и означают «элемент на пять элементов правее начала массива», то есть просто «пятый элемент».

Таким образом, важное и серьезное преимущество массивов — это быстрая индексация, быстрый поиск элемента по индексу со сложностью  $O(1)$ .

Но тот же факт, что память выделяется последовательно, определяет и недостатки: нельзя добавлять ячейки в массив при необходимости и непросто вставлять элементы. Даже если последние элементы массива не используются, то есть свободны, чтобы вставить элемент в начало, нужно передвинуть, скопировать все элементы со сдвигом вправо. И при большом количестве элементов эта операция становится очень длительной.

### Связные списки

Указанных недостатков лишены связные списки. В них можно быстро, за константное количество действий, то есть со сложностью  $O(1)$  добавлять и удалять элементы в любое место.

Это дается ценой медленной адресации. В списках наоборот поиск элемента ведется по индексу, а не только по значению делается за  $O(N)$ , где  $N$  — текущая длина списка.



**Связный список** — это динамическая структура, состоящая из элементов, каждый из которых содержит в себе данные и ссылки (одну или две) на последующий или (и) предыдущий элемент. Один элемент при этом может содержать информацию о том, что он последний элемент в списке, и один элемент — что он первый в списке.

Таким образом, связный список — это цепочка разрозненных кусочков памяти. Причем во время выполнения программы элементы могут не только добавляться, но и удаляться, переставляться. В случае с массивами память выделяется сразу под весь массив, а в случае списков по мере добавления/удаления элементов.

Элемент связного списка всегда содержит информацию об адресе последующего элемента. При этом, если список — это незамкнутая цепочка, то есть последний элемент списка в качестве адреса следующего элемента содержит пустую ссылку (null). Такой список называют линейным.

Если же последний элемент списка в поле ссылки на следующий элемент содержит адрес головы списка (первого элемента), то такой список называют циклическим.

Далее мы будем вести речь только о линейных списках.

В зависимости от того, имеют ли узлы две связи (на предыдущий и на следующий или только одну (только на следующий), выделяют односвязный и двусвязный списки.

**Односвязные списки** — это списки, в которых каждый элемент имеет только одну ссылку, как правило, на следующий элемент в данной структуре данных.

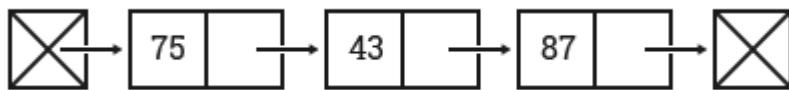


Рис. 4.4.

**Двусвязные списки** — это списки, в которых каждый элемент имеет две ссылки: на предыдущий и следующий за ним элементы списка.

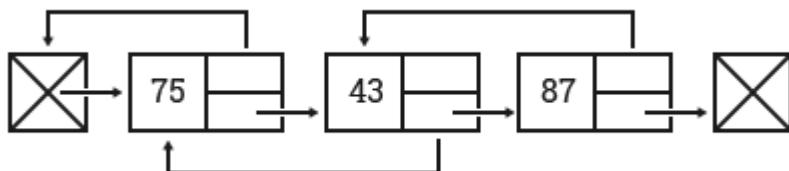


Рис. 4.5.

Таким образом, каждый элемент односвязного списка «знает» только элемент, который стоит после него, а каждый элемент двусвязного списка еще дополнительно «знает» элемент, который стоит перед ним. Следовательно, в первом случае мы можем двигаться по списку только в одном направлении от головы списка к хвосту, тогда как во втором — в обоих направлениях.

## Простой связный список

Для того чтобы лучше понять устройство связных списков, разработаем класс простого односвязного списка.

```
public class MyLinkedList<T> implements Iterable<T>
```

Будем сразу делать его параметризованным. Это позволит сделать класс более безопасным и удобным в использовании, и при этом никаких сложностей при этом не возникнет.

Список состоит из узлов («коробок с веревочками»). Удобно оформить их в виде вложенного класса:

```
private class Node {  
    private T data;  
    private Node next;  
  
    public Node(T data, Node next) {  
        this.data = data;  
        this.next = next;  
    }  
}
```

Узлы очень просты. Они содержат данные и ссылку на следующий элемент.

Главное здесь понимать, что узел не содержит другие узлы (в этом случае возникла бы бесконечная рекурсия), а лишь ссылки на них («веревочки», а не сами «коробки»).

В списке удобно хранить ссылки на первый и последний элементы. При этом получается эффективная работа в начале и в конце списка. Этот конец не надо будет искать для добавления в конец списка.

```
private Node first;  
private Node last;  
  
public MyLinkedList() {  
    first = null;  
    last = null;  
}
```

Такой конструктор необязателен, по умолчанию переменные `first` и `last` будут проинициализированы значением `null`. Мы его написали для наглядности.

Проверку на пустоту сделать очень просто:

```
public boolean isEmpty() {  
    return (first == null);  
}
```

При добавлении в конец и в начало списка два случая: список пустой или непустой.

```
public void add(T item) {  
    if (!isEmpty()) {  
        Node prev = last;  
        last = new Node(item, null);  
        prev.next = last;  
    } else {  
        last = new Node(item, null);  
        first = last;  
    }  
}  
  
public void addFirst(T item) {  
    if (!isEmpty()) {  
        Node next = first;  
        first = new Node(item, null);  
        first.next = next;  
    } else {  
        first = new Node(item, null);  
        last = first;  
    }  
}
```

Напишем еще в качестве примера метод удаления элемента по значению. Сначала его нужно найти, а потом рассмотреть случаи, когда удаляется последний элемент в списке, удаление происходит из начала, из конца и из середины.

```

public boolean remove(T item) {
    if (isEmpty()) {
        throw new IllegalStateException(
            "Cannot remove() from an empty list.");
    }
    boolean result = false;
    Node prev = null;
    Node curr = first;
    // ищем элемент
    while (curr.next != null && !curr.data.equals(item)) {
        prev = curr;
        curr = prev.next;
    }
    // если нашли - удаляем
    if (curr.data.equals(item)) {
        // удаляем последний элемент в списке
        if (first == last) {
            first = null;
        }
        // удаляем первый элемент
        else if (curr == first) {
            first = first.next;
        }
        // удаляем последний элемент
        else if (curr == last) {
            last = prev;
            last.next = null;
        }
        // удаляем из середины
        else {
            prev.next = curr.next;
        }
        result = true;
    }
    return result;
}

```

Остается реализовать итератор:

```

public Iterator<T> iterator() {
    return new Itr();
}

```

Как и для MyArrayList сам итератор удобно оформить вложенным классом:

```
private class Itr implements Iterator<T> {
    private Node cursor = first;
    @Override
    public T next() {
        T item = cursor.data;
        cursor = cursor.next;
        return item;
    }
    @Override
    public boolean hasNext() {
        return cursor != null;
    }
    @Override
    public void remove()
    }
}
```

Для краткости мы оставили реализацию метода `remove()` пустой. Ее несложно реализовать по аналогии с методом `remove()` самого списка.

Испытаем разработанный односвязный список.

```
MyLinkedList<String> myLinkedList = new MyLinkedList<>();
myLinkedList.add("one");
myLinkedList.add("two");
myLinkedList.add("three");
myLinkedList.addFirst("zero");
myLinkedList.remove("three");

for (String item : myLinkedList){
    System.out.print (item + " ");
}
```

Будет выведено:

```
zero one two
```

## Библиотечный класс `LinkedList`



В стандартной библиотеке Java есть класс, который реализует связный список.

Рассмотрим некоторые методы этого класса.

## Методы класса LinkedList

Символ E в таблице 4.1 означает тот тип, которым параметризован LinkedList.

Метод	Описание
LinkedList()	Конструктор, создающий пустой список
boolean add(E object)	Добавляет элемент в конец списка
void add(int location, E object)	Добавляет элемент в указанное место в списке. Выполняется за $O(N)$
void addFirst(E object)	Добавляет элемент в начало списка
void addLast(E object)	Добавляет элемент в конец списка. Данный метод эквивалентен методу add
void clear()	Очищает список от всех элементов.
Object clone()	Возвращает клонированный список типа LinkedList
boolean contains(Object object)	
E get(int location)	Возвращает элемент по соответствующему номеру. Выполняется за $O(N)$
E getFirst()	Возвращает первый элемент из списка
E getLast()	Возвращает последний элемент из списка
E peek()	Возвращает, но не удаляет первый элемент в случае пустого списка возвратит null
E peekLast()	Получает, но не удаляет последний элемент списка
E poll()	Получает и удаляет первый элемент из списка. Если ничего нет, то возвращает null
E pollLast()	Получает и удаляет последний элемент из списка. Если ничего нет, то возвращает null
int size()	Возвращает число элементов в списке

Метод	Описание
T[] toArray (T[] contents)	Возвращает массив элементов из списка. Используется, когда нам нужно использовать данные не в виде списка, а в виде массива. Если массив, указанный в параметрах способен вместить все элементы списка, то будет он использоваться, иначе создается новый массив
Object[] toArray ()	Аналогичен предыдущему методу, но тут сразу массив создается без массива от пользователя

Табл. 4.1.

Всего методов значительно больше, чем представлено в таблице. В следующих параграфах мы обсудим их более системно.

Приведем несколько примеров использования LinkedList.

### Пример 1. Простейшее использование класса LinkedList

В примере создаем список, добавляем два элемента и выводим список.

```
import java.util.LinkedList;

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

        LinkedList list = new LinkedList(); // Создаем список
        list.add("11"); // Добавляем новый элемент
        list.add("22"); // Добавляем новый элемент

        System.out.println(list); // Выводим список
    }
}
```

На выводе получим:

```
[11, 22]
```

### Пример 2. Использование основных методов класса

```
//Создаем новый список
LinkedList <String> list = new LinkedList();

//Добавляем новые элементы
list.add("11");
list.add("22");
list.add("33");
list.add("44");
list.add("55");
list.add("22");
list.addLast("Z");// Добавим в конец списка
list.addFirst("A");// Добавим в начало списка
list.add(1, "B");// Добавим вторым элементом

//Выводим список
System.out.println("Список:" + list);

list.remove("22");//Удалим первый элемент равный "22"
list.remove(2);//Удалим третий элемент
System.out.println("Список:" + list);//Выводим список

list.removeFirst();//Удалим первый элемент
list.removeLast();//Удалим последний элемент
System.out.println("Список:" + list);//Выводим список

String val = list.get(2);//Получим третий элемент из списка
System.out.println("val = " + val);
list.set(2, "Привет!"); //Изменим третий элемент
System.out.println("Список:" + list); //Выводим список
```

На выводе получим:

```
Список:[A, B, 11, 22, 33, 44, 55, 22, Z]
Список:[A, B, 33, 44, 55, 22, Z]
Список:[B, 33, 44, 55, 22]
val = 44
Список:[B, 33, Привет!, 55, 22]
```

## 4.2.2. Сравнение ArrayList и LinkedList

С точки зрения функционала данные классы очень похожи.

Важно понимать, что хотя многие методы выглядят одинаково, но из-за разницы во внутренней реализации скорость их работы будет сильно различаться. При больших размерах списков и объемах памяти для каждого элемента разница будет очень существенной. Особенно это заметно при работе с визуальными компонентами для ресурсоограниченных мобильных приложений. Грамотный выбор реализации списка основывается на том, какие операции будут чаще всего вызываться и какие из них для нас критичнее.

### Сложность выполнения операций в ArrayList и LinkedList

Сложность операций для классов ArrayList и LinkedList представлена в таблице 4.2.

Везде N — это количество элементов.

Операция	ArrayList	LinkedList
get(int index)	O(1)	O(N/4) в среднем
add(E element)	O(N/2) в среднем	O(1) *
add(int index, E element)	O(N/2) в среднем	O(N/4) в среднем
remove(int index)	O(N/2) в среднем	O(n/4) в среднем
Iterator.remove(int index, E element)	O(N/2) в среднем	O(1)
ListIterator.add(E element)	O(N/2)	O(1)

Табл. 4.2.

Кроме случаев, когда необходимо изменить длину внутреннего массива, в этих случаях O(N).

### Для расширяемых массивов

O(n/2) в среднем, но O(1) в лучшем случае (в конце массива), O(n) в худшем случае (в начале массива).

### Для списков

O(n/4) в среднем, но O(1) в лучшем случае (в начале списка), O(n/2) в худшем случае (в середине списка). Проведем эксперимент.

Сравним время добавления 100 000 элементов в конец, в начало и в середину в объекты ArrayList и LinkedList.

```
ArrayList<String> testList = new ArrayList<>();
//LinkedList<String> testList = new LinkedList<>();
long startTime = System.currentTimeMillis();
for (int i = 0; i < 100000; i++){
    testList.add("test");
    //testList.add(0, "test");
    //testList.add(testList.size() / 2, "test");
}
long finishTime = System.currentTimeMillis();
System.out.println("Worktime: " + (finishTime - startTime) + "ms");
```

Эту программу нужно запустить шесть раз, комментируя и раскомментируя соответствующие строки.

Добавление	Время в миллисекундах
ArrayList в конец	22 ms
LinkedList в конец	26 ms
ArrayList в начало	4315 ms (более четырех секунд)
LinkedList в начало	37 ms
ArrayList в середину	2152 ms
LinkedList в середину	29816 ms (почти тридцать секунд!)

Табл. 4.3.

Таблица 4.3 наглядно демонстрирует, что оба класса быстро добавляют в конец. У LibnkedList это происходит несколько медленнее, потому что время тратится на изменение ссылок. Добавление в начало для ArrayList трудная операция, а LinkedList, наоборот, работает очень быстро.

И наконец, добавление в середину отлично демонстрирует факт, что поиск элемента по индексу для LinkedList — очень длительная операция.

Надо сказать, что, если бы у нас был итератор, указывающий на середину, добавление в середину списка работало бы тоже очень быстро.

Программа:

```
LinkedList<String> testList = new LinkedList<>();
ListIterator<String> itr = testList.listIterator();
long startTime = System.currentTimeMillis();
for (int i = 0; i < 100000; i++){
    itr.add("test ");
    // поддерживаем итератор в середине списка
    if (i % 2 != 0) itr.previous();
}
long finishTime = System.currentTimeMillis();
System.out.println("Worktime: " + (finishTime - startTime) + "ms");
```

Дает результат 30 ms. Это более чем в семьдесят раз быстрее ArrayList!

Таким образом, ArrayList хорошо подходит в качестве замены массива, и в данном классе обеспечивается очень быстрый доступ к чтению элементов. Тогда как в LinkedList для этого требуется гораздо больше времени.

Добавление элементов в начало списка является сильной стороной LinkedList.

Кроме сложности операций данные классы можно сравнить по объему памяти, которая тратится на хранение элементов. Если ArrayList хранит фактически только сами элементы, то LinkedList хранит в каждом узле еще по две ссылки.

На рисунке 4.6 приведены графики требуемого объема памяти от количества элементов.

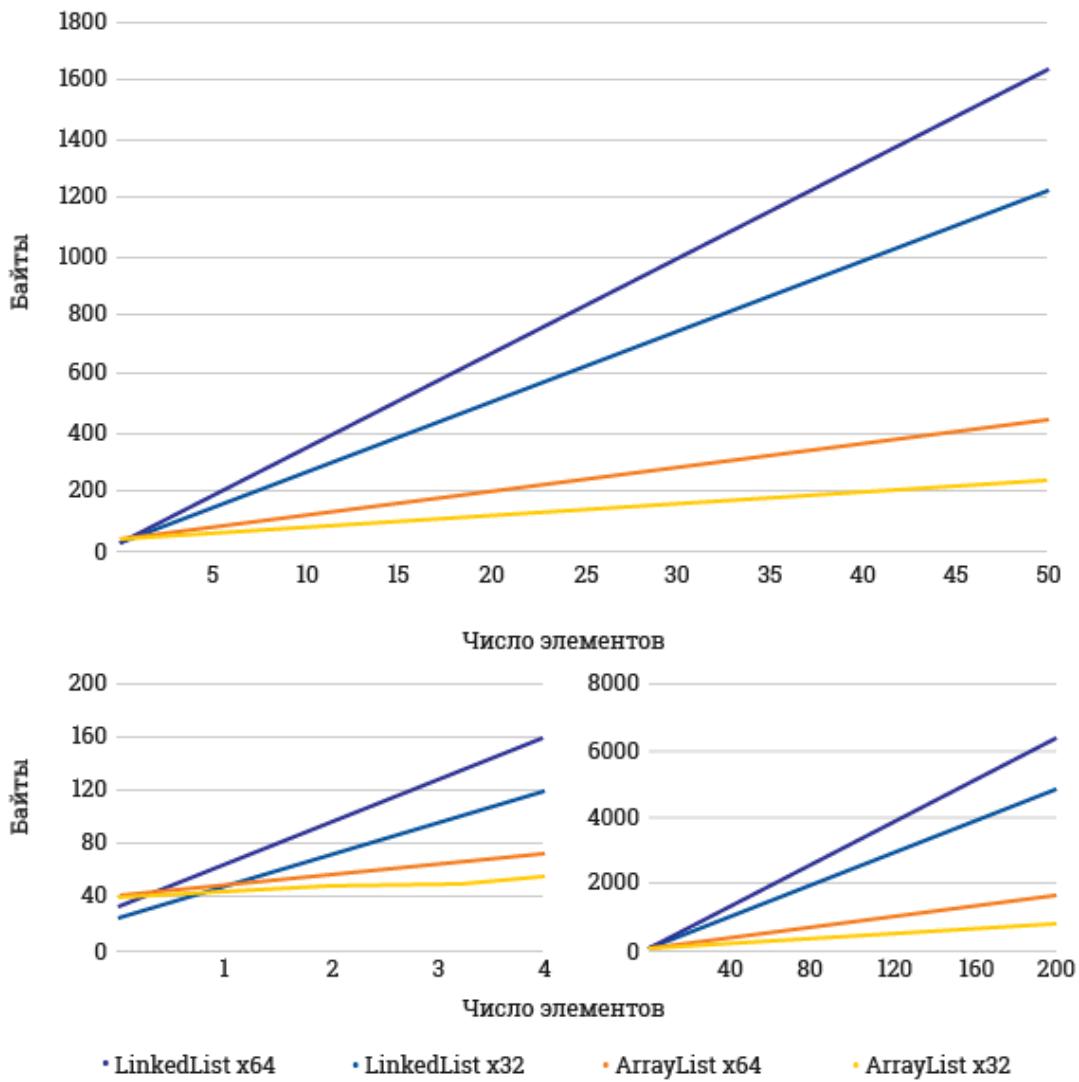


Рис. 4.6.

Наглядно видно, что `LinkedList` занимает гораздо больше места в памяти.

На практике класс `LinkedList` используют не часто, и среди программистов бытует мнение, что его выгодно применять только в определенных ситуациях, в первую очередь, где требуется именно двусвязный список.

## 4.2.3. Коллекции. Интерфейс List

ArrayList и LinkedList имеют много одинаковых методов. Они по-разному реализованы, работают с разной скоростью, но сигнатура у них одна. Так происходит, потому что оба класса являются коллекциями и реализуют интерфейс List.

### Коллекции

В Java существует большое разнообразие классов, реализующих структуры данных, поэтому для универсальности их представления введено понятие коллекции.

**Коллекция** — абстрактная структура данных, которая оперирует группой объектов. Использование коллекций дает широкие возможности при программировании, так как каждая коллекция эффективно справляется со своей задачей, поэтому, чтобы не «изобретать велосипед», программист должен знать основные коллекции языка, который он изучает.

Стандартные коллекции расположены в пакете `java.util`.

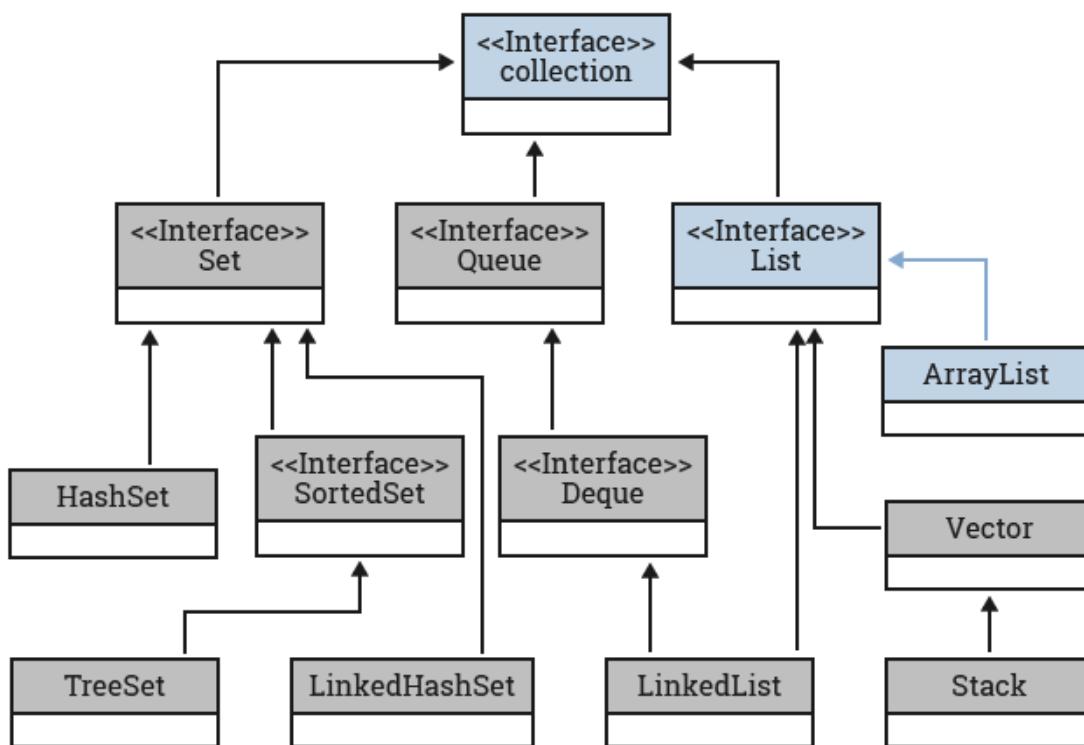


Рис. 4.7.

### Интерфейс Collection

В основе иерархии лежит интерфейс Collection, который описывает коллекцию объектов без какой-то конкретики по реализации данного набора объектов. Не следует путать его с классом Collections, в котором собраны алгоритмы для обработки коллекций. Все конкретные коллекции реализуют именно его в первую очередь.

Метод	Описание
<code>boolean add(E object)</code>	Добавляет элемент в конец списка
<code>boolean addAll(Collection collection)</code>	Добавляет в конец списка все элементы из коллекции. То есть из какой-то структуры данных вставляет все элементы
<code>void clear()</code>	Очищает список от всех элементов

Метод	Описание
boolean contains(Object object)	Проверяет наличие элемента в параметрах в данном списке
E remove ()	Удаляет первый элемент в списке. Эквивалентен методу removeFirst
boolean remove (Object object)	Удаляет один экземпляр элемента object в списке
int size ()	Возвращает число элементов в списке
T[] toArray (T[] contents)	Возвращает массив элементов из списка. Применяется, когда нужно использовать данные не в виде списка, а в виде массива. Если массив, указанный в параметрах, способен вместить все элементы списка, то будет использоваться он, иначе создается новый массив
Object[] toArray ()	Аналогичен предыдущему методу, но тут сразу массив создается без массива от пользователя

Табл. 4.4.

Их реализуют все коллекции. Таким образом, каждая коллекция имеет размер, который можно узнать, методы для добавления и удаления в нее элементов и других коллекций, коллекцию можно обойти при помощи итератора и так далее.

## Интерфейс List

Классы ArrayList и LinkedList реализуют интерфейс List. Этот интерфейс расширяет интерфейс Collections, поэтому они являются коллекциями. Здесь слово «список» имеет более широкое значение, чем словосочетание «связный список». Интерфейс List содержит методы, определяющие упорядоченную коллекцию.

К элементам коллекций, реализующих интерфейс List, можно обращаться по индексам.

Перечислим важные методы интерфейса List.

Метод	Описание
void add(int location, E object)	Добавляет элемент в указанное место в списке
boolean addAll(int location, Collection collection)	Добавляет все элементы из коллекции в указанное место
E get(int location)	Возвращает элемент по соответствующему номеру
int indexOf(Object object)	Проверяет наличие элемента в параметрах в данном списке и возвращает индекс первого вхождения в списке
int lastIndexOf(Object object)	Аналогичен предыдущему методу, но возвращает номер последнего вхождения элемента в списке. Если элемент встречается в списке только один раз, то оба метода вернут одно и то же число
ListIterator listIterator(int location)	Возвращает список-итератор данного списка с указанного индекса
ListIterator listIterator()	Аналогично предыдущему методу, но тут начинается обход с первого элемента
E set (int location, E object)	Заменяет содержимое элемента списка под соответствующим индексом

Табл. 4.5.

Несмотря на то что все коллекции, которые реализуют интерфейс List, снабжаются индексами, обращаться к элементам по индексам нужно с осторожностью. В некоторых реализациях такое обращение работает крайне медленно.

## Итератор ListIterator

Интерфейс List обязывает коллекции, которые его реализуют, возвращать особый вид итератора ListIterator. Он расширяет интерфейс Iterator. В дополнение к возможностям «обычного» итератора ListIterator имеет методы:

- void add(E obj) вставляет объект obj перед элементом, который должен быть возвращен следующим вызовом next();
- boolean hasPrevious() возвращает true, если в коллекции имеется предыдущий элемент, иначе возвращает false;
- E previous() возвращает предыдущий элемент, если такого нет, то генерируется исключение NoSuchElementException;
- int nextIndex() возвращает индекс следующего элемента. Если такого нет, то возвращается размер списка;
- int previousIndex() возвращает индекс предыдущего элемента. Если такого нет, то возвращается число -1;
- void set(E obj) присваивает текущему элементу, выбранному вызовом методов next() или previous(), ссылку на объект obj.

То есть «умеет» двигаться назад, добавлять новый элемент в коллекцию «перед собой» и изменять значение текущего элемента.

Благодаря использованию итератора ListIterator можно сделать работу со списком LinkedList очень эффективной. Например, в предыдущем параграфе мы быстро добавляли элементы в середину списка.

## 4.2.4. Стеки, очереди, деки

В реальных задачах от структуры данных редко нужно много возможностей при работе с контейнером.

При реализации многих алгоритмов часто используют контейнеры, в которых элементы добавляются только в конец, а забираются только из начала — *очереди*. Или, наоборот, добавляются только в конец и забираются только из конца — так называемые *стеки*. Реже нужно уметь добавлять в оба конца и забирать из обоих концов. Так работают *дву направленные очереди (деки)*.

### Очередь

*Очередь* — это список, в котором добавление элементов происходит в хвост (конец списка), а считывание элементов происходит с головы. Такая дисциплина обслуживания «*первым пришел — первым ушел*» носит название FIFO (First in — first out).

Это полностью соответствует обычной очереди людей, например, в кассу супермаркета. Когда люди подходят, то они встают в конец очереди, а кассиры обслуживаются тех, кто первыми стоит в данной очереди.

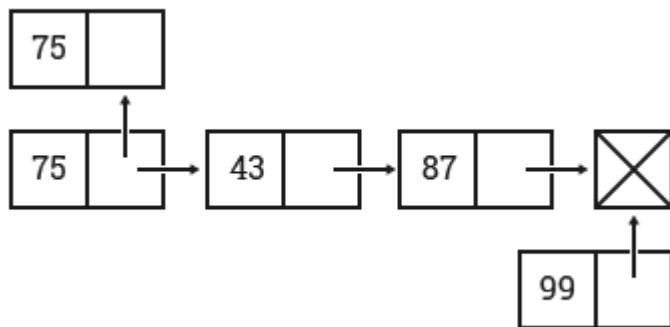


Рис. 4.8.

### Стек

*Стек* — это список, в котором добавление и считывание реализуется в соответствии с обратной очередью обслуживания «*последним пришел — первым ушел*». Этот принцип носит название LIFO (Last in — first out). Чтобы понять работу стека, представьте стеклянную колбу, в которую закладываются цветные шайбы (см. рис. 4.9). Если положить первой желтую шайбу, а потом синюю, то понятно, что желтую мы можем достать только после синей.



Рис. 4.9.

Получается стопка шайб, в которой сверху лежит последняя положенная шайба. В стеке принято называть ее вершиной стека. При реализации стека с помощью списков удобно в качестве указателя на голову списка использовать указатель на вершину. И тогда добавление и удаление элементов в стек легко реализуется со стороны «головы» и нет необходимости пробегать весь список.

Другой пример стека — это магазин автомата с патронами. Последним сработает патрон, который в магазин попал самым первым.

В программировании общеизвестно понятие стека вызовов функций. Его работа, как следует из названия, также основана на структуре данных «стек».

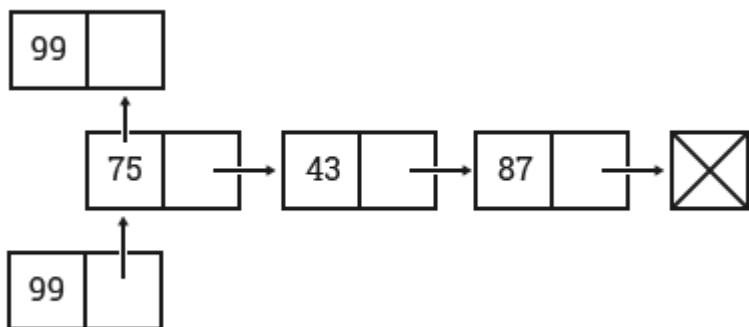


Рис. 4.10.

Нельзя сказать, что очередь лучше стека или наоборот — для решения каждой конкретной задачи выбирается та структура данных, которая подходит лучше.

## Интерфейсы Queue и Deque

Для организации очередей в Java есть интерфейс Queue, он расширяет интерфейс Collection. Это естественно, потому что очереди — это, конечно, коллекции.

Специфические для очереди методы:

- poll() возвращает первый элемент и удаляет его из очереди;
- peek() возвращает первый элемент очереди, не удаляя его;
- offer(Object obj) добавляет в конец очереди новый элемент и возвращает true, если вставка удалась.

Методы element() и remove() работают аналогично методам poll() и peek(), но если очередь пуста, возвращают исключение.

Интерфейс Deque расширяет Queue. Он дополняет практически каждый метод еще двумя суффиксами First и Last. Например, peekFirst() и peekLast().

Класс LinkedList реализует не только интерфейс List, но и интерфейс Deque. Поэтому с ним можно работать с очередью или с двунаправленной очередью.

Есть и другой класс, реализующий Deque, — ArrayDeque. Он реализует дек на массиве и поэтому обычно эффективнее: потребляет меньше памяти и работает быстрее.



Для того чтобы реализовать очередь (дек) на массиве, его «закольцовывают». Если «хвост» доходит до конца массива, его продолжают в начале массива. Аналогично делают с «головой».

Подробнее об этой и других реализациях очереди, например, при помощи двух стеков, можно почитать в Викиконспекте.

## Класс Stack

В Java нет интерфейса, определяющего функциональность стека. Класс Stack реализует интерфейс List и имеет дополнительные методы.

Метод	Описание
boolean empty()	Проверка стека на наличие элементов — он возвращает true, если стек пуст
E peek()	Возвращает верхний элемент, не удаляя его из стека
E pop()	Извлекает верхний элемент, удаляя его из стека
E push(E item)	Помещает элемент в вершину стека
int search(Object o)	Ищет заданный элемент в стеке, возвращая количество операций pop(), которые требуются для того, чтобы перевести искомый элемент в вершину стека. Если заданный элемент в стеке отсутствует, этот метод возвращает -1

Табл. 4.5.

Зачем используют специальный класс Stack, если все эти методы реализованы ArrayList, но с другими именами. Класс Stack наследуется от класса Vector, который имеет ту же функциональность, что и ArrayList, но при этом потокобезопасен, то есть с ним можно работать из разных потоков. Поэтому Stack можно использовать в многопоточной программе. Но и сами имена методов играют немаловажную роль. Программистам значительно удобнее работать со стеками при помощи привычных push() и pop().

Кстати, будьте внимательны. Метод empty() — проверка на пустоту именно в классе Stack. В других классах это может быть очистка, опустошение контейнера. Лучше использовать везде isEmpty() из интерфейса Collections.

Еще раз приведем главные методы очередей и стеков.

В Queue метод poll() возвращает и удаляет первый элемент в списке. В Stack метод pop() возвращает и удаляет последний элемент в списке. В Queue метод offer(), а в Stack метод push(), вставляют элемент в конец списка. В Queue метод peek() возвращает первый элемент в списке, а в Stack метод с таким же именем возвращает последний элемент списка.

И рассмотрим пример.

## Демонстрация работы очередей и стеков

```
ArrayDeque<String> queue = new ArrayDeque<>();
Stack<String> stack = new Stack<>();
String[] strings = { "first", "left", "center", "right", "last" };
for (String str : strings) {
    queue.offer(str);
    stack.push(str);
}
System.out.println("From queue:");
while (!queue.isEmpty())
    System.out.print(queue.poll() + " ");
System.out.println("\nFrom stack:");
while (!stack.isEmpty())
    System.out.print(stack.pop() + " ");
```

Будет выведено:

```
From queue:
first left center right last
From stack:
last right center left first
```

Для очереди в прямом порядке, для стека — в обратном.

## 4.3. Списки в Android. Адаптеры

Сайт: IT Академия SAMSUNG

Курс: MDev @ IT Академия Samsung

Книга: 4.3. Списки в Android. Адаптеры

Напечатано:: Егор Беляев

Дата: Суббота, 18 Апрель 2020, 19:29

# **Оглавление**

4.3.1. Введение

4.3.2. Список из ресурсов

4.3.3. ArrayAdapter. Простой пример

4.3.4. ArrayAdapter. Собственная разметка

Задание по Android-практикуму

## 4.3.1. Введение

Одним из наиболее часто используемых элементов интерфейса являются списки. Их реализация при создании Android-приложений имеет ряд особенностей: для вывода списка используется компонент ListView, а чтобы определить его содержание и структуру — связанный с ним адаптер.

Зачем так сложно реализовано? Почему бы просто не передать данные ListView напрямую? Чтобы экономно расходовать оперативную память. Дело в том, что мобильные устройства обладают малым количеством оперативной памяти, а операционная система Android использует всю имеющуюся (не важно сколько ее на устройстве), так как она построена вокруг виртуальной машины Java, которая сама решает, когда очищать ресурсы. Причем если виртуальная машина Java посчитает, что приложение использует много ресурсов, то она его закроет.

В этой теме мы рассмотрим ArrayAdapter, при изучении баз данных CursorAdapter, а после изучения ассоциативных массивов SimpleAdapter.

## 4.3.2. Список из ресурсов

Одним из наиболее часто используемых элементов интерфейса являются списки. Небольшой список можно организовать очень просто. Создать массив строк в ресурсах и указать его в ListView в качестве источника.

Например, создадим список Forbes.

Ресурсы к этому примеру располагаются по адресу <https://github.com/vv73/ForbesList>

В ресурсы добавим (лучше создать новый файл ресурсов) массив имен из таблицы. Например, в файле */values/list.xml* напишем.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string-array name="forbes">
        <item>Bill Gates</item>
        <item>Warren Buffett</item>
        <item>Jeff Bezos</item>
        <item>Amancio Ortega</item>
        <item>Mark Zuckerberg</item>
        ...
    </resources>
```

А в разметке активности создадим список ListView и в нем укажем свойство android:entries

```
<ListView
    android:id="@+id/list"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:entries="@array/forbes"
/>
```

Все готово. Даже id указывать необязательно. Но он может пригодиться в дальнейшем.



Bill Gates

Warren Buffett

Jeff Bezos

Amancio Ortega

Mark Zuckerberg

Carlos Slim Helu

Larry Ellison

Charles Koch

David Koch

Michael Bloomberg

Bernard Arnault

Larry Page

Sergey Brin



Рис. 4.11.

Список выглядит невзрачно (см. рис. 4.11). Для кастомизации внешнего вида и добавления дополнительной информации в элементы списка используют дополнительные объекты-адаптеры.

### 4.3.3. ArrayAdapter. Простой пример

Рассмотрим первый простой пример.

Требуется вывести название двенадцати месяцев на экран устройства в виде списка.

Реализуем это с помощью адаптера ArrayAdapter и ListActivity.

Самым простым способом вывода списка на экран является его вывод через ListActivity. В этом случае создается активность (файл разметки создавать не нужно), внутри которой на всю поверхность растянут компонент ListView. Внутри конструктора активности происходит заполнение списка.

Как уже было сказано, ArrayAdapter предназначен для работы с ListView, и данные представляет в виде массива, каждый элемент которого размещается в отдельном элементе TextView.

Списки выводятся с помощью ListView, а провайдерами (поставщиками) контента выступают адаптеры, которые готовят элементы списков для вывода на экран. Создадим новый проект, родителем главной активности назначим ListActivity. Далее создадим массив, в котором будут храниться месяцы:

```
String[] myArr = {"Январь", "Февраль", "Март", "Апрель", "Май", "Июнь", "Июль", "Август",
    "Сентябрь", "Октябрь", "Ноябрь", "Декабрь"};
```

Теперь для вывода списка на экран нужно:

- создать (при создании связать с данными) ArrayAdapter;
- назначить адаптер.

Это можно сделать в методе onCreate():

```
public class MainActivity extends ListActivity {

    String[] myArr = { "Январь", "Февраль", "Март", "Апрель", "Май", "Июнь", "Июль", "Август",
        "Сентябрь", "Октябрь", "Ноябрь", "Декабрь" };

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

        ArrayAdapter<String> monthAdapter =
            new ArrayAdapter<String>(this,
                android.R.layout.simple_list_item_1, myArr);

        setListAdapter(monthAdapter);
    }
}
```

Запускаем приложение — видим наш список на экране устройства (рис. 4.12).



Рис. 4.12.

Еще раз зададимся вопросом «зачем нужен адаптер?».

Обратимся к снимку экрана. На нем изображены всего шесть элементов списка. На самом деле система создала 7 визуальных экземпляров списка. При передвижении списка вверх или вниз система *не создает новые элементы*. Она *использует уже созданные*.

Представим, что пользователь начал водить по экрану пальцем и необходимо вывести новые элементы списка. Запускается следующий механизм: верхние элементы скрываются с экрана, заполняются новыми данными и отображаются внизу экрана. Если пронумеровать все визуализированные элементы, то можно получить следующую картину подмены (рис. 4.13).



Рис. 4.13.

Будет создано столько визуальных элементов списка, сколько вмещается на экран. Вернее, на один больше. Если бы этого механизма не существовало, то для 1 000 000 текстовых элементов массива системе пришлось бы создать 1 000 000 экземпляров TextView. Это повлекло бы крах приложения, так как память тут же бы закончилась.

Подмена и перезаполнение визуальных элементов происходит настолько быстро, что пользователь этого никогда не заметит.

## Реакция на выбор элемента

Как отследить пункт, на который нажал пользователь? Для этих целей в классе ListActivity реализован метод — `onListItemClick`. Достаточно его переопределить.

Например, будем выводить всплывающее сообщение с текстом, который отображается в выбранном элементе.

```
@Override  
protected void onListItemClick(ListView l, View v, int position, long id) {  
    String month = (String) getListAdapter().getItem(position);  
    Toast.makeText(this, month, Toast.LENGTH_SHORT).show();  
}
```

По позиции определяется строка в массиве, который лежит в адаптере и выводится текстом.

Обратите внимание на метод `getListAdapter()`, с его помощью реализуется доступ к адаптеру, с помощью которого осуществляется вывод информации на экран. С его помощью также можно получить доступ к данным, которые выводятся на экран, в частности с помощью метода `getItem()`.

Чтобы динамически изменить элемент отображаемого массива, нужно изменить массив и сообщить адаптеру об изменении.

```
monthArr[11] = "November";  
monthAdapter.notifyDataSetChanged();
```

С помощью метода `notifyDataSetChanged()` мы информируем адаптер о том, что элементы массива изменились и требуется обновить их отображение.

Добавьте этот код в слушатель клика.

## **4.3.4. ArrayAdapter. Собственная разметка**

Что делать, если необходимо вывести более сложный список элементов?

Например, с использованием нескольких надписей, изображениями и переключателями?

В этом случае можно использовать свой адаптер, например, пронаследованный от `ArrayAdapter`.

Работу можно разделить на следующие этапы:

- создание разметки;
- подготовка данных;
- создание класса адаптера;
- создание самого адаптера.

В качестве примера улучшим приложение с информацией о месяцах.

### **1. Разметка элементов списка**

Создадим разметку для адаптера в новом файле `adapter_item.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="wrap_content"
        android:layout_height="wrap_content" />

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Месяц"
        android:textAppearance="?android:attr/textAppearanceLarge" />

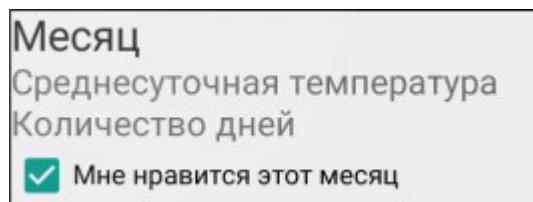
    <TextView
        android:id="@+id/textView2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Среднесуточная температура"
        android:textAppearance="?android:attr/textAppearanceMedium" />

    <TextView
        android:id="@+id/textView3"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Количество дней"
        android:textAppearance="?android:attr/textAppearanceMedium" />

    <CheckBox
        android:id="@+id/checkBox"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:checked="true"
        android:text="Мне нравится этот месяц" />

</LinearLayout>
```

Мы получим следующий вид:



## 2. Подготовка данных

В папку *res/drawable* сохраним файл *sun* с изображением солнышка.

Создайте новый файл класса:

```
public class MyMonth {  
    String month = ""; // Название месяца  
    double temp = 0.; // Средняя температура  
    int days = 0; // Количество дней  
    boolean like = true; // Нравится месяц  
}
```

Для заполнения массива месяцев (данные которого будут выводиться с помощью адаптера) в главной активности сформируем метод создания массива из объектов MyMonth.

Конструктор активности остается практически без изменений. Код главной активности:

```
package com.itschool.samsung.myadapter;

import android.app.Activity;
import android.os.Bundle;
import android.widget.ListView;

public class MainActivity extends Activity {

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

        MyMonthAdapter adapter = new MyMonthAdapter(this, makeMonth());
        ListView lv = (ListView) findViewById(R.id.listView);
        lv.setAdapter(adapter);
    }

    // Метод создания массива месяцев
    MyMonth[] makeMonth() {
        MyMonth[] arr = new MyMonth[12];

        // Названия месяцев
        String[] monthArr = {"Январь", "Февраль", "Март", "Апрель", "Май", "Июнь", "Июль", "Август",
                             "Сентябрь", "Октябрь", "Ноябрь", "Декабрь"};
        // Среднесуточная температура
        double[] tempArr = {-12.7, -11.3, -4.5, 7.7, 19.3, 23.9, 23.5, 22.8, 16.0, 5.2, -0.3, -9.3};
        // Количество дней
        int[] dayArr = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

        // Сборка месяцев
        for (int i = 0; i < arr.length; i++) {
            MyMonth month = new MyMonth();
            month.month = monthArr[i];
            month.temp = tempArr[i];
            month.days = dayArr[i];
            arr[i] = month;
        }
        return arr;
    }
}
```

### 3. Класс адаптера

Для того чтобы данные отображались правильно, нужно «научить» адаптер выводить информацию. Связать поля объекта `` с элементами разметки `list_item.xml`.

Для этого создадим собственный класс адаптера.

Приведем его целиком.

```
import android.content.Context;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ArrayAdapter;
import android.widget.CheckBox;
import android.widget.ImageView;
import android.widget.TextView;

public class MyMonthAdapter extends ArrayAdapter<MyMonth> {

    public MyMonthAdapter(Context context, MyMonth[] arr) {
        super(context, R.layout.adapter_item, arr);
    }

    @Override
    public View getView(int position, View convertView, ViewGroup parent) {

        final MyMonth month = getItem(position);

        if (convertView == null) {
            convertView = LayoutInflater.from(getContext()).inflate(R.layout.adapter_item, null);
        }

        // Заполняем адаптер
        ((TextView) convertView.findViewById(R.id.textView)).setText(month.month);
        ((TextView) convertView.findViewById(R.id.textView2)).setText(String.valueOf(month.temp));
        ((TextView) convertView.findViewById(R.id.textView3)).setText(String.valueOf(month.days));
        // Выбираем картинку для месяца
        if (month.temp > 0.)
            ((ImageView) convertView.findViewById(R.id.imageView)).setImageResource(R.drawable.sun);
        else
            ((ImageView) convertView.findViewById(R.id.imageView)).setImageResource(R.drawable.snow);

        CheckBox ch = (CheckBox) convertView.findViewById(R.id.checkBox);
        ch.setChecked(month.like);
        ch.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                month.like = ((CheckBox) v).isChecked();
            }
        });
        return convertView;
    }
}
```

Обратите внимание, что в конструктор адаптера передается массив объектов, данными которого и необходимо заполнить список, и внутри он передается в конструктор родителя.

В методе `getView()` происходит заполнение каждого выводимого элемента. В первой строке происходит выделение объекта `MyMonth`, данные которого будут выводиться. Метод `getItem` позволяет получить элементы используемого массива. В поле `position` сообщаем номер текущего создаваемого элемента. Далее происходит проверка, создан ли текущий экземпляр

адаптера или его необходимо создать. В следующих строках происходит заполнение надписей и выбор отображаемой картинки (картинка выбирается в зависимости от температуры месяца).

Далее происходит работа с галочкой. Она заполняется в соответствии с текущим состоянием поля like текущего месяца. Также на нее назначается слушатель события OnClick, который изменяет значение поля like текущего месяца, если пользователь нажал на галочку. Таким образом, в результате выполнения программы в массиве arr сохранено состояние визуальных элементов адаптера и это можно использовать в дальнейшей обработке.

Единственное, что нам надо переопределить, это метод getView(). Он вызывается списком (объектом ListView) в тот момент, когда нужно заполнить элемент списка данными. И адаптер делает это: возвращает View с уже заполненными данными.

При вызове getView() в адаптер передается convertView, это элемент списка. Если он не равен null, его можно использовать повторно. Иначе нужно его создать. Обычно это делают при помощи статической функции from() класса LayoutInflater «надувателя разметок», которому в качестве параметра передают xml, и он строит новое View в соответствии с ним.



Когда convertView уже создан, он наполняется данными. Список передает в метод getView() также position в массиве того объекта, который требуется отобразить. При помощи метода суперкласса getItem() адаптер получает нужный объект Person и «раскладывает» его на convertView.

Схематично взаимодействие выглядит следующим образом (рис. 4.14).

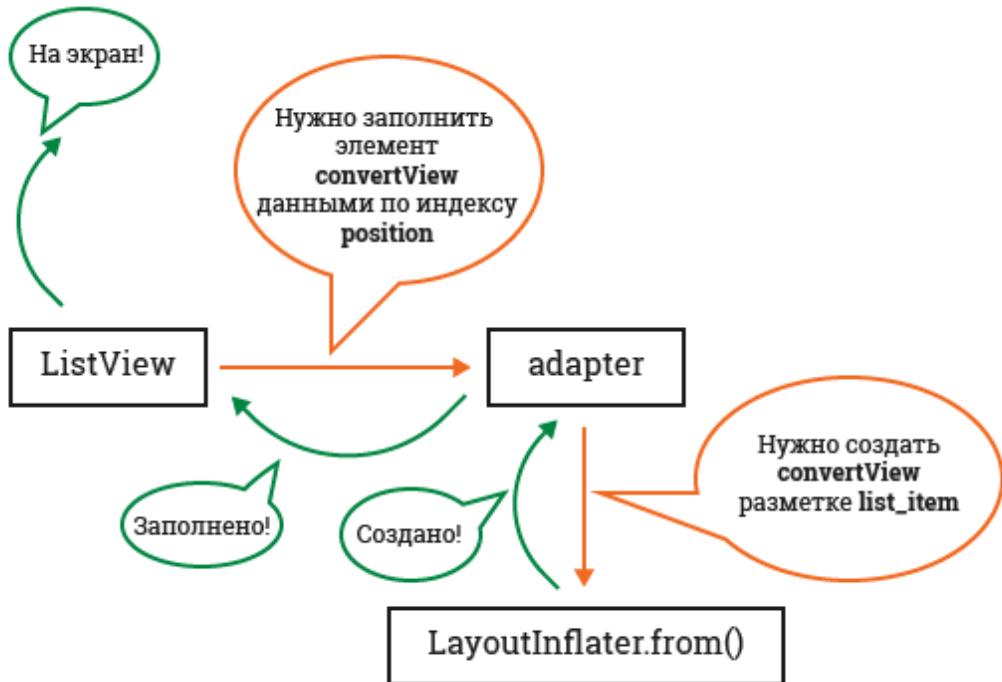


Рис. 4.14.

#### 4. Создание и настройка адаптера

Последнее, что остается сделать, это создать адаптер, передать ему данные и подключить созданный адаптер к `listView`. Для этого в `onCreate()` нужно добавить строки:

```

ListView lv = (ListView)findViewById(R.id.list);
PersonAdapter adapter = new MyMonthAdapter(this, persons);
lv.setAdapter(adapter);
    
```

Теперь приложение выглядит так (см. рис. 4.15).



Мне нравится этот месяц



Октябрь

5.2

31

Мне нравится этот месяц



Ноябрь

-0.3

30

Мне нравится этот месяц



Декабрь

-9.3

Рис. 4.15.

# Задание по Android-практикуму

Усложните приложение со списком богатейших людей планеты: каждый элемент списка должен иметь кроме имени миллиардера флаг его страны и состояние.

Ресурсы к этому заданию располагаются по адресу <https://github.com/vv73/ForbesList>

Для работы с флагами нужно скопировать содержимое архива с флагами (flags.zip) в папку *drawable*. В нем значительно больше флагов, чем будет выведено в списке, но этот сет может пригодиться в других проектах.

В качестве разметки можно взять файл `list_item.xml` (см. рис. 4.16).

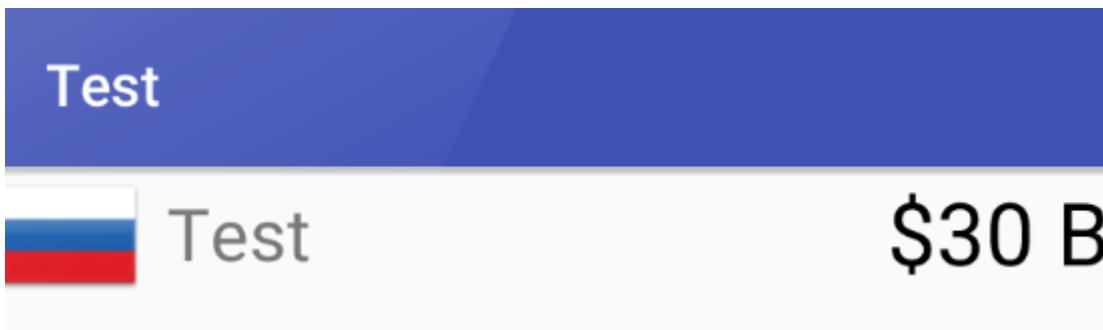


Рис. 4.16.

Для хранения данных можно использовать класс Person.

В методе `onCreate()` активности создайте два массива идентификаторов ресурсов флагов и состояний. Данные сразу можно взять из таблицы.

Третий массив имен загружается из ресурсов.

У приложения должен получиться примерно такой вид (рис. 4.17).

Bill Gates	\$86 B
Warren Buffett	\$75.6 B
Jeff Bezos	\$72.8 B
Amancio Ortega	\$71.3 B
Mark Zuckerberg	\$56 B
Carlos Slim Helu	\$54.5 B
Larry Ellison	\$52.2 B
Charles Koch	\$48.3 B
David Koch	\$48.3 B
Michael Bloomberg	\$47.5 B
Bernard Arnault	\$41.5 B
Larry Page	\$40.7 B
Sergey Brin	\$39.8 B
	\$39.8 B

Рис. 4.17.

Добавьте интерактивность.

Например, возможность открывать в браузере поисковую страничку по выбранной персоне. Для этого можно установить на список такой слушатель:

```
lv.setOnItemClickListener(new AdapterView.OnItemClickListener() {
    @Override
    public void onItemClick(AdapterView<?> parent, View view, int position, long id) {
        String name = ((Person)parent.getItemAtPosition(position)).name;
        String url = "https://www.google.ru/search?q=" + name.replace(" ", "+");
        Intent i = new Intent(Intent.ACTION_VIEW);
        i.setData(Uri.parse(url));
        startActivity(i);
    }
});
```

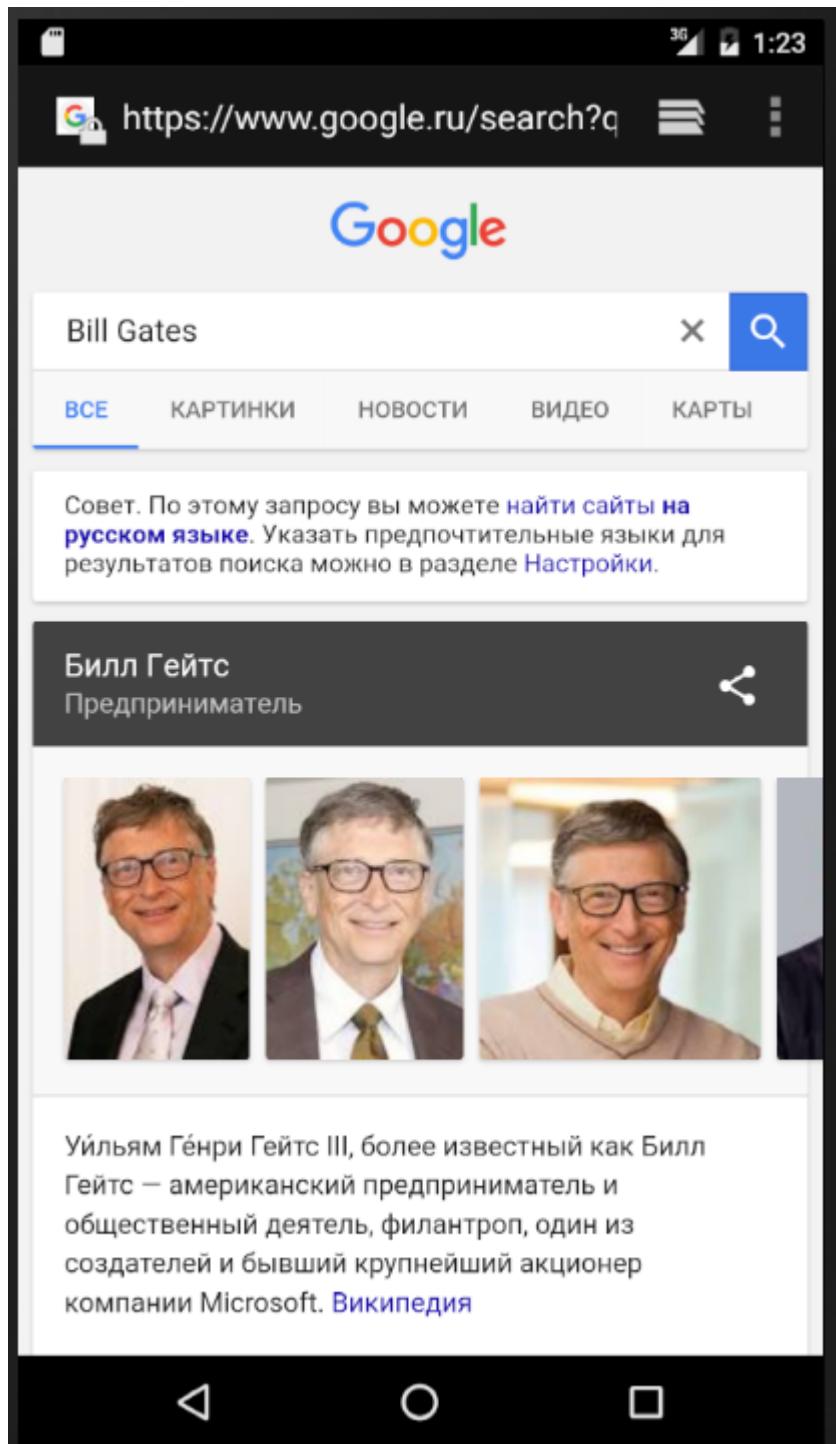


Рис. 4.18.

Для того чтобы URL стал корректным, нужно заменить в имени пробелы на знак «+».

## **4.4. СУБД. Реляционная модель**

Сайт: IT Академия SAMSUNG

Курс: MDev @ IT Академия Samsung

Книга: 4.4. СУБД. Реляционная модель

Напечатано:: Егор Беляев

Дата: Суббота, 18 Апрель 2020, 19:29

# **Оглавление**

- 4.4.1. Введение
- 4.4.2. Реляционная модель
- 4.4.3. Реляционная БД из нескольких таблиц
- 4.4.4. Классификация СУБД

## 4.4.1. Введение

С увеличением объемов информации возникла потребность в ее надежном хранении и быстром поиске. Для этих целей применяют базы данных: хранение данных определенным способом и программные средства для их эффективной обработки.

Можно дать такие определения.

**База данных (БД)** — это специальным образом организованная совокупность данных о некоторой предметной области, хранящаяся во внешней памяти компьютера.

**Система управления базами данных (СУБД)** — это программные средства для создания и обработки (добавления, изменения, поиска данных).

В современном мире БД используют во многих областях.

Когда покупатель в супермаркете оплачивает товары на кассе, то кассир через сканер получает штрих-код, и информационная система ищет товар с таким штрих-кодом в БД, получает его цену, записывает информацию, что данный товар куплен и одновременно удаляет купленный товар из списка тех, которые есть в наличии. Отдел закупок супермаркета со своей стороны видит, сколько товаров в остатке, и может проанализировать, с какой скоростью товар раскупается, далее принимает решение, сколько товара заказать на следующую неделю.

Для сдачи ЕГЭ все школьники должны зарегистрироваться в общероссийской БД. Для этого школы предоставляют информацию об учениках: ФИО, номер удостоверения пенсионного страхования, паспортные данные, регион, школа, выбранные предметы и т. д. После сдачи ЕГЭ в соответствии с определенной технологией комиссии вводят результаты в БД. Затем выпускники и приемные комиссии вузов имеют возможность увидеть эти результаты.

В каком виде можно хранить информацию из приведенных примеров? Каким образом обеспечить различные права на доступ и изменение информации для разных сторон, которые используют одну и ту же информацию? Если у нас хранится информация о миллионах товаров или школьников, как обеспечить быстрый поиск нужной информации?

Обычные текстовые файлы не могут решить описанные выше задачи по ряду причин:

- любой поиск и изменение данных не эффективны, потому что текстовые файлы требуют последовательного чтения;
- один и тот же объект (товар, ученик) можно описать по-разному, поэтому если использовать естественный язык, то информацию будет очень сложно найти и в принципе провести какой-то анализ или обработку;
- из-за того, что текстовый файл изначально не разделен на части, невозможно обеспечить доступ множества пользователей к отдельным его частям с разным уровнем доступа: кому-то для чтения, кому-то для изменения, причем одновременно.

## 4.4.2. Реляционная модель

Исторический прорыв в области хранения информации был совершен в 1970 году, когда англичанин Эдгар Кодд из компании IBM предложил реляционную модель данных. С ее появлением пришли понятия «база данных» (БД) и система управления БД (СУБД).

Понятно, что можно придумать множество различных способов, как хранить и обрабатывать данные в БД. Этот способ организации данных в БД определяется моделью данных.

Модель данных применительно к БД — это некоторые логические правила представления информации в памяти вычислительного устройства.

Позже была предложена более универсальная модель «сущность-связь», и Кодд предложил расширенную реляционную модель RM/V2. Модель «сущность-связь» очень легко перекладывается на реляционную модель, которая на данный момент остается самой распространенной и лежит в основе большинства СУБД.

### Таблицы: поля и записи

**Реляционная модель данных** — это представление совокупности данных в виде двумерных таблиц.

Рассмотрим понятия, лежащие в основе реляционной модели, на конкретном примере.

Пусть перед нами стоит задача хранить информацию по сотрудникам некоторой компании. Рассмотрим на примере сущности «Сотрудник». Для проектирования и иллюстрации модели «сущность-связь» широко используют схемы БД, которые аналогичны диаграмме классов UML для отражения ОО модели.

У каждой сущности есть ряд свойств, которые ее описывают, — *атрибуты*. Пусть в нашем примере для каждого сотрудника мы хотим хранить такие поля: «Фамилия», «Имя», «Отчество», «Отдел», «Должность», «Адрес». Тогда сущность примет вид таблицы 4.6.

**Сотрудник**

Фамилия

Имя

Отчество

Отдел

Должность

Адрес

Табл. 4.6.

В заголовке мы указали наименование сущности, внутри список атрибутов. В реляционной модели этой картинке соответствует таблица «Сотрудник» с столбцами: «Фамилия», «Имя», «Отчество», «Отдел», «Должность», «Оклад».

Если мы заполним таблицу набором значений атрибутов, например, по двум сотрудникам, то в терминах реляционной модели мы получим две записи. А таблица имеет 6 полей (см. табл. 4.7).

Адрес	Должность	Отдел	Отчество	Имя	Фамилия
г. Тверь ул. Мира д.6 кв.2 кассир	Бухгалтерия	Игоревич	Олег	Иванов	
г. Тверь ул. Весенняя д.9	кладовщик	Склад	Петрович	Пётр	Соколов

Табл. 4.7.

Реляционная модель тесно связана с реляционной алгеброй, в терминах которой набор значений атрибутов (запись) носит название кортеж.

## Ключевые поля

Каждый объект любой таблицы БД должен быть уникальным, иначе могут быть неприятности, связанные с неоднозначностью размещения одного и того же объекта несколько раз. Убирают эту неоднозначность ключевые поля, которые хранят только уникальную информацию об объекте и никогда не повторяются. Как быть в ситуации, когда в БД у людей полностью совпадают ФИО и дата рождения? Тогда ключом человека может быть серия и номер паспорта, а этого же человека в налоговой службе — ИНН, в пенсионном фонде — номер СНИЛС.

Пример с человеком описывает важность ключа, а также тот факт, что один и тот же человек может храниться в разных БД, а значит иметь разные свойства в таблицах. То есть человек один, а сущности разные. Еще примеры ключевых данных: серийный номер изделия, регистрационный номер авто, ну и, конечно же, номер на денежной банкноте, который хранится в БД казначейства и различает одинаковые купюры.

Таким образом, у каждой сущности должно быть определено специальное поле — ключ.

Ключ — это атрибут или группа атрибутов, значения которых уникальны для каждого экземпляра сущности. Ключ позволяет быстро идентифицировать каждый экземпляр сущности. Для ранее рассмотренного примера одним ключом может быть поле «Табельный номер», он уникален для каждого сотрудника и по нему легко найти конкретного сотрудника (табл. 4.8).

### Сотрудник

---

Табельный номер

---

Фамилия

Имя

Отчество

Отдел

Должность

Адрес

Табл. 4.8.

В схемах БД для выделения ключевые атрибуты отделяются от остальных чертой либо какими-то знаками (ключиками, звездочками и т. д.).

## Идеи Кодда на языке таблиц

- Каждая таблица описывает одну сущность/отношение — порядок расположения полей в таблице не имеет значения;
- все значения одного поля имеют одинаковый тип данных;
- в таблице не может быть двух полностью одинаковых записей;
- порядок записей в таблице не определен.

При работе с реляционной БД характерно:

- количество и состав полей определяет разработчик БД, пользователи же работают с записями таблицы (добавляют, удаляют, редактируют);
- любое поле должно иметь уникальное имя;
- поля имеют тип (схоже с понятием тип переменной). Как правило, в СУБД можно задать следующие типы полей: строка символов, текст — целое число — вещественное число — денежная сумма — дата, время — логическое поле (истина или ложь, да или нет). Поля могут быть обязательными для заполнения или нет;
- таблица может содержать столько записей, сколько позволяет объем памяти;

- для работы с ними используют язык запросов SQL. Программисты не работают напрямую с файлами БД, для этого используют специальные программы — системы управления базой данных (СУБД), о которых пойдет речь далее.

## 4.4.3. Реляционная БД из нескольких таблиц

До сих пор мы рассматривали примеры отдельных таблиц, но очевидно, что весь объем данных предметной области нельзя уместить в одну таблицу. Причем это связано не только с тем соображением, что это будет слишком большая таблица. Вспомним классы в ООП. Здесь в чем-то похожий подход — данные группируются в соответствии с логикой принадлежности к той или иной сущности. На практике БД — это набор таблиц, которые связаны между собой.

### Пример 4.2

Перед нами стоит задача разработать БД для ведения семейного бюджета, главная задача которой фиксировать доходы и расходы семьи, чтобы затем можно было провести их анализ.

Начнем с того, что БД должна хранить данные о всех расходах и доходах, то есть какие происходили денежные операции. И для каждой операции еще нужно знать, на что были потрачены деньги или откуда пришли (зарплата, кредит и т. д.). Рассмотрим таблицу 4.9 «Операция» со следующей структурой

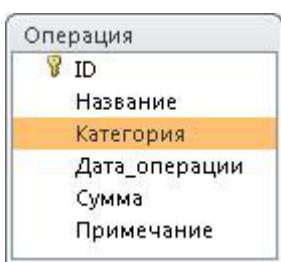


Рис. 4.9.

ID — это ключевое поле, поэтому на схеме слева от наименования нарисован ключик.

В этой структуре плохо то, что мы храним все детальные операции, но не можем провести их анализ, сгруппировать: например, посчитать, сколько за месяц мы потратили на питание, сколько на транспорт и т. д. Значит, нам нужно указание, из какой категории каждая операция.

Заполним таблицу 4.10 «Операция».

Примечание	Сумма	Дата операции	Категория	Название	ID
	120 000	01.03.2018	Остаток	Начальный баланс	1
	-3 300	06.03.2018	Продукты	Продукты	2
Поздравление на 8 марта	-1 800	08.03.2018	Разное	Разное	3
	-1 000	08.03.2018	Развлечения	Развлечения	4
	-210	09.03.2018	Продукты	Продукты	5
	50 000	10.03.2018	Зарплата	Зарплата	6

Табл. 4.10.

### Нормализация

Данные этой таблицы страдают избыточностью из-за совпадений некоторых значений поля «Категория». Причем, из-за того, что это текстовые поля, будет расходоваться много памяти. Более того, если мы захотим изменить наименование категории, то нам придется сделать это во всех записях таблицы. Убрать избыточность позволит отдельная таблица или таблицы. В нашем случае можно создать таблицу, которая хранит сущность «Категория» (см. табл. 4.11).

Приход/ Расход	Название	ID
1	Остаток	1

Приход/ Расход	Название	ID
0	Продукты	2
0	Разное	3
0	Развлечения	4
1	Зарплата	0
0	Транспорт	6
0	Медицина	7
0	Образование	8
1	Кредит	9
0	Выплата по кредиту	10

Табл. 4.11.

В таблицу мы дополнительно включили новое поле, в котором будем фиксировать, какая это категория: доход или расход. Это позволит группировать операции не только по категориям, но и еще более укрупненно: например, все расходы за месяц. Если приход, то ставим в таблице 1, если расход, то — 0.

## Связи

Кроме этого, мы должны связать таблицу операций с категориями. Для этого нужно выяснить, какая из них будет главная, а какая подчиненная. Поскольку таблица «Категория» хранит неповторяющиеся значения, она будет *главной*, а таблица «Операция» будет *подчиненной*.

Как физически связываются таблицы? Для этого подчиненной таблице добавляют еще одно поле для связи с главной. То есть таблице «Операция» нужно добавить поле ID\_категории. Затем связем ключевое поле главной таблицы с новым полем подчиненной таблицы. Новое поле подчиненной таблицы называют «внешний ключ» и оно отображает информацию, которая хранится в другой таблице, а именно в ключевом поле главной таблицы. То ключевое поле главной таблицы называется *главным*. Для отображения связи между сущностями на схемах БД используют линии (см. рис. 4.19).

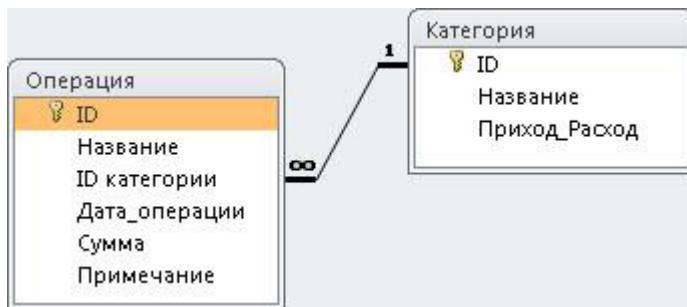


Рис. 4.19.

Наполнение таблицы 4.12 «Операция» станет следующим.

Примечания	Сумма	ID категории	Дата операции	Наименование	ID
	120000	1	01.03.2018	Начальный баланс	1
	-3 300	2	06.03.2018	Хлеб, мясо, овощи, фрукты	2
Поздравление с 8 марта	-1 800	3	08.03.2018	Цветы	3
	-1 000	4	08.03.2018	Билеты в цирк	4
	-210	1	09.03.2018	Хлеб, творог, сметана	5
	50 000	5	10.03.2018	Зарплата Сергея, февраль	6

Табл. 4.12.

## Ссылочная целостность

Что случится, если в таблице «Категория» удалить любую запись? Очевидно, что возникнет проблема в таблице «Операция», потому что для операции по заданному коду категории мы не сможем получить нужную информацию. То есть сущность «Операция» зависит от сущности «Категория». Если в таблице есть внешний ключ, то он должен обязательно совпадать с одним из значений первичного ключа в другой таблице. Это называется соблюдением **ссылочной целостности**. Современные СУБД имеют различные средства поддержки ссылочной целостности БД. Например, пользователю не разрешат удалить таблицу, если есть таблицы, которые от нее зависят.

## Типы связей

При проектировании БД и работы с ней очень важно устанавливать связи между таблицами. Не менее важно понимать какого типа эти связи. Рассмотрим основные из них.

### 1. Связь «один ко многим»

На рисунке 4.19 со схемой БД, соединяющей таблицы, отображены 1 и  $\infty$ . Это условное обозначение типа связи 1:M — «один ко многим». Это самая распространенная связь.

Связь «один ко многим» означает, что с одной записью в таблице, на стороне которой стоит «1», могут быть связаны сколько угодно записей из другой таблицы, где стоит  $\infty$ . То есть во второй таблице может быть несколько записей с одинаковым значением внешнего ключа, что мы и наблюдали.

### 2. Связь «один к одному»

Редко, но встречается связь «1:1», когда каждой записи в первой таблице соответствует 0 или 1 запись в связанной таблице. Это обычно используют для вертикального разделения большой таблицы на две части. Например, сделать часть полей таблицы доступной всем, а остальные поля вынести в другую таблицу и показывать в зависимости от прав пользователя.

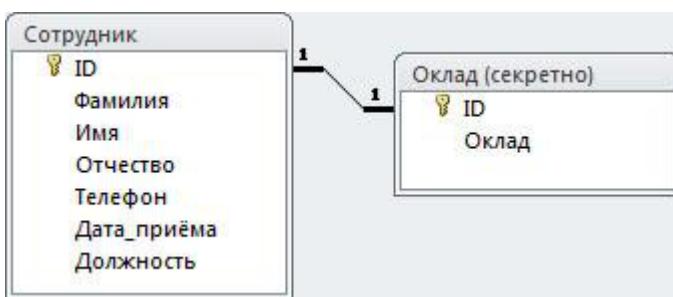


Рис. 4.20.

### 3. Связь «многие ко многим»

Наиболее сложная, но при этом часто встречающаяся связь «M:M». Например, в школе преподаватель может вести несколько предметов и в то же время предмет с таким наименованием могут преподавать несколько учителей.

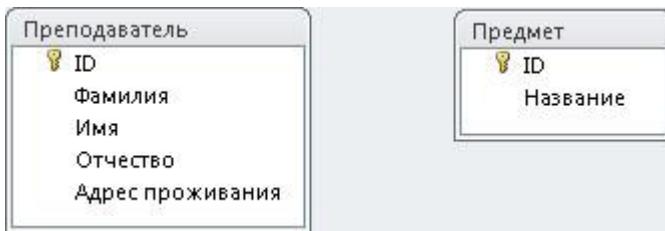


Рис. 4.21.

Значит есть сущности «Преподаватель» и «Предмет», между которыми связь «многие ко многим».

Реляционные БД не могут на физическом уровне явно реализовать связь «многие ко многим». И действительно, невозможно в плоской таблице к одной записи без ее дублирования привязать разные значения внешнего ключа.

**Появление при проектировании связи «многие ко многим» всегда говорит о том, что не достает третьей сущности, которая описывает процесс взаимодействия этих двух сущностей.**

В данном случае явно не хватает сущности «Преподавание». То есть нужно просто ввести новую сущность, которая и будет хранить все нужные соответствия ключей из таблиц «Преподаватель» и «Предмет» (см. рис. 4.22).

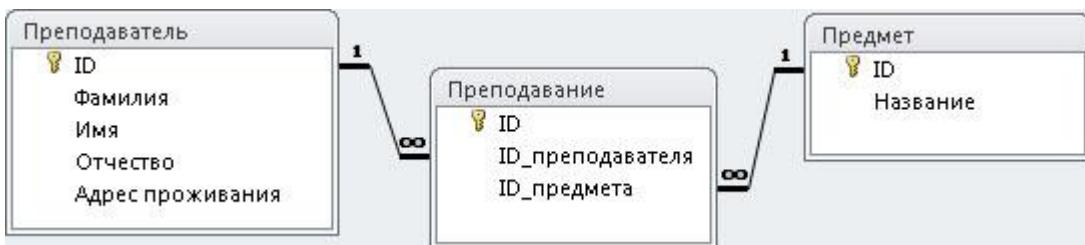


Рис. 4.22.

Пример содержимого таблиц.

Адрес проживания	Отчество	Имя	Фамилия	ID
г. Москва, ул. Орджоникидзе 3	Иванович	Иван	Иванов	1
г. Москва, ул. Рабочая 12	Петрович	Пётр	Петров	2
г. Москва, 1-й Зборовский пер. 3	Николаевна	Анна	Сидорова	3

Табл. 4.13. Преподаватель

ID	Название
1	Математика
2	Алгебра
3	Геометрия
4	География
5	Информатика
6	Информатика
7	Русский язык

Табл. 4.14. Предмет

ID	Предмета	ID	Предподавателя	ID
1				1
2				2
3				3
6				4
7				5
5				6
1				7
2				8

Табл. 4.15. Преподавание

Из таблицы 4.15 «Преподавание» легко определить, что, к примеру, преподаватель Петров ведет литературу и русский язык.

Обратим еще внимание на то, что в полях ID\_преподавание и ID\_предмета только целочисленные значения. Это существенно увеличивает производительность при поиске информации. Но главное — не приведет к избыточности и не позволит допустить синтаксическую ошибку при ручном вводе данных.

#### 4.4.4. Классификация СУБД

Рассмотрим более подробно инструменты, которые позволяют работать с БД, — системы управления базами данных (СУБД).

**Система управления базами данных СУБД ( DataBase Management System, DBMS )** — программное обеспечение, с помощью которого пользователи могут определять, создавать и поддерживать базу данных, а также получать к ней контролируемый доступ.

СУБД — это программа, которая:

- создает БД и редактирует в ней информацию (добавление, удаление и изменение информации);
  - позволяет выполнять вычисления над данными;
  - предоставляет или ограничивает доступ к БД;
  - обеспечивает одновременный доступ к данным для нескольких пользователей;
  - обеспечивает целостность БД;
  - позволяет восстановить БД в случае сбоя аппаратного или программного обеспечения.
- В истории вычислительной техники можно проследить развитие двух основных областей ее использования. Первая область — применение вычислительной техники для выполнения численных расчетов, которые слишком долго или вообще невозможно производить вручную. Характерной особенностью данной области применения вычислительной техники является наличие сложных алгоритмов обработки, которые применяются к простым по структуре данным, объем которых, как правило, сравнительно невелик.

Вторая область — это использование средств вычислительной техники в автоматических или автоматизированных информационных системах.

Информационная система представляет собой программно-аппаратный комплекс, обеспечивающий выполнение следующих функций:

- надежное хранение информации в памяти компьютера;
- выполнение специфических для данного приложения преобразований информации и вычислений;
- предоставление пользователям удобного и легко осваиваемого интерфейса.

Обычно такие системы имеют дело с большими объемами информации, имеющей достаточно сложную структуру, поэтому используют БД. Классическими примерами информационных систем являются банковские системы, автоматизированные системы управления предприятиями, системы резервирования авиационных или железнодорожных билетов, мест в гостиницах и т. д.

Не стоит путать понятия «информационная система» и СУБД. СУБД и БД являются лишь частями информационной системы. Помимо них в ИС всегда присутствует прикладное программное обеспечение (ПО), которое реализует удобный интерфейс пользователя для работы с данными: формы ввода и поиска, отчеты и т. д.

Что это значит? Например, в ИС по продаже билетов можно огромным количеством способов выдавать информацию из БД, но в системе реализуют только те формы и отчеты, которые необходимы, например, отчет, который показывает, сколько и какие свободные места есть в поезде на заданную дату, каким образом можно добраться из пункта А в пункт Б в определенные даты и т. д.

Первые БД основывались на иерархической и сетевой моделях. В 1965 году на конференции CODASYL (Conference on Data Systems Languages) была сформирована рабочая группа Data Base Task Group (DBTG), которая занялась определением спецификаций среды, которая

допускала бы разработку баз данных и управление данными. В 1971 году DBTG выделила две группы языков.

- Язык определения данных (Data Definition Language, DDL), который позволит описать структуру БД.
- Язык манипулирования данными (Data Manipulation Language, DML), предназначенный для управления данными в БД.

Несмотря на то что этот отчет официально не был одобрен Национальным Институтом Стандартизации США (American National Standards Institute — ANSI), большое количество систем было разработано в полном соответствии с этими предложениями группы DBTG. Теперь они называются CODASYL-системами, или DBTG-системами. CODASYL-системы и системы на основе иерархических подходов представляют собой СУБД первого поколения.

В 1970 году Э. Ф. Кодд, работавший в корпорации IBM, опубликовал статью о реляционной модели данных, позволяющей устраниТЬ недостатки прежних моделей. И это дало начало общеизвестным результатам:

- появился структурированный язык запросов SQL, который стал стандартным языком любых реляционных СУБД;
- в 80-х годах были созданы различные коммерческие реляционные СУБД, например, DB2 или SQL/DS корпорации IBM, Oracle корпорации Oracle, др.



Существует несколько сотен различных реляционных СУБД для мейнфреймов и персональных ЭВМ. В качестве примера многопользовательских СУБД может служить система CA-OpenIngres фирмы Computer Associates и система Informix фирмы Informix Software, Inc. Примерами реляционных СУБД для персональных компьютеров являются Access фирмы Microsoft, Paradox и Visual dBase фирмы Borland, а также R-Base фирмы MicroRIM. Реляционные СУБД относятся к СУБД второго поколения.

Однако реляционная модель также обладает некоторыми недостатками — в частности, ограниченными возможностями моделирования. Для решения этой проблемы был выполнен большой объем исследовательской работы. В 1976 году Чен предложил модель «сущность-связь» (Entity-Relationship model — ER-модель), которая стала основой методологии концептуального проектирования баз данных и методологии логического проектирования реляционных баз данных. В 1979 году Кодд сделал попытку устраниТЬ недостатки собственной основополагающей работы и опубликовал расширенную версию реляционной модели — RM/T (1979), затем еще одну версию — RM/V2 (1990). Попытки создания модели данных, позволяющей более точно описывать реальный мир, называют семантическим моделированием данных (semantic data modeling). В ответ на все возрастающую сложность приложений баз данных появились две новых разновидности: объектно-ориентированные СУБД, или ОО СУБД (Object-Oriented DBMS — OODBMS), и объектно-реляционные СУБД, или ОР СУБД (Object-Relational DBMS — ORDBMS). Попытки реализации подобных моделей представляют собой СУБД третьего поколения.

Существует множество способов классификации СУБД, рассмотрим наиболее значимые.

## **Классификация по месту размещения СУБД**

СУБД делят на локальные, удаленные и распределенные.

### **Локальные**

Это самый простой случай, когда прикладное ПО, СУБД и БД находятся на одном устройстве пользователя (например, компьютер, планшет или смартфон). Такой способ реализации хорошо подходит для небольших ИС, где предполагается один пользователь. В ОС Android для создания локальных БД уже встроена СУБД SQLite.

Для ОС Windows удобнее использовать СУБД, которые дополнительно предоставляют готовые инструменты для создания прикладной части ИС — средства конструирования интерфейса пользователя (Microsoft Access, OpenOffice Base). Преимущества локальных БД: независимость от работы компьютерных сетей.

Недостатки: невозможность работать нескольким пользователям одновременно.

## **Удаленные и распределенные**

Их реализуют, когда предполагается работа множества пользователей с одной БД. Для этих систем зачастую характерен большой объем хранимых данных, и они строятся на клиент-серверной архитектуре.

Отличие между удаленной и распределенной базами данных в том, что для первой характерно размещение СУБД и БД на одном сервере. Распределенная же БД размещается на нескольких серверах.

## **Классификация по способу доступа к БД**

### **Файл-серверные**

В файл-серверных СУБД файлы данных располагаются централизованно на файл-сервере. СУБД располагается на каждом клиентском компьютере (рабочей станции). Доступ СУБД к данным осуществляется через локальную сеть. Синхронизация чтений и обновлений осуществляется посредством файловых блокировок. Преимуществом этой архитектуры является низкая нагрузка на процессор файлового сервера. Недостатки: потенциально высокая загрузка локальной сети; затрудненность или невозможность централизованного управления; затрудненность или невозможность обеспечения таких важных характеристик, как высокая надежность, высокая доступность и безопасность. Применяются чаще всего в локальных приложениях, которые используют функции управления БД; в системах с низкой интенсивностью обработки данных и низкими пиковыми нагрузками на БД.

На данный момент файл-серверная технология считается устаревшей, а ее использование в крупных информационных системах — недостатком.

Примеры: Microsoft Access, Paradox, dBase, Visual FoxPro.

### **Клиент-серверные**

Главной отличительной чертой построения клиент-серверной СУБД в том, что прикладное ПО (клиент) физически отделено от СУБД и БД. При этом к БД клиент может обращаться только через СУБД.

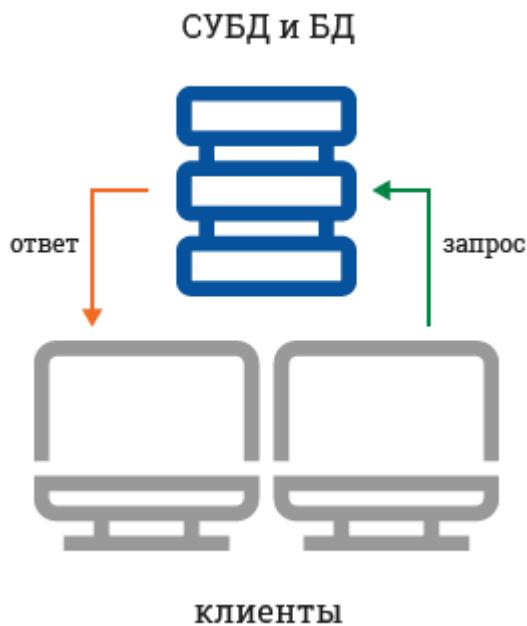


Рис. 4.23.

Клиентское приложение на устройстве пользователя по сети отправляет команды (запросы) к СУБД на выполнение различных операций. Команды клиента к СУБД формируются на специальном языке запросов SQL (Structured Query Language — язык структурных запросов).

Что делает сервер:

- ожидает запросы от клиентов по сети;
- выставляет запросы в очередь на выполнение;
- выполняет запросы;
- отправляет ответы обратно клиенту.

В частном случае серверная и клиентская части могут быть установлены на одном компьютере. Недостаток клиент-серверных СУБД состоит в повышенных требованиях к серверу (высокая стоимость). Достоинства: потенциально более низкая загрузка локальной сети; удобство централизованного управления; удобство обеспечения таких важных характеристик, как высокая надежность, высокая доступность и безопасность.

Примеры: Oracle, Firebird, Interbase, IBM DB2, MS SQL Server, Sybase Adaptive Server Enterprise, PostgreSQL, MySQL, Caché.

## Встраиваемые

Встраиваемая СУБД — СУБД, которая может поставляться как составная часть некоторого программного продукта, не требуя процедуры самостоятельной установки. Встраиваемая СУБД предназначена для локального хранения данных своего приложения и не рассчитана на коллективное использование в сети. Физически встраиваемая СУБД чаще всего реализована в виде подключаемой библиотеки. Доступ к данным со стороны приложения может происходить через SQL либо через специальные программные интерфейсы.

Примеры: SQLite (она часто используется при программировании под Android), BerkeleyDB, Firebird Embedded, Microsoft SQL Server Compact.

## **4.5. СУБД SQLite. Основы языка SQL**

Сайт: IT Академия SAMSUNG  
Курс: MDev @ IT Академия Samsung  
Книга: 4.5. СУБД SQLite. Основы языка SQL  
Напечатано:: Егор Беляев  
Дата: Суббота, 18 Апрель 2020, 19:29

# **Оглавление**

- 4.5.1. СУБД SQLite
- 4.5.2. Введение в SQL
- 4.5.3. Создание таблиц — оператор CREATE TABLE
- 4.5.4. Добавление записей в таблицу — INSERT
- 4.5.5. Выборка данных — инструкция SELECT
- 4.5.6. Изменение таблицы — UPDATE
- 4.5.7. Удаление записей — DELETE
- 4.5.8. Агрегированные запросы
- 4.5.9. Работа с базой данных SQLite на Android-устройстве
- 4.5.10. Мини-проект (продолжение 4.1)

## 4.5.1. СУБД SQLite

*SQLite* — компактная локальная реляционная СУБД. SQLite не использует парадигму клиент-сервер, то есть движок SQLite не является отдельно работающим процессом, что особенно важно при работе на мобильных устройствах. Такой подход уменьшает накладные расходы, время отклика и упрощает программу. SQLite хранит всю базу данных (включая определения, таблицы, индексы и данные) в единственном стандартном файле на том устройстве, на котором исполняется программа.

Несколько процессов или потоков могут одновременно без каких-либо проблем читать данные из одной базы. Запись в базу можно осуществить только в том случае, если никаких других запросов в данный момент не обслуживается; в противном случае попытка записи заканчивается неудачей, и в программу возвращается код ошибки. Благодаря архитектуре движка возможно использовать SQLite как на встраиваемых системах, так и на выделенных машинах с гигабайтными массивами данных.

По ряду оценок функциональность SQLite находится где-то между MySQL и PostgreSQL. Однако на практике SQLite нередко оказывается в 2–3 раза (и даже больше) быстрее. Такое возможно благодаря высокоупорядоченной внутренней архитектуре и устранению необходимости в соединениях типа «сервер-клиент» и «клиент-сервер».



На практике приложения под Android не используют СУБД SQLite для хранения данных большого объема и со сложной структурой. Для таких случаев наиболее распространено применение клиент-серверных СУБД.

Далее мы приступим к изучению SQL — языка запросов для реляционных БД. И по ходу изложения будем отмечать особенности, характерные для СУБД SQLite.

## 4.5.2. Введение в SQL

Одним из языков, появившихся в результате разработки реляционной модели данных, является язык SQL (Structured Query Language), который в настоящее время получил очень широкое распространение и фактически стал стандартом реляционных баз данных.

Язык SQL достаточно компактен, содержит небольшое количество команд. Мы рассмотрим их подмножество, необходимое в обычной практической работе.

### Запись SQL-операторов

Идентификаторы языка SQL предназначены для обозначения объектов в базе данных: имен таблиц, столбцов и других объектов базы данных. В соответствии со стандартом SQL идентификатор:

- по умолчанию может содержать строчные и прописные буквы латинского алфавита (A-Z, a-z), цифры (0–9), символ подчеркивания (\_);
- может иметь длину до 128 символов;
- должен начинаться с буквы;
- не может содержать пробелы.

Большинство компонентов языка не чувствительны к регистру. Отдельные SQL-операторы и их последовательности будут иметь произвольное количество отступов и переносятся по строкам, что позволяет создавать более читаемый вид при использовании выравнивания.

### Комментарии в SQL

Комментарии в SQLite обозначаются двумя последовательными минусами (-), которые комментируют остаток строки. Многострочные комментарии, как в языке Java, обозначаются парами символов /\* \*/.

## 4.5.3. Создание таблиц — оператор CREATE TABLE

Таблицы базы данных создаются с помощью оператора CREATE TABLE. Эта команда создает пустую таблицу, то есть таблицу, не имеющую строк. Значения в эту таблицу добавляются с помощью оператора INSERT. Оператор CREATE TABLE определяет имя таблицы и множество поименованных столбцов в указанном порядке. Для каждого столбца могут быть определены тип и размер. Каждая создаваемая таблица должна иметь по крайней мере один столбец.

Базовый синтаксис оператора создания таблицы имеет следующий вид:

```
CREATE TABLE <имя_таблицы>
(<имя_столбца> <тип_данных>
 [NULL | NOT NULL ] [, ...n])
```

Здесь и далее квадратные скобки используются для необязательных частей конструкции, то есть сами квадратные скобки не пишутся, а то, что в них, можно опускать. Угловые скобки тоже не пишутся, а указывают на тип данных, вместо которых нужно подставить реальные значения, например, имя таблицы.

Конструкция NOT NULL обозначает, что поле должно быть обязательно заполнено.

Пример 4.3

Создать таблицу для хранения данных о товарах. Необходимо учесть такие сведения, как название, тип товара и его цену.

```
CREATE TABLE Product
(id INTEGER PRIMARY KEY AUTOINCREMENT,
Name TEXT NOT NULL UNIQUE,
Price REAL)
```

Обратите внимание на поле id. Это поле встречается в таблицах SQL и является первичным ключом, однозначно определяет объект, в нашем случае — товар.

Первый столбец обозначен, как *primary key* (первичный ключ), это понятие мы уже разобрали в предыдущей теме. Слово *autoincrement* указывает, что база данных будет автоматически увеличивать значение ключа при добавлении каждой записи, что и обеспечивает его уникальность (поле — счетчик).

### Создание таблицы в SQLite

В SQLite существует договоренность первый столбец всегда называть \_id, это не жесткое требование SQLite, однако может понадобиться при использовании контент-провайдера в Android.

Стоит также иметь в виду, что в SQLite, в отличие от многих других СУБД, типы данных столбцов являются лишь подсказкой, то есть не вызовет никаких нареканий попытка записать строку в столбец, предназначенный для хранения целых чисел или наоборот. Этот факт можно рассматривать как особенность базы данных, а не как ошибку, на это обращают внимание авторы SQLite.

Возможные типы полей:

- TEXT — строки или символы в кодировке UTF-8, UTF-16BE или UTF-16LE;

- INTEGER — целое число;
- REAL — дробное число;
- BLOB — бинарные данные;
- TIMESTAMP — метка времени;
- NULL — пустое значение.

В SQLite отсутствует тип, работающий с датами. Можно использовать строковые значения, например, как 2015-02-16 (16 февраля 2015 года). Для даты со временем рекомендуется использовать формат 2015-02-16T07:58. В таких случаях можно использовать некоторые функции SQLite для добавления дней, установки начала месяца и т. д. Учтите, что SQLite не поддерживает часовые пояса. Также не поддерживается тип boolean. Используйте числа 0 для false и 1 для true. Не используйте тип BLOB для хранения данных (картинки) в Android. Лучше хранить в базе путь к изображениям, а сами изображения хранить в файловой системе.

Как было отмечено выше, SQLite поддерживает типы TEXT (аналог String в Java), INTEGER (аналог long в Java) и REAL (аналог double в Java). Остальные типы следует конвертировать, прежде чем сохранять в базе данных. SQLite сама по себе не проверяет типы данных, поэтому вы можете записать целое число в колонку, предназначенную для строк и наоборот.

### Упражнение 4.10.1

Выполним Пример 1 в SQLite через консольную утилиту sqlite. Все дальнейшие примеры также будем запускать через консольную утилиту.

## 4.5.4. Добавление записей в таблицу — INSERT

Оператор INSERT применяется для добавления записей в таблицу. Формат оператора:

```
INSERT INTO <имя_таблицы>
[(имя_столбца [,...n])]
{VALUES (значение[,...n])|
<SELECT_оператор>}
```

Первая форма оператора INSERT с параметром VALUES предназначена для вставки единственной строки в указанную таблицу. Список столбцов указывает столбцы, которым будут присвоены значения в добавляемых записях. Список может быть опущен, тогда подразумеваются все столбцы таблицы (кроме объявленных как счетчик), причем в определенном порядке, установленном при создании таблицы.

Если в операторе INSERT указывается конкретный список имен полей, то любые пропущенные в нем столбцы должны быть объявлены при создании таблицы как допускающие значение NULL, за исключением тех случаев, когда при описании столбца использовался параметр DEFAULT. Список значений должен следующим образом соответствовать списку столбцов:

- количество элементов в обоих списках должно быть одинаковым;
- должно существовать прямое соответствие между позицией одного и того же элемента в обоих списках, поэтому первый элемент списка значений должен относиться к первому столбцу в списке столбцов, второй — ко второму столбцу и т. д.;
- типы данных элементов в списке значений должны быть совместимы с типами данных соответствующих столбцов таблицы.

### Пример 4.4

Добавить в таблицу Product новую запись.

```
INSERT INTO Product (Name, Price) VALUES ("Ливерная колбаса", 50.70)
```

Если столбцы таблицы Product указаны в полном составе и в том порядке, в котором они перечислены при создании, оператор можно упростить:

```
INSERT INTO Товар VALUES ("Ливерная колбаса", 50.70)
```

## 4.5.5. Выборка данных — инструкция SELECT

### Инструкция SELECT

Оператор SELECT — один из наиболее важных и самых распространенных операторов SQL. Он позволяет производить выборки данных из таблиц и преобразовывать к нужному виду полученные результаты. Оператор SELECT имеет следующий формат:

```
SELECT [ALL | DISTINCT] {*}|[имя_столбца  
[AS      новое_имя]]} [,...n]  
FROM  имя_таблицы [[AS] псевдоним] [,...n]  
[WHERE <условие_поиска>]  
[GROUP BY имя_столбца [,...n]]  
[HAVING <критерии выбора групп>]  
[ORDER BY имя_столбца [,...n]]
```

Оператор SELECT определяет поля (столбцы), которые будут входить в результат выполнения запроса. В списке они разделяются запятыми и приводятся в такой очередности, в какой должны быть представлены в результате запроса. Символом \* можно выбрать все поля, а вместо имени поля применить выражение из нескольких имен.

### Предложение FROM

Предложение FROM задает имена таблиц и представлений, которые содержат поля, перечисленные в операторе SELECT. Необязательный параметр псевдонима — это сокращение, устанавливаемое для имени таблицы.

Обработка элементов оператора SELECT выполняется в следующей последовательности:

1. FROM — определяются имена используемых таблиц;
2. WHERE — выполняется фильтрация строк объекта в соответствии с заданными условиями;
3. GROUP BY — образуются группы строк, имеющих одно и то же значение в указанном столбце;
4. HAVING — фильтруются группы строк объекта в соответствии с указанным условием;
5. SELECT — устанавливается, какие столбцы должны присутствовать в выходных данных;
6. ORDER BY — определяется упорядоченность результатов выполнения операторов.

Порядок предложений и фраз в операторе SELECT не может быть изменен. Только два предложения SELECT и FROM являются обязательными, все остальные могут быть опущены. SELECT — закрытая операция: результат запроса к таблице представляет собой другую таблицу.

Пример 4.5. Составить список всех данных о всех продуктах.

```
SELECT * FROM Product
```

Пример 4.6. Составить список всех продуктов.

```
SELECT Name FROM Product
```

Результат выполнения запроса может содержать дублирующиеся значения.

### Предикат DISTINCT

Для исключения дублирования используют предикат DISTINCT. Необходимо помнить, что использование DISTINCT может резко замедлить выполнение запросов. Откорректированный пример выглядит следующим образом:

```
SELECT DISTINCT Name FROM Product
```

## Предложение WHERE

С помощью WHERE-параметра пользователь определяет, какие строки появятся в результате запроса. За ключевым словом WHERE следует перечень условий поиска, определяющих те строки, которые должны быть выбраны при выполнении запроса.

Существует пять основных типов условий поиска (или предикатов).

1. Сравнение: сравниваются результаты вычисления одного выражения с результатами вычисления другого.
2. Диапазон: проверяется, попадает ли результат вычисления выражения в заданный диапазон значений.
3. Принадлежность множеству: проверяется, принадлежит ли результат вычислений выражения заданному множеству значений.
4. Соответствие шаблону: проверяется, отвечает ли некоторое строковое значение заданному шаблону.
5. Значение NULL: проверяется, содержит ли данный столбец определитель NULL (неизвестное значение).

## Сравнение

В языке SQL можно использовать следующие операторы сравнения: = — равенство; < — меньше; > — больше; <= — меньше или равно; >= — больше или равно; <> — не равно.

Пример 4.7. Показать все товары, стоимостью более 100.

```
SELECT * FROM Product WHERE Price > 100
```

В выражениях можно использовать логические операторы AND, OR или NOT, а также скобки.

Пример 4.8. Вывести список товаров, цена которых больше 100 и меньше или равна 200.

```
SELECT  
    FROM Product  
WHERE Price > 100 AND Price <= 200
```

## Соответствие шаблону

С помощью оператора LIKE можно выполнять сравнение выражения с заданным шаблоном, в котором допускается использование символов-заменителей:

- символ % — вместо этого символа может быть подставлено любое количество произвольных символов;
- символ \_ заменяет один символ строки;
- [] — вместо символа строки будет подставлен один из возможных символов, указанный в этих ограничителях;
- [^] — вместо соответствующего символа строки будут подставлены все символы, кроме указанных в ограничителях.

Пример 4.9. Вывести список колбас и сыров.

```
SELECT Name, Price  
FROM Product  
WHERE Name LIKE "Сыр%" OR  
Name LIKE "Колбаса%"
```

Обратите внимание. Вместо союза «и», который мы использовали в русском языке, нужно использовать «OR», то есть «или».

###Диапазон

Оператор BETWEEN используется для поиска значения внутри некоторого интервала, определяемого своими минимальным и максимальным значениями. При этом указанные значения включаются в условие поиска.

\*\*Пример 8.\*\* Вывести список товаров, цена которых лежит в диапазоне от 100 до 150

```
```sql  
SELECT Name, Price  
FROM Product  
WHERE Price BETWEEN 100 And 150
```

Пример 4.10. Вывести список товаров, цена которых НЕ лежит в диапазоне от 100 до 150.

```
SELECT Name, Price  
FROM Product  
WHERE Price NOT BETWEEN 100 AND 150
```

## Принадлежность множеству

Оператор IN используется для сравнения некоторого значения со списком заданных значений, при этом проверяется, соответствует ли результат вычисления выражения одному из значений в предоставленном списке. При помощи оператора IN может быть достигнут тот же результат, что и в случае применения оператора OR, однако оператор IN выполняется быстрее.

Пример 4.11.

```
SELECT Name, Price  
FROM Product  
WHERE Name IN ("Сырок Глазированный", "Сыр домашний")
```

## Значение NULL

Оператор IS NULL используется для сравнения текущего значения со значением NULL — специальным значением, указывающим на отсутствие любого значения.

Пример 4.12. Вывести товары с неуказанными ценами.

```
SELECT Name FROM Product  
WHERE Price IS NULL
```

IS NOT NULL используется для проверки присутствия значения в поле.

## Предложение ORDER BY

В общем случае строки в результирующей таблице SQL-запроса никак не упорядочены. Однако их можно требуемым образом отсортировать, для чего в оператор SELECT помещается фраза ORDER BY. Сортировка может выполняться по нескольким полям, в этом случае они перечисляются за ключевым словом ORDER BY через запятую.

Пример 4.13. Вывести список продуктов в алфавитном порядке.

```
SELECT Name  
      FROM Product  
     ORDER BY Name
```

Пример 4.14. Вывести список продуктов от самого дорогого до самого дешевого.

```
```sql SELECT Name, Price FROM Product ORDER BY Price DESC ```
```

## 4.5.6. Изменение таблицы — UPDATE

Оператор UPDATE применяется для изменения значений в группе записей или в одной записи указанной таблицы. Формат оператора:

```
UPDATE имя_таблицы SET имя_столбца=<выражение>[,...n]
[WHERE <условие_отбора>]
```

Пример 4.15. Установить нулевую цену для товаров с неуказанной ценой.

```
UPDATE Product SET Price=0 WHERE Price IS NULL
```

Пример 4.16. Увеличить цену дешевых товаров на 25%.

```
UPDATE Product SET Price=Price*1.25
WHERE Price < 10
```

## 4.5.7. Удаление записей — DELETE

Оператор DELETE предназначен для удаления группы записей из таблицы. Формат оператора:

```
DELETE  
FROM <имя_таблицы>[ WHERE <условие_отбора>]
```

Пример 4.17. Удалить все записи о товарах.

```
DELETE FROM Product
```

В документации по SQLite указано, что не поддерживаются конструкции SQL, которые позволяют удалить или изменить существующий столбец в таблице (например, его имя или тип). Но можно прибегнуть к «хитрости»: создать другую таблицу, скопировав в нее нужные данные из старой:

```
-- создаем временную таблицу  
CREATE TEMPORARY TABLE Temper_backup(name,temp);  
-- копируем данные из таблицы Temper во временную таблицу Temper_backup  
INSERT INTO Temper_backup SELECT name,temp FROM Temper;  
-- удаляем таблицу Temper  
DROP TABLE Temper;  
-- создаем таблицу Temper  
CREATE TABLE Temper(name,NEW);  
-- вставляем данные из таблицы Temper_backup в таблицу Temper  
INSERT INTO Temper SELECT name,temp FROM Temper_backup;  
-- удаляем таблицу Temper_backup  
DROP TABLE Temper_backup;
```

## 4.5.8. Агрегированные запросы

С помощью итоговых (агрегатных) функций запроса можно получить статистические сведения о множестве отобранных значений выходного набора.

Пользователю доступны следующие основные итоговые функции:

- Count (выражение) — определяет количество записей в выходном наборе SQL-запроса;
- Min/Max (выражение) — определяет наименьшее и наибольшее из множества значений в некотором поле запроса;
- Avg (выражение) — эта функция позволяет рассчитать среднее значение множества значений, хранящихся в определенном поле отобранных запросом записей. Оно является арифметическим средним значением, то есть суммой значений, деленной на их количество;
- Sum (выражение) — вычисляет сумму множества значений, содержащихся в определенном поле отобранных запросом записей.

Все эти функции оперируют со значениями в единственном столбце таблицы или с арифметическим выражением и возвращают единственное значение. Функции COUNT, MIN и MAX применимы как к числовым, так и к нечисловым полям, тогда как функции SUM и AVG могут использоваться только в случае числовых полей. При вычислении результатов любых функций сначала исключаются все пустые значения, после чего требуемая операция применяется только к оставшимся конкретным значениям столбца.

Вариант COUNT(\*) — особый случай использования функции COUNT, его назначение состоит в подсчете всех строк в результирующей таблице, независимо от того, содержатся там пустые, дублирующиеся или любые другие значения.

Если до применения обобщающей функции необходимо исключить дублирующиеся значения, следует перед именем столбца в определении функции поместить ключевое слово DISTINCT. Оно не имеет смысла для функций MIN и MAX, однако его использование может повлиять на результаты выполнения функций SUM и AVG, поэтому необходимо заранее обдумать, должно ли оно присутствовать в каждом конкретном случае. Кроме того, ключевое слово DISTINCT может быть указано в любом запросе не более одного раза.

Агрегатные функции используются в запросах SELECT.

Пример 4.18. Вывести первый товар по алфавиту.

```
SELECT MIN(Name)
      FROM Product
```

Пример 4.19. Определить количество товаров.

```
SELECT COUNT(*)
      FROM Product
```

## 4.5.9. Работа с базой данных SQLite на Android-устройстве

Когда ваше приложение создает базу данных, она сохраняется в каталоге DATA/data/имя\_пакета/databases/имя\_базы.db. Например, с помощью контент-провайдера несколько приложений могут использовать одну и ту же базу данных.

Приведенный метод возвращает путь к каталогу DATA.

```
Environment.getDataDirectory()
```

Основными пакетами для работы с базой данных являются android.database и android.database.sqlite.

Для создания и обновления базы данных в Android предусмотрен класс SQLiteOpenHelper. При разработке приложения, работающего с базами данных, необходимо создать класс-наследник от SQLiteOpenHelper, в котором обязательно реализовать два абстрактных метода:

- onCreate(SQLiteDatabase db) — вызывается один раз при создании БД;
- onUpgrade (SQLiteDatabase db, int oldVersion, int newVersion) — вызывается, когда необходимо обновить БД (в данном случае под обновлением имеется в виду не обновление записей, а обновление структуры базы данных).

По желанию можно реализовать метод:

- onOpen() — вызывается при открытии базы данных.

```
// Класс для создания БД
private class OpenHelper extends SQLiteOpenHelper {

    OpenHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        String query = "CREATE TABLE " + TABLE_NAME + " (" +
                COLUMN_ID + " INTEGER PRIMARY KEY AUTOINCREMENT, " +
                COLUMN_DATE + " LONG, " +
                COLUMN_TITLE + " TEXT, " +
                COLUMN_ICON + " INTEGER); ";
        db.execSQL(query);
    }

    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        db.execSQL("DROP TABLE IF EXISTS " + TABLE_NAME);
        onCreate(db);
    }
}
```

В реальном приложении изменение структуры базы данных и ее таблиц, конечно, должно происходить без потери пользовательских данных.

В этом же классе принято объявлять открытые строковые константы для названия таблиц и полей создаваемой базы данных, которые клиенты могут использовать для определения столбцов при выполнении запросов к базе данных.

```

// Данные базы данных и таблиц
private static final String DATABASE_NAME = "itschool.db";
private static final int DATABASE_VERSION = 1;
private static final String TABLE_NAME = "MyData";

// Название столбцов
private static final String COLUMN_ID = "_id";
private static final String COLUMN_DATE = "Date";
private static final String COLUMN_TITLE = "Title";

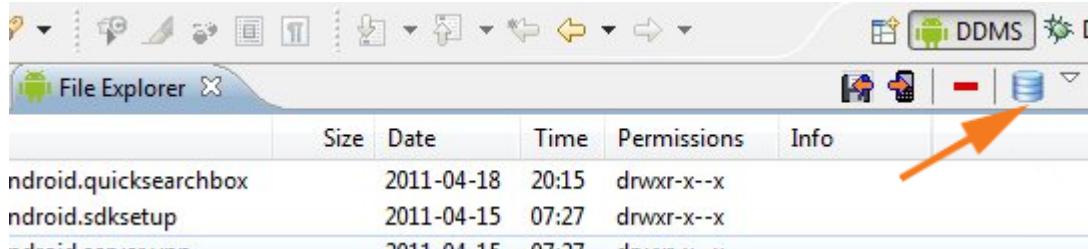
// Номера столбцов
private static final int NUM_COLUMN_ID = 0;
private static final int NUM_COLUMN_DATE = 1;
private static final int NUM_COLUMN_TITLE = 2;

```

Для приложения можно создать несколько БД, все они будут доступны из любого класса программы, но недоступны для других программ. Класс, который предоставляет методы для добавления, обновления, удаления и выборки данных из БД носит имя `SQLiteDatabase`. Он работает с базой данных SQLite напрямую и имеет свои методы для открытия, запроса, обновления данных и закрытия базы, такие как `insert()`, `update()`, `delete()` соответственно.



Существует плагин `SQLiteManager` для `Eclipse`, позволяющий видеть данные в базе. Для работы может потребоваться скачать библиотеку `com.questoid.sqlitemanager_1.0.0.jar` и скопировать ее в папку `Eclipse/dropins`. После этого перезагрузите `Eclipse` и откройте перспективу `DDMS`. Плагин будет находиться во вкладке `File Explorer`.



Щелкнув по значку, можно увидеть таблицы. Вкладка `Browse Data` позволяет смотреть непосредственно сами данные.

Основные этапы при работе с базой данных следующие:

- создание и открытие базы данных;
- создание таблицы;
- создание Insert-интерфейса (используется для вставки данных);
- создание Query-интерфейса (используется для выполнения запроса, обычно для выборки данных);
- закрытие базы данных.

## 4.5.10. Мини-проект (продолжение 4.1)

Закончим разработку нашего приложения «Чемпионат России по футболу/хоккею», начатого в предыдущей теме.

Нами спроектирована следующая структура таблицы (табл. 4.16), в которой мы будем хранить результаты матчей.

Столбец	Тип данных	Ограничения
id	INTEGER	PRIMARY KEY AUTOINCREMENT
TeamHome	TEXT	
TeamGuest	TEXT	
GoalsHome	INT	
GoalsGuest	INT	

Табл. 4.16.

Вы можете использовать свой вариант БД, но тогда это будет необходимо учесть в коде.

Файл разметки /res/layout/activity\_main.xml

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

    <ListView
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:id="@+id/list"
        />

</LinearLayout>
```

Файл разметки /res/layout/activity\_add.xml



```
        android:text="@string/teamhomefield" />

    <EditText
        android:id="@+id/TeamHome"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:inputType="text" />

</LinearLayout>
<LinearLayout
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:orientation="vertical" >

    <TextView
        style="@style/TextStyle"
        android:layout_width="120dp"
        android:layout_height="wrap_content"
        android:text="@string/teamguestfield" />

    <EditText
        android:id="@+id/TeamGuest"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:inputType="text" >

        <requestFocus />
    </EditText>

</LinearLayout>
</LinearLayout>
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal" >
        <LinearLayout
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:orientation="vertical" >

            <TextView
                style="@style/TextStyle"
                android:layout_width="123dp"
                android:layout_height="wrap_content"
                android:text="@string/goalshomefield" />

            <EditText
                android:id="@+id/GoalsHome"
                android:layout_width="fill_parent"
                android:layout_height="wrap_content"
                android:inputType="text" />

</LinearLayout>
```

```
<LinearLayout
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:orientation="vertical" >

    <TextView
        style="@style/TextStyle"
        android:layout_width="120dp"
        android:layout_height="wrap_content"
        android:text="@string/goalsguestfield" />

    <EditText
        android:id="@+id/GoalsGuest"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:inputType="text" />

    </LinearLayout>
</LinearLayout>
</LinearLayout>
</ScrollView>
</RelativeLayout>
```

Файл /res/layout/item.xml

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

    <LinearLayout
        android:layout_height="wrap_content"
        android:layout_width="wrap_content"
        android:orientation="horizontal"
        android:paddingLeft="5dp" >
        <TextView
            android:id="@+id/TeamHome"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textSize="20sp"
            android:textStyle="bold"
            android:textColor="#677566"
            />
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textSize="20sp"
            android:textStyle="bold"
            android:textColor="#677566"
            android:text="-"
            />
        <TextView
            android:id="@+id/TeamGuest"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textSize="20sp"
            android:textStyle="bold"
            android:textColor="#677566"
            />
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textSize="20sp"
            android:textStyle="bold"
            android:textColor="#677566"
            android:text="     "
            />
        <TextView
            android:id="@+id/TeamTotal"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textSize="20sp"
            android:textStyle="bold"
            android:textColor="#677566"
            />
    </LinearLayout>
</LinearLayout>
```

## Файл /res/values/strings.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="hello">Hello World, SimpleDBActivity!</string>
    <string name="app_name">SimpleDB</string>
    <string name="title">Чемпионат мира по футболу</string>
    <string name="teamhomefield">Команда хозяев</string>
    <string name="teamguestfield">Команда гостей</string>
    <string name="goalshomefield">Голы хозяев</string>
    <string name="goalsguestfield">Голы гостей</string>
    <string name="save">Сохранить</string>
    <string name="edit">Редактировать</string>
    <string name="delete">Удалить</string>
    <string name="cancel">Отмена</string>
    <string name="icon">Icon</string>
    <string name="deleteAll">Удалить все</string>
    <string name="exit">Выход</string>
    <string name="add">Добавить</string>
</resources>
```

## Файл /res/menu/menu\_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<menu 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"
    tools:context="com.example.op.simpledb.MainActivity">
    <item android:id="@+id/add"
        android:title="@string/add"
        android:icon="@android:drawable/ic_menu_add"/>

    <item android:id="@+id/deleteAll"
        android:title="@string/deleteAll"
        android:icon="@android:drawable/ic_menu_delete"/>

    <item android:id="@+id/exit"
        android:title="@string/exit"
        android:icon="@android:drawable/ic_menu_close_clear_cancel"/>
</menu>
```

## Файл /res/menu/context\_menu.xml

```
<?xml version="1.0" encoding="utf-8"?>
<menu
    xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/edit"
        android:title="@string/edit" />
    <item android:id="@+id/delete"
        android:title="@string/delete" />
</menu>
```

## Файл класса Matches

```
public class Matches implements Serializable{  
    private long id;  
    private String teamhouse;  
    private String teamguest;  
    private int goalshouse;  
    private int goalsguest;  
  
    public Matches (long id, String teamh, String teamg, int gh,int gg) {  
        this.id = id;  
        this.teamhouse = teamh;  
        this.teammguest = teamg;  
        this.goalshouse = gh;  
        this.goalsguest=gg;  
    }  
  
    public long getId() {  
        return id;  
    }  
  
    public String getTeamhouse() {  
        return teamhouse;  
    }  
  
    public String getTeamguest() {  
        return teamguest;  
    }  
  
    public int getGoalshouse() {  
        return goalshouse;  
    }  
  
    public int getGoalsguest() {  
        return goalsguest;  
    }  
}
```

Файл DBMatches.java

```
public class DBMatches {

    private static final String DATABASE_NAME = "simple.db";
    private static final int DATABASE_VERSION = 1;
    private static final String TABLE_NAME = "tableMatches";

    private static final String COLUMN_ID = "id";
    private static final String COLUMN_TEAMHOME = "TeamHome";
    private static final String COLUMN_TEAMGUAST = "TeamGuest";
    private static final String COLUMN_GOALSHOME = "GoalsHome";
    private static final String COLUMN_GOALSGUAST = "GoalsGuast";

    private static final int NUM_COLUMN_ID = 0;
    private static final int NUM_COLUMN_TEAMHOME = 1;
    private static final int NUM_COLUMN_TEAMGUAST = 2;
    private static final int NUM_COLUMN_GOALSHOME = 3;
    private static final int NUM_COLUMN_GOALSGUEST = 4;

    private SQLiteDatabase m DataBase;

    public DBMatches(Context context) {
        OpenHelper mOpenHelper = new OpenHelper(context);
        m DataBase = mOpenHelper.getWritableDatabase();
    }

    public long insert(String teamhouse, String teamguest, int goalshouse, int goalsguest) {
        ContentValues cv = new ContentValues();
        cv.put(COLUMN_TEAMHOME, teamhouse);
        cv.put(COLUMN_TEAMGUAST, teamguest);
        cv.put(COLUMN_GOALSHOME, goalshouse);
        cv.put(COLUMN_GOALSGUAST, goalsguest);
        return m DataBase.insert(TABLE_NAME, null, cv);
    }

    public int update(Matches md) {
        ContentValues cv = new ContentValues();
        cv.put(COLUMN_TEAMHOME, md.getTeamhouse());
        cv.put(COLUMN_TEAMGUAST, md.getTeamguest());
        cv.put(COLUMN_GOALSHOME, md.getGoalshouse());
        cv.put(COLUMN_GOALSGUAST, md.getGoalsguest());
        return m DataBase.update(TABLE_NAME, cv, COLUMN_ID + " = ?", new String[] { String.valueOf(md.getId()) });
    }

    public void deleteAll() {
        m DataBase.delete(TABLE_NAME, null, null);
    }

    public void delete(long id) {
        m DataBase.delete(TABLE_NAME, COLUMN_ID + " = ?", new String[] { String.valueOf(id) });
    }
}
```

```

public Matches select(long id) {
    Cursor mCursor = m DataBase.query(TABLE_NAME, null, COLUMN_ID + " = ?", new String[]
{String.valueOf(id)}, null, null, null);

    mCursor.moveToFirst();
    String TeamHome = mCursor.getString(NUM_COLUMN_TEAMHOME);
    String TeamGuest = mCursor.getString(NUM_COLUMN_TEAMGUAST);
    int GoalsHome = mCursor.getInt(NUM_COLUMN_GOALSHOME);
    int GoalsGuest=mCursor.getInt(NUM_COLUMN_GOALSGUEST);
    return new Matches(id, TeamHome, TeamGuest, GoalsHome,GoalsGuest);
}

public ArrayList<Matches> selectAll() {
    Cursor mCursor = m DataBase.query(TABLE_NAME, null, null, null, null, null, null);

    ArrayList<Matches> arr = new ArrayList<Matches>();
    mCursor.moveToFirst();
    if (!mCursor.isAfterLast()) {
        do {
            long id = mCursor.getLong(NUM_COLUMN_ID);
            String TeamHome = mCursor.getString(NUM_COLUMN_TEAMHOME);
            String TeamGuest = mCursor.getString(NUM_COLUMN_TEAMGUAST);
            int GoalsHome = mCursor.getInt(NUM_COLUMN_GOALSHOME);
            int GoalsGuest=mCursor.getInt(NUM_COLUMN_GOALSGUEST);
            arr.add(new Matches(id, TeamHome, TeamGuest, GoalsHome,GoalsGuest));
        } while (mCursor.moveToNext());
    }
    return arr;
}

private class OpenHelper extends SQLiteOpenHelper {

    OpenHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }
    @Override
    public void onCreate(SQLiteDatabase db) {
        String query = "CREATE TABLE " + TABLE_NAME + " (" +
                COLUMN_ID + " INTEGER PRIMARY KEY AUTOINCREMENT, " +
                COLUMN_TEAMHOME+ " TEXT, " +
                COLUMN_TEAMGUAST + " TEXT, " +
                COLUMN_GOALSHOME + " INT,"+
                COLUMN_GOALSGUAST+" INT);";
        db.execSQL(query);
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        db.execSQL("DROP TABLE IF EXISTS " + TABLE_NAME);
        onCreate(db);
    }
}

```

```
}
```

Файл MainActivity.java

```
public class MainActivity extends Activity {  
    DBMatches mDBConnector;  
    Context mContext;  
    ListView mListview;  
    SimpleCursorAdapter scAdapter;  
    Cursor cursor;  
    myListAdapter myAdapter;  
  
    int ADD_ACTIVITY = 0;  
    int UPDATE_ACTIVITY = 1;  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
  
        mContext=this;  
        mDBConnector=new DBMatches(this);  
        mListview=(ListView)findViewById(R.id.list);  
        myAdapter=new myListAdapter(mContext,mDBConnector.selectAll());  
        mListview.setAdapter(myAdapter);  
        registerForContextMenu(mListview);  
  
    }  
  
    @Override  
    public boolean onCreateOptionsMenu(Menu menu) {  
        // Inflate the menu; this adds items to the action bar if it is present.  
        getMenuInflater().inflate(R.menu.menu_main, menu);  
        return true;  
    }  
  
    @Override  
    public boolean onOptionsItemSelected(MenuItem item) {  
        switch (item.getItemId()) {  
            case R.id.add:  
                Intent i = new Intent(mContext, AddActivity.class);  
                startActivityForResult (i, ADD_ACTIVITY);  
                updateList();  
                return true;  
            case R.id.deleteAll:  
                mDBConnector.deleteAll();  
                updateList();  
                return true;  
            case R.id.exit:  
                finish();  
                return true;  
            default:  
                return super.onOptionsItemSelected(item);  
        }  
    }  
  
    @Override  
    public void onCreateContextMenu(ContextMenu menu, View v, ContextMenu.ContextMenuItemInfo m
```

```

    menuInfo) {
        super.onCreateContextMenu(menu, v, menuInfo);
        MenuInflater inflater = getMenuInflater();
        inflater.inflate(R.menu.context_menu, menu);
    }

    @Override
    public boolean onContextItemSelected(MenuItem item) {
        AdapterView.AdapterContextMenuInfo info = (AdapterView.AdapterContextMenuInfo) item
m getMenuInfo();
        switch(item.getItemId()) {
            case R.id.edit:
                Intent i = new Intent(mContext, AddActivity.class);
                Matches md = mDBConnector.select(info.id);
                i.putExtra("Matches", md);
                startActivityForResult(i, UPDATE_ACTIVITY);
                updateList();
                return true;
            case R.id.delete:
                mDBConnector.delete (info.id);
                updateList();
                return true;
            default:
                return super.onContextItemSelected(item);
        }
    }
    private void updateList () {
        myAdapter.setArrayMyData(mDBConnector.selectAll());
        myAdapter.notifyDataSetChanged();
    }

    @Override
    protected void onActivityResult(int requestCode, int resultCode, Intent data) {

        if (resultCode == Activity.RESULT_OK) {
            Matches md = (Matches) data.getExtras().getSerializable("Matches");
            if (requestCode == UPDATE_ACTIVITY)
                mDBConnector.update(md);
            else
                mDBConnector.insert(md.getTeamhouse(), md.getTeamguest(), md.getGoalshouse
()), md.getGoalsguest());
            updateList();
        }
    }

    class myListAdapter extends BaseAdapter {
        private LayoutInflater mLayoutInflater;
        private ArrayList<Matches> arrayMyMatches;

        public myListAdapter (Context ctx, ArrayList<Matches> arr) {
            mLayoutInflater = LayoutInflater.from(ctx);
            setArrayMyData(arr);
        }
    }
}

```

```

public ArrayList<Matches> getArrayMyData() {
    return arrayMyMatches;
}

public void setArrayMyData(ArrayList<Matches> arrayMyData) {
    this.arrayMyMatches = arrayMyData;
}

public int getCount () {
    return arrayMyMatches.size();
}

public Object getItem (int position) {

    return position;
}

public long getItemId (int position) {
    Matches md = arrayMyMatches.get(position);
    if (md != null) {
        return md.getId();
    }
    return 0;
}

public View getView(int position, View convertView, ViewGroup parent) {

    if (convertView == null)
        convertView = mLayoutInflater.inflate(R.layout.item, null);

    TextView vTeamHome= (TextView)convertView.findViewById(R.id.TeamHome);
    TextView vTeamGuest = (TextView)convertView.findViewById(R.id.TeamGuest);
    TextView vTotal=(TextView)convertView.findViewById(R.id.TeamTotal);

    Matches md = arrayMyMatches.get(position);
    vTeamHome.setText(md.getTeamhouse());
    vTeamGuest.setText(md.getTeamguest());
    vTotal.setText(md.getGoalshouse()+":"+md.getGoalsguest());

    return convertView;
}
} // end myAdapter
}

```

Файл AddActivity.java

```
public class AddActivity extends Activity {
    private Button btSave,btCancel;
    private EditText etTeamHome,etTeamGuest,etGoalsHome,etGoalsGuest;
    private Context context;
    private long MyMatchID;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_add);

        etTeamHome=(EditText)findViewById(R.id.TeamHome);
        etTeamGuest=(EditText)findViewById(R.id.TeamGuest);
        etGoalsHome=(EditText)findViewById(R.id.GoalsHome);
        etGoalsGuest=(EditText)findViewById(R.id.GoalsGuest);
        btSave=(Button)findViewById(R.id.butSave);
        btCancel=(Button)findViewById(R.id.butCancel);

        if(getIntent().hasExtra("Matches")){
            Matches matches=(Matches) getIntent().getSerializableExtra("Matches");
            etTeamHome.setText(matches.getTeamhouse());
            etTeamGuest.setText(matches.getTeamguest());
            etGoalsHome.setText(Integer.toString(matches.getGoalshouse()));
            etGoalsGuest.setText(Integer.toString(matches.getGoalsguest()));
            MyMatchID=matches.getId();
        }
        else
        {
            MyMatchID=-1;
        }
        btSave.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Matches matches=new Matches(MyMatchID,etTeamHome.getText().toString(),etTeamGuest.getText().toString(),
                        etGoalsHome.getText().toString(),
                        etGoalsGuest.getText().toString());
                Integer.parseInt(etGoalsHome.getText().toString()),
                Integer.parseInt(etGoalsGuest.getText().toString());
                Intent intent=getIntent();
                intent.putExtra("Matches",matches);
                setResult(RESULT_OK,intent);
                finish();
            }
        });
    });

    btCancel.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            finish();
        }
    });
}
```

После запуска приложения нажимаем на кнопку настроек и в меню выбираем «Добавить», чтобы внести новую запись в БД. Открывается другая Activity для ввода данных. Затем сохраним и видим введенную информацию на экране (рис. 4.24).

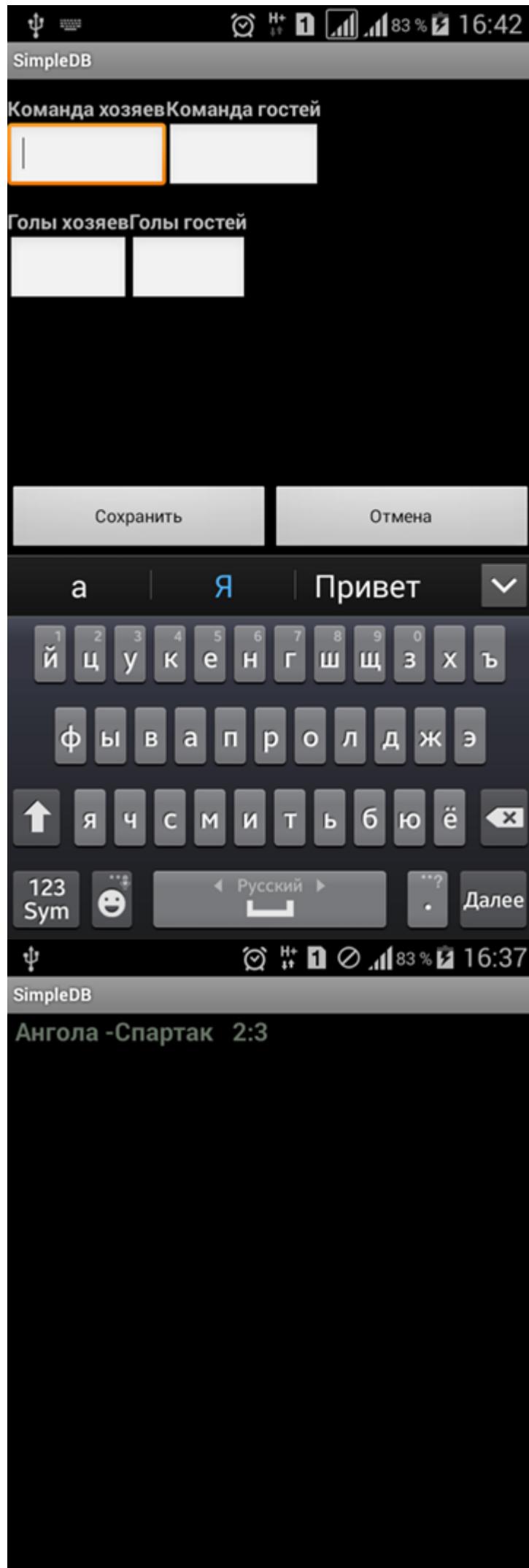




Рис. 4.24.

При нажатии на строчку с результатами игр появляется меню, с помощью которого мы можем удалить или отредактировать ее.

*Компания Samsung Electronics выражает благодарность за участие в подготовке данного материала преподавателям IT ШКОЛЫ SAMSUNG Дубинину Андрею Валентиновичу и Плотникову Денису Геннадьевичу.*

## **4.6. Рекурсия**

Сайт: IT Академия SAMSUNG

Курс: MDev @ IT Академия Samsung

Книга: 4.6. Рекурсия

Напечатано:: Егор Беляев

Дата: Суббота, 18 Апрель 2020, 19:29

# **Оглавление**

- 4.6.1. Рекурсия в программировании и не только
- 4.6.2. Стек вызовов
- 4.6.3. Линейная рекурсия
- 4.6.4. Ветвящаяся рекурсия

## 4.6.1. Рекурсия в программировании и не только

**Рекурсия** — определение, описание, изображение какого-либо объекта или процесса внутри самого этого объекта или процесса, то есть ситуация, когда объект является частью самого себя. Термин «рекурсия» используется в различных специальных областях знаний — от лингвистики до логики, но наиболее широкое применение находит в математике и информатике. В качестве примеров рекурсии в изображениях можно привести знаменитые фракталы: ковер Серпинского (рис. 4.25) и снежинку Коса (рис. 4.26).

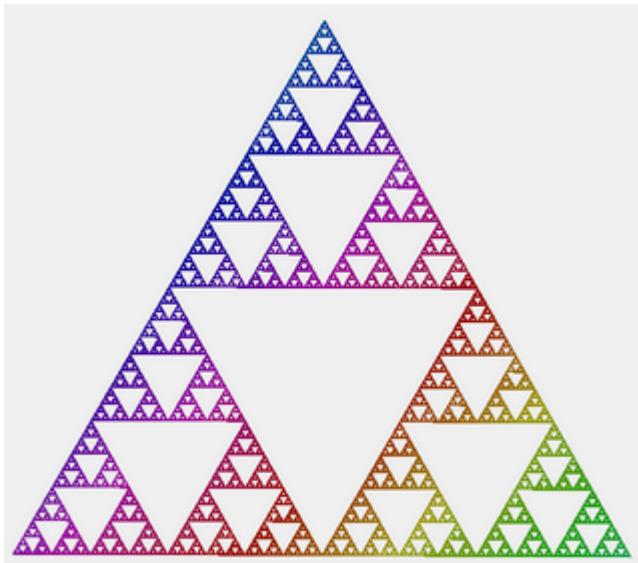


Рис. 4.25.

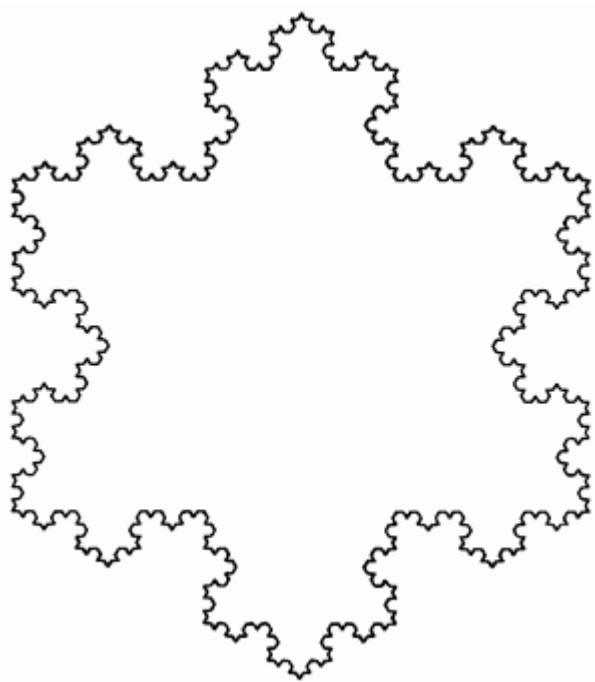


Рис. 4.26.

Еще один пример из жизни. Можно взять зеркало и встать перед другим зеркалом. В итоге вы увидите бесконечное повторение своего изображения.

Канонический пример из математики: факториал целого неотрицательного числа, определенный рекурсивно:

$$n! = \begin{cases} n \cdot (n-1)!, & n > 0 \\ 1, & n = 0 \end{cases}$$

И наконец, в программировании есть рекурсивные функции (методы). Рекурсивные методы внутри своего тела вызывают сами себя.

Рекурсия не слишком широко используется в программировании. Любой циклический алгоритм можно реализовать рекурсивно, однако обратное правило неверно: не каждый рекурсивный алгоритм можно реализовать с помощью цикла. К примеру, при работе со структурами данных типа «дерево», которые мы изучим в следующей теме, обойтись без рекурсии нельзя.

Рекурсия имеет ряд существенных минусов, о которых буде изложено далее. Следовательно, если существует выбор, то следует использовать циклы. Многие поклонники рекурсии используют ее в силу ее лаконичности и выразительности.

## 4.6.2. Стек вызовов

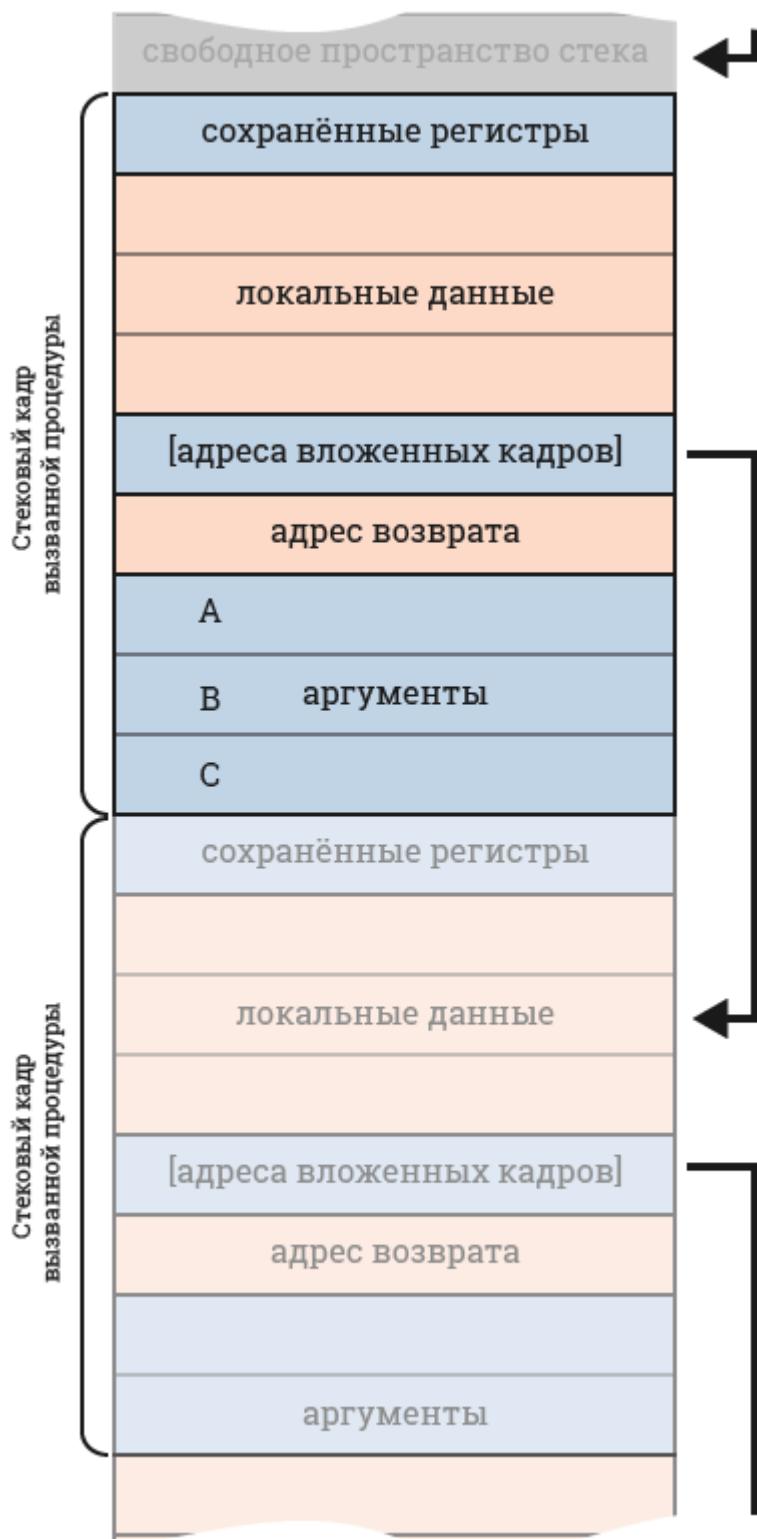


Рис. 4.27.

Рекурсию сложно изучить без понимания такого ключевого понятия, как стек вызовов. В одном потоке на данный момент времени исполняется только один метод программы. При этом, какой метод запускать, понятно только по ходу выполнения программы. Возникает вопрос, как вернуться назад в точку программы после выполненного метода. Для этого применяется стек вызовов.

Ранее мы уже изучили структуру данных «стек» и помним, что он работает на основе списка с дисциплиной обслуживания LIFO «последним пришел — первым ушел» (вспомните колбочку и шайбы).

В стеке вызовов хранится информация для возврата управления

из вызываемых методов в метод его вызвавший.

Пусть метод А вызывает метод Б. При вызове метода Б в стек заносится адрес точки возврата — адрес в памяти инструкции в методе А, которая должна выполниться после выполнения метода Б. Если метод Б вызовет еще один метод С, то в стек вызовов будет занесен очередной адрес возврата — инструкция из метода Б и т. д.

При возврате из метода С адрес возврата снимается со стека и управление передается на следующую инструкцию в приостановленном методе Б. По завершении метода Б из стека будет снят адрес возврата в метод А и т. д.

В реальности для языков высокого уровня, такого как Java, в стеке хранится не только адрес возврата, но и другая необходимая информация, например, аргументы (параметры) и локальные переменные метода (см. рис. 4.27).

Становится понятным, что каждый вызов метода занимает объем памяти в стеке и тем самым уменьшает доступную оперативную память.

### 4.6.3. Линейная рекурсия

Подробнее разберем принцип написания рекурсивных программ на примере вычисления целой степени числа. Следует отметить, что данный пример выбран лишь для иллюстрации работы рекурсии и в реальности таким способом данную задачу решать не рекомендуется.

n-я степень числа X получается путем умножения самого числа X на его (n-1) степень. Нулевая степень числа равна 1, число в первой степени равно самому числу (это и есть условия завершения рекурсии).

$$x = \begin{cases} 1, & n = 0 \\ x \cdot x^{n-1}, & n > 0 \\ \frac{1}{x^{[n]}}, & n < 0 \end{cases}$$

Таким образом, можно констатировать следующее:

- при возведении числа в степень 0 результат равен 1;
- n-я положительная степень числа X получается путем домножения самого X на (n-1) степень этого числа;
- n-я отрицательная степень числа X получается путем деления 1 на X, умноженное на (n-1) степень этого числа.

Для вычисления степени можно составить следующую программу:

```
public class MyMath {

    /* Рекурсия: вычисление целой степени числа */
    static double nPow(double x, int n)
    {
        double y;
        if (n == 0) /* условие завершения рекурсии */
            return 1;
        if (n < 0)
            y = 1. / nPow(x, -n);
        else
            y = x * nPow(x, n - 1);
        return y;
    }

    public static void main(String[] args) {
        double x = 2.; //основание
        int n = 3; //степень
        System.out.println(x + " ^ " + n + " = " + nPow(x, n));
    }
}
```

В результате работы программы на экране появится следующее:

```
2.0 ^ 3 = 8.0
```

Теперь, понимая, как работает стек вызовов, мы можем представить работу рекурсивных методов. Здесь нужно отчетливо осознавать, что каждый рекурсивный вызов — это не вызов одного и того же метода. Для компилятора это совершенно разные функции с собственными локальными переменными и аргументами.

Далее на рисунке 4.28 приведена иллюстрация работы метода на примере вычисления  $2^3$ . Черными стрелками обозначены рекурсивные вызовы, зелеными — обратный ход рекурсии и возвращаемые значения. В данном случае произошло три рекурсивных вызова. Иллюстрация показывает, что цепочка вызовов линейна, такие алгоритмы называют *линейной рекурсией*.

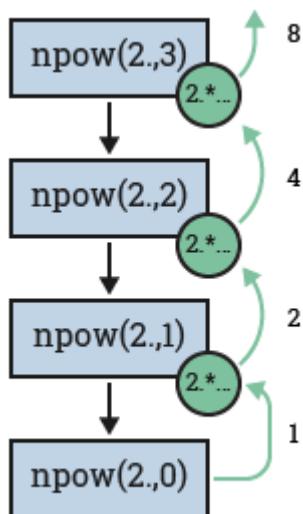


Рис. 4.28.

В данном случае степень вычисляется, спускаясь от  $n$  вниз.

Условие  $n=0$  — это признак остановки рекурсии.

Когда цепочка рекурсивных вызовов достигнет данного значения стек вызовов начнет освобождаться в обратном порядке. Каждый рекурсивный алгоритм должен содержать условие остановки — условие завершения рекурсии, в обратном случае алгоритм будет бесконечным.

В данном примере каждый раз при рекурсивном вызове метода nPow() стек пополняется. Очевидно, что при отсутствии условия остановки он переполнится и возникнет ошибка. Кроме того, на вызов очередного метода необходимо больше времени чем на одну итерацию цикла. В этом и заключаются основные недостатки рекурсивных алгоритмов.

Поэкспериментируйте с программой:

- при каком значении  $n$  время выполнения программы значительно замедлится (больше 1 мин)?
- реализуйте возведение в степень через цикл, при каком значении  $n$  время выполнения программы значительно замедлится (больше 1 мин)?

## 4.6.4. Ветвящаяся рекурсия

Кроме линейной рекурсии существует ветвящаяся. Она возникает тогда, когда из метода происходит более чем один рекурсивный вызов. Разберем ее на примере вычисления чисел Фибоначчи.

**Числа Фибоначчи** — элементы числовой последовательности, в которой каждое последующее число равно сумме двух предыдущих чисел (см. табл. 4.17).

Число Фибоначчи	1	1	2	3	5	8	13	21	34	55	...
n	0	1	2	3	4	5	6	7	8	9	...

Табл. 4.17.

Числа Фибоначчи получаются следующим образом:

- n-е число Фибоначчи — это сумма двух предыдущих чисел Фибоначчи (n-1) и (n-2);
- выделяются из общего правила два первых числа Фибоначчи: нулевое и первое числа равны 1 (это и будет условием выхода из рекурсии).

Программа по вычислению чисел Фибоначчи может выглядеть следующим образом:

```
public class MyMath {  
  
    /* Рекурсия: вычисление чисел Фибоначчи */  
    static long nFib(int n) {  
        if (n == 0 || n == 1)  
            return 1; // Условие завершения рекурсии — «заглушка»  
        return nFib(n - 1) + nFib(n - 2); //возвращаем сумму предыдущих чисел  
    }  
  
    public static void main(String[] args) {  
        // вывод первых 7 чисел Фибоначчи  
        for (int i = 0; i < 7; i++)  
            System.out.println("Fibonacci(" + i + ")=" + nFib(i));  
    }  
}
```

В результате работы программы на экран выводится следующее:

```
Fibonacci(0)=1  
Fibonacci(1)=1  
Fibonacci(2)=2  
Fibonacci(3)=3  
Fibonacci(4)=5  
Fibonacci(5)=8  
Fibonacci(6)=13
```

На рисунке 4.29 приведена иллюстрация работы рекурсивной функции на примере вычисления пятого числа Фибоначчи nFib(5). Черными стрелками показаны рекурсивные вызовы, зелеными — обратный ход рекурсии и возвращаемые значения. В данном примере произошло четырнадцать рекурсивных вызовов функции nFib().

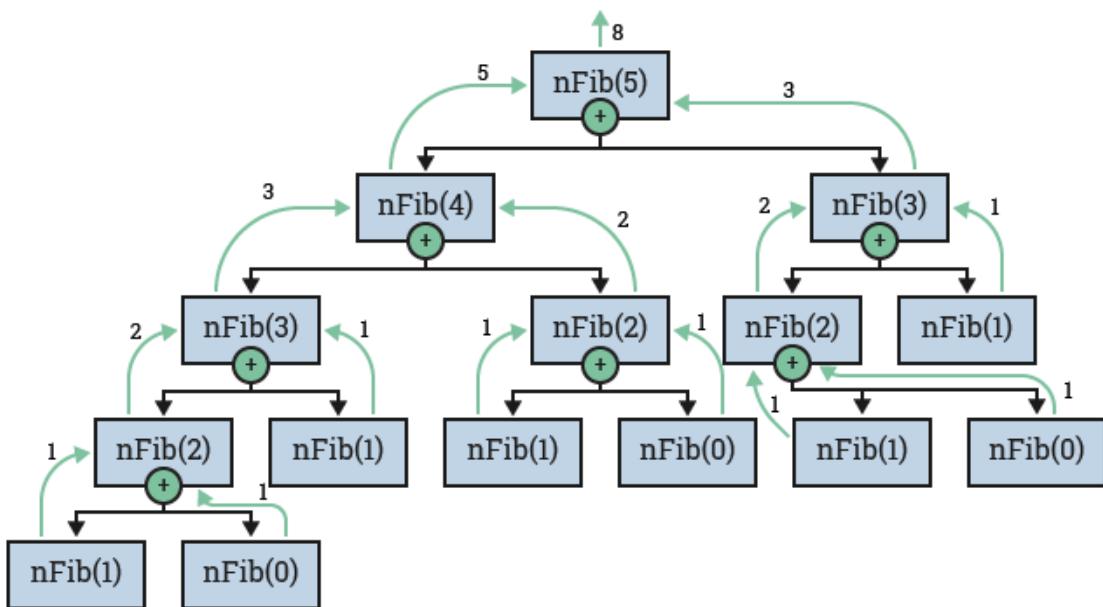


Рис. 4.29.

При изучении рисунка становится понятным, что данный алгоритм является не вполне эффективным. Например, очевидны повторения: целые ответвления совпадают в различных частях иерархической структуры. Это значит, что мы вычисляли уже известные числа Фибоначчи, что является еще одним существенным минусом рекурсивных алгоритмов.

### Пример 4.20

Разработаем приложение, которое рисует рекурсивные фигуры на плоскости. Часто в литературе такие фигуры называют ***фракталами***.

Пусть наше приложение на экране устройства рекурсивно рисует окружности, как это показано на рисунке 4.30. Далее, меняя параметры в рекурсивных функциях, мы сможем убедиться, что сложность и форма рисунка зависит только от нашей фантазии.

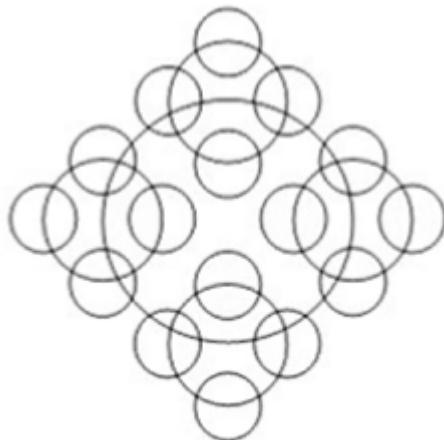


Рис. 4.30.

Воспользуемся уже знакомым нам способом рисования. Создадим новый класс MyDraw и унаследуем его от View.

Проделаем те же действия, что мы выполняли в разделе 3.7.1.

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);
    }
}
```

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);
}
```

Параметры рисования определяются в объекте класса Paint. Кроме того, нам понадобятся размеры холста h и w.

```
Paint paint;
int w, h;
```

В конструкторе класса MyDraw создадим объект типа Paint и зададим необходимые стили рисования.

```
paint = new Paint();
paint.setColor(Color.GREEN);
paint.setStyle(Paint.Style.STROKE);
paint.setStrokeWidth(3);
```

Для инициализации размеров холста переопределим метод onSizeChanged() следующим образом:

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

В классе MyDraw создадим функцию drawCircles(). В качестве параметров передадим ей ссылку на объект типа Canvas, координаты центра и радиус окружности. Нарисовать картинку, подобную рисунку выше, можно, вызвав данный метод рекурсивно, но с другими параметрами. При этом необходимо предусмотреть условие остановки вызова функции (например, это может быть значение радиуса окружности). Метод drawCircles() может выглядеть следующим образом:

```
public void drawCircles(Canvas canvas, int x, int y, int r){  
    canvas.drawCircle(x, y, r, paint);  
  
    if(r > 50) {  
        drawCircles(canvas, x, y - r, r / 2);  
        drawCircles(canvas, x + r, y, r / 2);  
        drawCircles(canvas, x, y + r, r / 2);  
        drawCircles(canvas, x - r, y, r / 2);  
    }  
}
```

Метод будет вызывать себя до тех пор, пока значение радиуса будет больше пяти пикселей, при этом каждый раз при новом рекурсивном вызове значение радиуса уменьшается вдвое.

Метод drawCircles() будет вызывать в методе onDraw(), при этом первая окружность будет нарисована в центре экрана.

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

После запуска приложения на экране появится следующая картина (рис. 4.31).

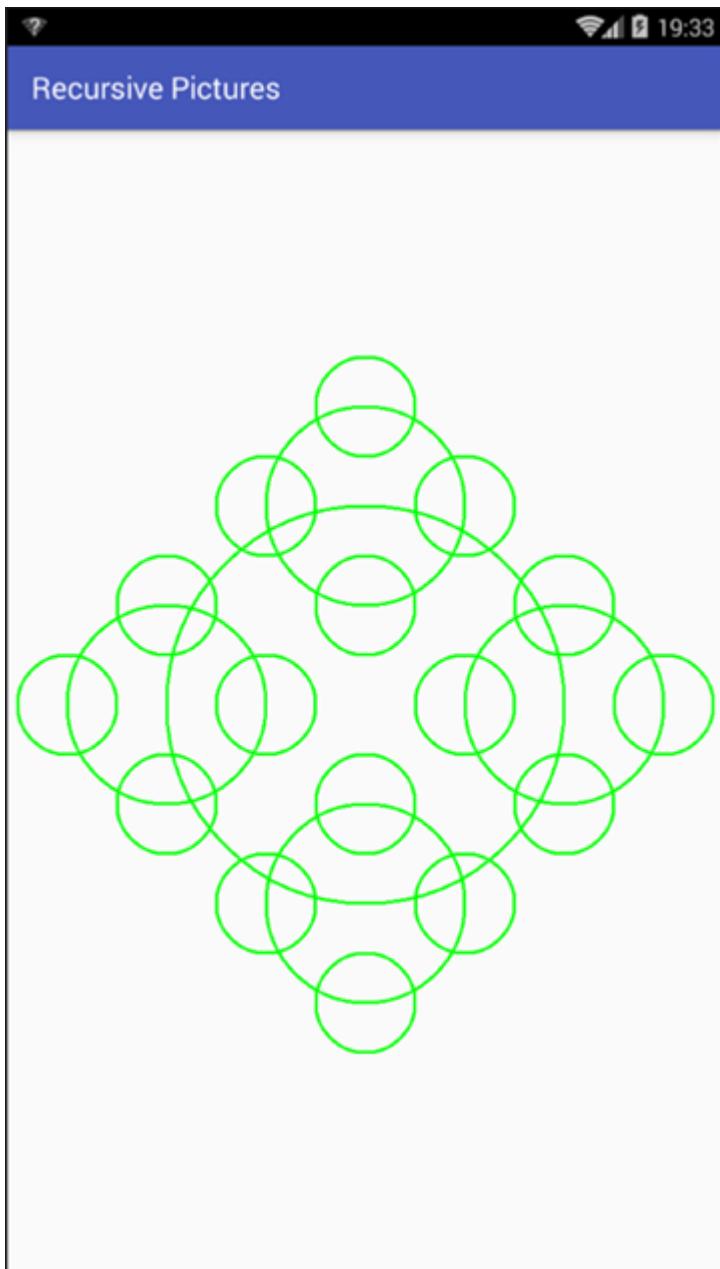


Рис. 4.31.

При изменении условия остановки рекурсивных вызовов функции drawCircles() можно получить абсолютно другую картину. Например, если в условии рисовать окружности с радиусом больше десяти пикселей ( $r > 10$ ), то получится следующее (рис. 4.32).

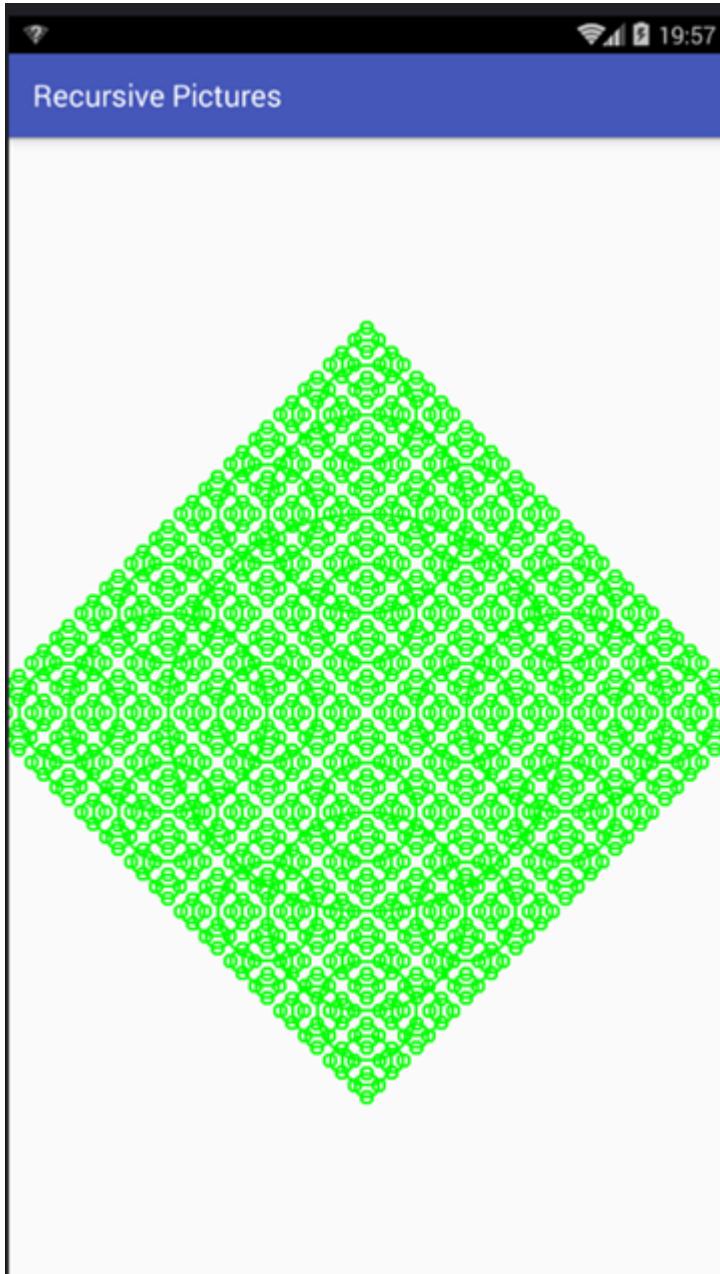


Рис. 4.32.

Теперь меняя параметры в функции drawCircles(), добавляя рисование других фигур, вы можете увидеть разнообразные изображения.

### Пример 4.21

Запустите приведенное Android-приложение, реализующее вывод каталогов устройства, начиная с заданной, и вывод на экран списка обнаруженных папок.

Определим разметку следующим образом:

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context="com.samsung.itschool.recursionexample.MainActivity">
    <TextView
        android:id="@+id/tvLog"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>
</ScrollView>
```

Далее приведен код с комментариями.

```
package com.samsung.itschool.recursionexample;

import android.os.AsyncTask;
import android.os.Bundle;
import android.os.Environment;
import android.support.v7.app.AppCompatActivity;
import android.widget.TextView;

import java.io.File;

public class MainActivity extends AppCompatActivity {

    //строка для хранения списка директорий
    private StringBuilder strTree = new StringBuilder();

    //поле для вывода результатов работы программы на экран
    private TextView logTview;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        logTview = (TextView) findViewById(R.id.tvLog);
        //Создаем AsyncTask для работы с рекурсией
        assyncRecursion task = new assyncRecursion();
        //Запускаем AsyncTask
        task.execute();
    }

    //Класс для выполнения тяжелых задач в отдельном потоке и передача в UI-поток результатов работы.
    private class assyncRecursion extends AsyncTask<Void, Void, Void> {

        /*Рекурсивный метод формирует текст со списком директорий
        String path - путь к директории
        String indent - строка, иллюстрирующая глубину вложения директории (длина ветки дерева директорий),
        за каждый уровень добавляем --
        */
        private void recursiveCall(String path, String indent) {
            //получаем список файлов и директорий внутри текущей директории
            File[] fileList = new File(path).listFiles();
            for (File file : fileList) {
                if (file.isDirectory()) { //для каждой директории из списка
                    //сохраняем глубину вложения и имя директории в строку strTree
                    strTree.append(indent).append(file.getName()).append("\n");
                    //рекурсивный вызов для каждой директории из списка
                    recursiveCall(file.getAbsolutePath(), indent + "-- ");
                }
            }
        }
    }
}

//Метод выполняется в отдельном потоке, нет доступа к UI
```

```

@Override
protected Void doInBackground(Void... params) {
    //вызов метода с корневой папкой внешней памяти, глубина 0
    recursiveCall(Environment.getExternalStorageDirectory().getPath(), "");
    return null;
}

//Метод выполняется после doInBackground, есть доступ к UI
@Override
protected void onPostExecute(Void result) {
    super.onPostExecute(result);
    //отображаем сформированный текст со списком директорий
    logTview.setText(strTree);
}

}
}

```

Заметьте, что используем AsyncTask для отделения тяжелой рекурсивной программы в отдельный поток.

При первом вызове рекурсивного метода передается путь до внешней памяти устройства с помощью библиотечного метода Environment.getExternalStorageDirectory().getPath().



В Android-устройстве выделяют следующие виды памяти:

- **внутренняя (internal) память** — это часть встроенной в телефон карты памяти. При ее использовании по умолчанию папка приложения защищена от доступа других приложений (Using the Internal Storage);
- **внешняя (external) память** — это общее «внешнее хранилище» для ваших файлов, которое может быть как на встроенной, так и SD-карте. Но, как правило, в современных версиях Android — это часть встроенной памяти (Using the External Storage);
- **удаляемая (removable) память** — все хранилища, которые могут быть удалены из устройства пользователем, например, SD-карта.

В результате работы программы на экране появится иерархия каталогов (рис. 4.33).



Рис. 4.33.

## **4.7. Деревья**

Сайт: IT Академия SAMSUNG

Курс: MDev @ IT Академия Samsung

Книга: 4.7. Деревья

Напечатано:: Егор Беляев

Дата: Суббота, 18 Апрель 2020, 19:30

# **Оглавление**

- 4.7.1. Дерево. Разновидности деревьев
- 4.7.2. Понятие бинарного дерева
- 4.7.3. Понятие сбалансированного дерева
- 4.7.4. Библиотечный класс TreeSet

## 4.7.1. Дерево. Разновидности деревьев

Деревья принадлежат к числу основных структур данных, используемых в программировании. Они обладают некоторыми уникальными преимуществами, которых не было у других структур данных, рассмотренных ранее. Деревья сочетают в себе преимущества двух других структур: упорядоченного массива и связанного списка. Поиск в сбалансированном дереве поиска — это двоичное дерево, организованное специальным образом, выполняется так же быстро, как в упорядоченном массиве, а операции вставки и удаления элементов — так же быстро, как в связанном списке.

Древовидная структура является одним из способов представления иерархической структуры в графическом виде. Такое название она получила потому, что график выглядит как перевернутое дерево. Корень дерева (корневой узел) находится на самом верху, а листья (потомки) — внизу.

Деревья широко применяются в компьютерных технологиях. Примером является файловая система, представляющая собой иерархическую структуру из файлов и каталогов. Язык XML также имеет древовидную структуру.

В качестве примера древовидной структуры представим оглавление книги (см. рис. 4.34).

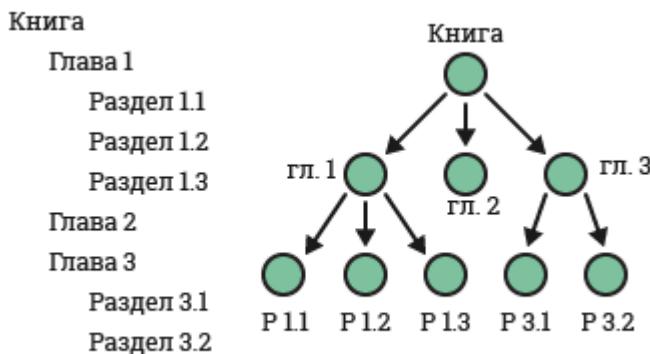


Рис. 4.34.

Узел «Книга» является корнем дерева, так как находится в вершине дерева, книга содержит определенные главы, а главы в свою очередь состоят из разделов, каждый раздел может содержать подразделы и т. д.

На рисунке стрелками изображены родительские отношения (ребра, ветви дерева) между узлами (вершинами) дерева. На верхнем уровне каждый «родитель» указывает стрелкой на своих «потомков». То есть в этой иерархической структуре вершина всегда «знает» своих потомков.

Для того чтобы более точно оперировать структурой Дерево, нужно дать определение некоторым ключевым понятиям:

- корневой узел — самый верхний узел дерева, он не имеет предков (на рисунке узел «Книга»);
- лист, листовой или терминальный узел — конечный узел, то есть не имеющий потомков (на рисунке узлы «Гл. 2», «Р 1.1», «Р 1.2», «Р 1.3», «Р 3.1», «Р 3.2»);
- внутренний узел — любой узел дерева, имеющий потомков, то есть не лист (на рисунке узлы «Гл. 1», «Гл. 3»).

С корневого узла начинается выполнение большинства операций над деревом, потому что, как и в списках, чтобы получить доступ к любому элементу структуры, необходимо, переходя по ветвям, перебирать элементы, начиная с головы — корневого узла. Корневой узел — это своеобразный вход в дерево.

Большинство алгоритмов работы с деревом строятся на том, что каждый узел дерева рассматривается как корневой узел поддерева, «растущего» из этого узла. Такой подход дает возможность зацикливать выполнение операций при прохождении по элементам дерева. Но в связи с тем, что при прохождении по дереву (в отличие от массива) неизвестно сколько шагов будет в этом цикле, используется другой инструмент — рекурсивный вызов.

В данной теме мы рассмотрим только основные классификации деревьев.

1. По числу детей дерева делят на:
  - двоичные (или бинарные): каждый узел может иметь не более двух детей;
  - с числом ветвей больше 2: часто такие деревья называют K-деревьями.
2. По признаку сбалансированности дерева: сбалансированные и несбалансированные.  
Это свойство мы рассмотрим ниже.

## 4.7.2. Понятие бинарного дерева

**Двоичное (бинарное) дерево** — это древовидная структура данных, где каждый узел имеет не более двух детей. Этих детей называют левым (Л) и правым (П) потомком или «сыном». Ниже приведен рисунок 4.35, представляющий двоичное дерево.

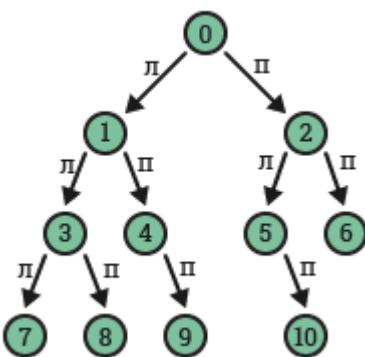


Рис. 4.35.

- Предок — это вершина, из которой исходят ветви к вершинам следующего уровня. Для узла 3 предком является узел 1, левым сыном — 7, правым — 8;
- потомок — это вершина, в которую входят ветви, исходящие из одной общей вершины. Левый потомок является корневым узлом левого поддерева, а правый потомок — корневым узлом правого поддерева;
- уровень вершины — это количество дуг от корня дерева до вершины;
- высотой дерева — максимальный уровень вершины. На рисунке от узла 6 до корня узлы располагаются на трех уровнях, а от узлов 7, 8, 9, 10 — на четырех, высота дерева равна 3;
- ключами называются значения узлов.

Двоичным (бинарным) деревом поиска называется структура, которая обладает следующим свойством: если  $x$  — узел бинарного дерева с ключом  $k$ , то все узлы в левом поддереве должны иметь ключи, меньшие  $k$ , а в правом поддереве большие  $k$ .

### Упражнение 4.7.1

Реализуем класс бинарного дерева поиска на языке Java для хранения целых чисел. Будем так же, как и при организации связных списков, считать, что `null` — это отсутствие потомка.

Не простой для понимания идеей является то, что узел и дерево — это фактически одно и то же.

- Поля: значение в узле дерева (`value`), ссылка левого потомка (`lchild`) и ссылка на правого потомка (`rchild`).
- Методы: заполнение дерева и печать в консоль содержимого дерева (обхода дерева).

```

public class BinaryTree {

    int value;
    BinaryTree lchild; // левый потомок
    BinaryTree rchild; // правый потомок

    public BinaryTree(int value) {
        this.value = value;
        this.lchild = null;
        this.rchild = null;
    }

    /* метод вставки элементов в дерево
       node - ссылка на текущий узел дерева
       для создания дерева будем передавать null в качестве node
       valueNode - значение, которое добавляем в дерево
    */
    public BinaryTree insertNode(BinaryTree node, int targetValue) {
        // если дерево пустое
        if (node == null) {
            node = new BinaryTree(targetValue);
            return node;
        }
        /*если значение в текущем узле больше valueNode, то переходим в
         левое поддерево*/
        if (node.value > targetValue) {
            //если левого потомка нет, то создаем его с значением valueNode
            if (node.lchild == null)
                return node.lchild = new BinaryTree(targetValue);
            // если левый потомок есть, то переходим ниже в левое поддерево
            else
                return insertNode(node.lchild, targetValue);
        }
        /*если значение в текущем узле меньше вставляемого,
         то переходим в правое поддерево*/
        else if (node.value < targetValue) {
            //если правого потомка нет, то создаем его с значением valueNode
            if (node.rchild == null)
                return node.rchild = new BinaryTree(targetValue);
            // если правый потомок есть, то переходим ниже в правое поддерево
            else
                return insertNode(node.rchild, targetValue);
        }
        // Если ключи одинаковые, вставки не происходит
        return null;
    }

    // метод вывода дерева на экран
    public void printBinaryTree(BinaryTree node, int level) {
        if (node != null) {
            printBinaryTree(node.lchild, level + 1);
            for (int i = 0; i < level; i++)
                System.out.print("    ");//чем ниже уровень, тем отступу больше
        }
    }
}

```

```

        System.out.println(node.value);
        printBinaryTree(node.rchild, level + 1);
    }
}

public static void main(String[] args) {
    // массив значений для добавления в дерево
    int b[] = { 10, 25, 20, 6, 4, 8, 50, 30, 6 };
    BinaryTree tree = new BinaryTree(b[0]);
    // добавление элементов массива в дерево
    for (int i = 1; i < b.length; i++)
        tree.insertNode(tree, b[i]);
    // вывод содержимого дерева
    tree.printBinaryTree(tree, 0);
}
}

```

Запустите программу и на экране получите результат — дерево ветвями вправо.

Чтобы понять результат, разберем алгоритм вставки значения в дерево на нашем примере. Мы поочередно вставляли элементы массива { 10, 25, 20, 6, 4, 8, 50, 30, 6 }.

Вначале дерево было пустым, поэтому 10 был сохранен в корне дерева (рис. 4.36).

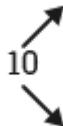


Рис. 4.36.

Следующее значение 25 сравнили с 10, так как оно больше, перешли вправо, правый потомок был пустым, поэтому он был создан конструктором с параметром 25 (рис. 4.37).

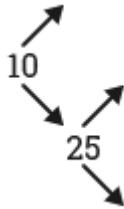


Рис. 4.37.

Третье значение 20 сравнили с 10 — ушли вправо (потому что 20 больше, чем 10), сравнили с 25 — ушли влево (потому что 20 меньше, чем 25) и создали новый узел (рис. 4.38).

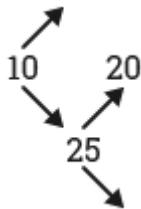


Рис. 4.38.

Аналогично добавляются все значения, кроме последнего. Последнее значение 6 не подпадает ни под одно из условий при сравнении с уже существующим узлом 6. Значение не больше и не меньше, а равно значению в узле дерева! В программе для такого условия никаких

действий не производится. Таким образом, повторяющиеся значения не вставляются, и в результате мы получили дерево (рис. 4.39).

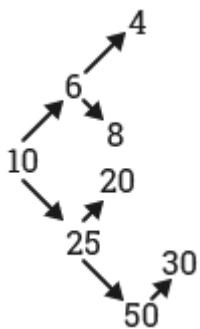


Рис. 4.39.

Методы вставки значений и вывода дерева используют рекурсивные вызовы функции.

Функция вывода работает по так называемому левостороннему обходу:

- начинает с корня;
- спускается по левым ветвям до самого левого листа (4);
- выводит значение листа (4), затем его родителя (6) и правого потомка (8);
- подымается через уровень к родителю родителя и выводит его (10), спускается к правому потомку;
- повторяет все с шага 2 для текущего узла (25), то есть начинает новый левосторонний обход с выводом значений: 20, 25, 30, 50.

Для имитации структуры дерева в методе вывода дерева перед значением узла выводится количество пробелов, пропорциональное уровню узла. Если закомментировать цикл печати пробелов, то будет выведена отсортированная последовательность.

```
4 6 8 10 20 25 30 50
```

### 4.7.3. Понятие сбалансированного дерева

В предыдущем упражнении мы создали такое бинарное дерево, в котором для каждого узла выполняются следующие правила:

- все значения в узлах его левого поддерева меньше значения (ключа) этого узла;
- все значения в узлах его правого поддерева больше значения (ключа) этого узла;
- одинаковые значения в дереве не допускаются.

Очевидно, что в таком дереве легко найти элемент, двигаясь от корня и переходя на левое или правое поддерево в зависимости от значения (ключа) в каждом узле.

Дерево позволяет легко реализовать поиск, а также позволяет быстро выполнять операции вставки и удаления элементов. Другие структуры данных — массивы, сортированные массивы, связанные списки — по крайней мере в одной из этих областей работают недостаточно быстро.

Однако двоичный поиск предполагает, что диапазон поиска каждый раз делится пополам, а построенное дерево вовсе не обязательно может обеспечивать такое условие.

Например, запустим нашу программу из Упражнения 4.7.1 для уже упорядоченного массива  $\{10, 15, 28, 30, 45, 50, 70\}$ , получим дерево (рис. 4.40).

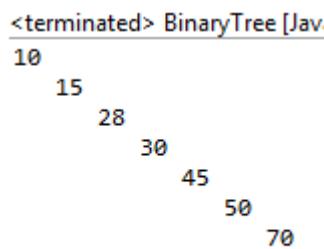


Рис. 4.40.

Узлы выстраиваются в линию без ветвления. Такую структуру называют «бамбук». Поскольку каждый узел больше узла, вставленного перед ним, каждый узел является правым потомком, поэтому все узлы располагаются по одну сторону от корня. Дерево получается предельно несбалансированным.

Если вставить элементы, упорядоченные по убыванию, то каждый узел будет левым потомком своего родителя, а дерево окажется несбалансированным с другой стороны.

Сбалансированное дерево — это дерево, для каждой вершины которого высота ее двух поддеревьев различается не более чем на 1.

Наше первое дерево было сбалансированным, последнее дерево — крайне несбалансированное.

В несбалансированном дереве теряется возможность быстрого поиска, а также вставки или удаления. Трудоемкость поиска в последнем дереве ничем не отличается от поиска в списке.

Чтобы обеспечить быстрое время поиска, на которое способно дерево, необходимо позаботиться о том, чтобы дерево всегда оставалось сбалансированным (или по крайней мере почти сбалансированным).

Методов получения сбалансированного дерева разработано множество приемов, но все они достаточно сложны, и мы не будем рассматривать их в данной теме.

## 4.7.4. Библиотечный класс TreeSet

Как и для списков, в Java реализован библиотечный класс работы с упорядоченными множествами TreeSet.

Он получил такое название потому, что реализован при помощи сбалансированного двоичного дерева. Для пользователя — это отсортированное множество, в нем нет повторения элементов и все объекты хранятся в отсортированном виде по возрастанию.

Класс TreeSet является наследником класса AbstractSet и реализует интерфейс NavigableSet.

В классе TreeSet определены конструкторы:

- TreeSet() — создает пустое дерево;
- TreeSet(Collection <? extends E> c) — создает дерево, в которое добавляют все элементы коллекции c;
- TreeSet(SortedSet <E> set) — создает дерево, в которое добавляют все элементы сортированного набора set;
- TreeSet(Comparator <? super E> comparator) — создает пустое дерево, где все добавляемые элементы впоследствии будут отсортированы компаратором.

Класс TreeSet обладает следующими методами (табл. 4.18).

Метод	Описание
E ceiling(E obj)	Ищет в наборе наименьший элемент, для которого истинно $e \geq$ объект. Если такой элемент найден, он возвращается. В противном случае возвращается значение null
E floor(E obj)	Ищет в наборе наибольший элемент, для которого истинно $e \leq$ объект. Если такой элемент найден, он возвращается. В противном случае возвращается значение null
add(E obj)	Добавляет элемент e в коллекцию, если такого еще там нет. Возвращает true, если элемент добавлен
remove(E obj)	Удаляет указанный элемент из коллекции
E contains(E obj)	Возвращает true, если указанный элемент имеется в коллекции
E subSet(E нижнГраница, E верхнГраница)	Возвращает объект интерфейса NavigableSet, включающий все элементы вызывающего набора, которые больше нижнГраница и меньше верхнГраница
size()	Возвращает количество элементов коллекции
clear()	Удаляет все элементы коллекции

Табл. 4.18.

### Упражнение 4.7.2

Создадим Java-приложение, демонстрирующее работу класса TreeSet.

```
import java.util.SortedSet;
import java.util.TreeSet;

/* Пример использования класса TreeSet*/
public class Main {
    public static void main(String[] args) {
        //Создание пустого дерева
        TreeSet<String> tree = new TreeSet<String>();

        //Добавление элементов
        tree.add("abc");
        tree.add("aba");

        //Элементы выводятся в сортированном (лексикографическом) порядке
        //Классы должны быть Comparable
        System.out.println("===== Tree =====");
        for (String s : tree){
            System.out.println(s);
        }
        System.out.println();

        //Удаление элементов
        System.out.println(tree.remove("aaa"));
        //удаление несуществующего элемента возвращает False
        System.out.println(tree.remove("abc"));

        System.out.println("===== Tree =====");
        for (String s : tree) {
            System.out.println(s);
        }
        System.out.println();

        tree.add("zzz");
        tree.add("xyz");
        tree.add("zca");

        //Быстрая проверка наличия элемента
        System.out.println(tree.contains("xyz") + " " + tree.contains("aab"));

        // Выводит наименьший элемент, больший или равный указанного
        System.out.println("ceiling = " + tree.ceiling("zyb"));
        System.out.println(tree.ceiling("zzzz")); //null если нет такого элемента

        //Выводит наибольший, элемент меньший или равный указанного
        System.out.println("floor = " + tree.floor("zyb"));
        System.out.println(tree.floor("a")); //null если нет такого элемента

        System.out.println("===== Subtree =====");
        //Взятие подмножества элементов >= zaa и < zzzz
        SortedSet<String>subtree = tree.subSet("zaa", "zzzz");
        for (String s : subtree){
            System.out.println(s);
        }
    }
}
```

```
        System.out.println();  
    }  
}
```

Результат работы программы выглядит так.

```
===== Tree =====  
aba  
abc  
  
false  
true  
===== Tree =====  
aba  
  
true false  
ceiling = zzz  
null  
floor = zca  
null  
===== Subtree =====  
zca  
zzz
```

## **4.8. Алгоритмы сортировок**

Сайт: IT Академия SAMSUNG

Курс: MDev @ IT Академия Samsung

Книга: 4.8. Алгоритмы сортировок

Напечатано:: Егор Беляев

Дата: Суббота, 18 Апрель 2020, 19:30

# **Оглавление**

- 4.8.1. Введение в сортировку данных
- 4.8.2. Сортировка пузырьком
- 4.8.3. Сортировка вставками
- 4.8.4. Быстрая сортировка
- 4.8.5. Реализация сортировок в библиотечных классах Java

## 4.8.1. Введение в сортировку данных

Допустим требуется собрать матрешку.



В реальности для решения поставленной задачи матрешка должна находиться на ровной контрастной поверхности при хорошем освещении, а у человека, ее собирающего, должны быть глаза и руки. В таком случае решение поставленной задачи достаточно тривиально: необходимо определить наименьшую часть и положить ее в основание, и далее повторить алгоритм с оставшимися половинами. Ключевой предпосылкой является возможность оценки размера частей матрешки за малое время.

Тем не менее, именно эта предпосылка зачастую препятствует созданию универсального алгоритма: оценить размер всех частей матрешки одновременно не видится возможным при значительном их количестве. Требуется сравнивать размер основания и верхней половины для каждой части в отдельности. Идеальное расположение верхних частей видится следующим образом (см. рис. 4.41).



Рис. 4.41.

В данном случае для сборки матрешки требуется каждый раз помещать на основание самую правую половину. Чтобы расположить половинки матрешки таким образом, можно воспользоваться алгоритмами сортировки.

Сортировка данных помогает в реализации эффективного поиска. Прежде всего, имеется в виду бинарный поиск, который успешно работает лишь на отсортированных последовательностях и существенно быстрее последовательного.

Отсортированная последовательность получается не только путем сортировки изначально неупорядоченных данных, но и путем реализации следующих алгоритмов:

- вставки с сохранением порядка. При этом каждый новый элемент вставляется таким образом, чтобы последовательность оставалась отсортированной;
- слияния отсортированных последовательностей. При этом получение нового отсортированного массива, который состоит из элементов двух отсортированных массивов, с наименьшими издержками по времени.

## Определения

Зачастую требуется отсортировать набор данных, которые хранятся в виде списка или массива, причем эти данные могут храниться в примитивных типах или объектах.

В последнем случае, у каждого элемента целесообразно определить ключ сортировки — поле или набор полей, по которым набор должен быть отсортирован. К примеру, есть список, где хранится информация об учениках. Для каждого ученика определен его класс, возраст, пол, фамилия и имя. В данном случае возможна сортировка по различным полям. В случае если требуется сортировка списка фамилии, то ключом сортировки будет поле «фамилия», а если требуется сортировка учеников по классам, в которых они учатся, то ключом сортировки выступает поле «класс». Более того, может быть составной ключ сортировки, например, если требуется сортировка по классам, а внутри каждого класса ученики должны быть упорядочены по фамилии. В таком случае ключом сортировки будет пара полей «класс, фамилия».

Далее введем определения для понятий, задействованных в процессе сортировки.

**Алгоритм сортировки** — это алгоритм для упорядочивания некоторого набора элементов (в списке, массиве и пр.) в соответствии с заданным порядком.

**Абсолютный порядок** — правило получения результата сравнения двух элементов.

**Натуральный порядок** — абсолютный порядок, основанный на традиционном соотношении «больше-меньше» между элементами (например, множество целых чисел).

**Ключ сортировки** — поле (либо набор полей) элемента, которое является критерием порядка при сортировке элементов.

**«O» большое** — математическое обозначение, характеризующее трудоемкость алгоритмов.

В применении к алгоритмам сортировки выражение «трудоемкость алгоритма составляет  $O(f(n))$ » означает, что с ростом количества сортируемых элементов  $n$ , время работы алгоритма сортировки будет расти не быстрее, чем некоторая константа, умноженная на  $f(n)$ . Так как умножение на некоторую константу уже предусмотрено, то, если  $f(n)$  имеет вид  $2n$ , то умножение на двойку опускается и вместо  $O(2n)$  подразумевается  $O(n)$ .

## Оценка эффективности алгоритма сортировки

Как правило, алгоритмы сортировки оценивают по двум основным критериям: времени выполнения и использования памяти.

Первый критерий является основным и характеризует быстродействие алгоритма. Данный критерий также называется вычислительной сложностью. Для алгоритмов сортировки важны наихудшее, среднее и наилучшее поведение алгоритма в зависимости от количества сравниваемых элементов  $n$ . Хороший алгоритм сортировки в общем случае работает за  $O(n * \log n)$ , плохой — за  $O(n^2)$ . Наилучшее поведение для алгоритма сортировки —  $O(n)$ . Время работы алгоритма зачастую измеряют в количестве операций сравнения, которые необходимы для его реализации.

Второй критерий эффективной сортировки заключается в выделении дополнительной памяти под временное хранение данных. За частую данным алгоритмам необходимо  $O(\log n)$  памяти. В данном случае не учитывается место под временное хранение массива данных.



Алгоритмы сортировки характеризуют следующими свойствами: устойчивость (англ. *stability*) — сортировка не меняет взаимного расположения элементов с одинаковыми ключами. Естественность поведения — эффективность метода при обработке уже

упорядоченных или частично упорядоченных данных. Алгоритм ведет себя естественно, если учитывает эту характеристику входной последовательности и работает лучше.

**Использование операции сравнения.** Алгоритмы, использующие для сортировки сравнение элементов между собой, называются основанными на сравнениях. Минимальная трудоемкость худшего случая для этих алгоритмов составляет  $O(n \cdot \log n)$ , но они отличаются гибкостью применения. Для специальных случаев (типов данных) существуют более эффективные алгоритмы.

**Использование дополнительной памяти.** Многие быстрые сортировки используют дополнительную память для хранения промежуточных результатов. Объем дополнительной памяти зависит от размера входных данных.

**Использование дополнительных знаний.** В особых случаях присутствует дополнительная информация об элементах массива или об их расположении во входных данных. Применение специализированных алгоритмов сортировки может значительно повысить быстродействие, вплоть до  $O(n)$ . Кусочно-сортированный массив является примером такого случая.

Отличная визуализация работы алгоритмов сортировки приведена на youtube-канале известного проекта Algo-rythmics. Можно увидеть сравнение работы пяти наиболее известных алгоритмов сортировки:

- Insertion sort — сортировка вставками;
- Selection sort — сортировка выбором;
- Bubble sort — пузырьковая сортировка;
- Quick sort — быстрая сортировка;
- Shell sort — сортировка Шелла.

Далее рассмотрим три из приведенных выше алгоритмов — сортировку пузырьком, сортировку вставками и быструю сортировку.

## 4.8.2. Сортировка пузырьком

Алгоритм пузырьковой сортировки считается учебным и практически не применяется на практике. Тем не менее, он представляется весьма полезным для формирования алгоритмического мышления. Соседние элементы массива сравниваются между собой и, в случае необходимости, меняются местами. В итоге больший элемент в результате первой итерации будет последним в массиве, и в последующих итерациях уже не производится его сравнение с остальными элементами массива. Таким образом будет  $n-1$  сравнений. Далее по аналогии второй по величине элемент помещается на предпоследнее место и так далее. В конце алгоритма первым становится элемент с наименьшим числовым значением, а последним — с наибольшим. В итоге наибольшие элементы поэтапно перемещаются к концу массива («всплывают» до нужной позиции, как пузырек в воде).

Ниже приведена одна из возможных реализаций алгоритма на языке Java.

```
public static void bubbleSort(int[] a){  
    for(int i = a.length-1 ; i > 0 ; i--){  
        for(int j = 0 ; j < i ; j++){  
            if( a[j] > a[j+1] ){  
                int tmp = a[j];  
                a[j] = a[j+1];  
                a[j+1] = tmp;  
            }  
        }  
    }  
}
```

### 4.8.3. Сортировка вставками

Сортировка вставками, так же как и пузырьковая сортировка, как правило, используется в учебном программировании. Она проста в реализации и достаточно эффективна на частично упорядоченных последовательностях.

Существует множество модификаций алгоритма сортировки вставками. Разберем один из более распространенных вариантов. Его алгоритм заключается в следующем.

1. Поэтапно перебираем все элементы массива, начиная со второго. Данный элемент назовем текущим. Переходим ко второму шагу. В случае, если элементы массива закончились, конец алгоритма.
2. Все элементы до текущего (слева) уже являются отсортированной последовательностью. Выбираем текущий элемент и поэтапно сравниваем с элементами, стоящими слева. В случае, если элемент превышает текущий, меняем их местами. Процедуру повторяем до тех пор, пока не найдем позицию, где слева от текущего будет меньший элемент. В итоге получаем отсортированную последовательность на один элемент больше.
3. Переходим к первому шагу.

Приведенный ниже код реализует описанный выше алгоритм.

```
public static void insertionSort(int[] a) {  
    for (int i = 1; i < a.length; i++) {  
        int currElem = a[i];  
        int prevKey = i - 1;  
        while (prevKey >= 0 && a[prevKey] > currElem) {  
            a[prevKey + 1] = a[prevKey];  
            prevKey--;  
        }  
        a[prevKey+1] = currElem;  
    }  
}
```

Как и в случае пузырьковой сортировки, сложность алгоритма при худшем варианте входных данных составляет  $O(n^2)$ . Алгоритм возможно ускорить с помощью использования бинарного поиска для нахождения позиции текущего элемента в уже отсортированной части.

## 4.8.4. Быстрая сортировка

Закончим далеко не полный обзор существующих алгоритмов сортировки быстрой сортировкой. Данный способ полностью оправдывает свое название. Он является одним из самых эффективных и основан на стратегии «разделяй и властвуй». Этапы алгоритма заключаются в следующем.

1. Выбор опорного элемента. В целом данный выбор не оказывает влияние на корректность алгоритма и без дополнительных сведений о сортируемой информации невозможно оценить, какой выбор лучше. Поэтому используют различные способы: выбор среднего или первого по положению; выбор элемента со случайным индексом, выбор среднего арифметического между максимальным и минимальным элементами последовательности и т. п.
2. Разделение массива. На данном этапе происходит реорганизация последовательности так, чтобы все элементы по величине меньшие или равные опорному элементу стали находиться слева от него, а большие элементы — справа.
3. Для каждого полученного куска последовательности рекурсивно повторяем шаги 1–2 до тех пор, пока не получим последовательность, которая состоит из одного или двух элементов.

Обработка гарантированно завершится. На каждом следующем уровне рекурсии длина обрабатываемой части массива сокращается, как минимум на единицу.

Ниже приведена одна из возможных реализаций алгоритма. Некоторые реализации алгоритма можно посмотреть здесь.

```

public static void quickSort(int[] a, int low, int high) {
    if (a.length == 0)
        return; // завершение выполнение алгоритма, если длина массива равна 0

    if (low >= high)
        return; // завершение выполнение алгоритма, если уже нечего делить

    // выбор опорного элемента
    int middle = low + (high - low) / 2;
    int op = a[middle];

    // разделить на подмассивы, который больше и меньше опорного элемента
    int i = low, j = high;
    while (i <= j) {
        while (a[i] < op) {
            i++;
        }
        while (a[j] > op) {
            j--;
        }
    }

    if (i <= j) { // меняем элементы местами
        int temp = a[i];
        a[i] = a[j];
        a[j] = temp;
        i++;
        j--;
    }
}

// вызов рекурсии для сортировки левой и правой части
if (low < j)
    quickSort(a, low, j);

if (high > i)
    quickSort(a, i, high);
}

```

Общая сложность алгоритма определяется лишь количеством разделений исходной последовательности, то есть глубиной рекурсии. Глубина рекурсии, в свою очередь, зависит от сочетания входных данных и метода определения опорного элемента. Не будем углубляться в вычисления и приведем итоговые оценки:

- в наилучшем случае, когда последовательность хорошо перемешана и каждое деление дает примерно одинаковые по размеру отрезки средняя сложность составит  $O(n \log_2 n)$ ;
- в худшем случае при самом несбалансированном варианте общее время работы составит  $O(n^2)$ . Для больших значений  $n$  худший случай может привести к исчерпанию памяти во время работы программы (переполнению стека вызовов функций из-за рекурсии).

Помимо разобранных нами трех алгоритмов сортировки известно множество других, причем у каждого алгоритма еще могут быть свои модификации.

## 4.8.5. Реализация сортировок в библиотечных классах Java

В решении практических задач на языке Java зачастую нет необходимости разрабатывать собственные алгоритмы сортировки. Для этого достаточно использовать стандартные методы, которыми располагает стандартная библиотека Java.

### Сортировка методами классов Arrays и Collections

В таблице 4.19 приведены методы сортировки класса Arrays.

Операция	Описание метода
static void sort(int[] a)	Метод, сортирующий массив a в натуральном порядке
static void sort(int[] a, int fromIndex, int toIndex)	Метод, сортирующий часть массива a от fromIndex до toIndex в натуральном порядке
static <T> void sort(T[] a, Comparator<? super T> c)	Метод, сортирующий массив a в порядке, заданном объектом с интерфейсом Comparator
static <T> void sort(T[] a, int fromIndex, int toIndex, Comparator<? super T> c)	Метод, сортирующий часть массива a от fromIndex до toIndex в порядке, заданном объектом с интерфейсом Comparator

Табл. 4.19.

Первые два метода в таблице перегружены для всех простых типов и класса Object. Необходимо помнить, что требование реализации интерфейса Comparable в явном виде не указано в сигнатуре метода static void sort(Object[] a).

Далее в таблице 4.20 приведены методы сортировки класса Collections.

Метод	Описание метода
static <T extends Comparable<? super T>> void sort(List<T> list)	Метод, сортирующий список list в натуральном порядке
static <T> void sort(List<T> list, Comparator<? super T> c)	Метод, сортирующий список list в порядке, заданном объектом с интерфейсом Comparator

Табл. 4.20.

Все приведенные методы сортировок устойчивы, но их алгоритмы не закреплены в стандарте Java. Реализация методов сортировки производится разработчиками JVM и отличается в зависимости от платформы. Например, в JVM для Java 8 метод sort класса Arrays реализован на основе Dual-Pivot Quicksort алгоритма. Это модификация классического алгоритма Quicksort, которая на многих данных показывает результат в  $O(n \cdot \log_2 n)$ , где обычный Quicksort выдает  $O(n^2)$ .



Начиная со стандарта Java 8, в классе Arrays появился метод `parallelSort()` для более быстрой сортировки больших объемов информации. Следует подчеркнуть, что если размер входных данных будет недостаточно большим, то сортировка производится в одном потоке, что связано с накладными расходами для создания новых потоков.

## Сортировка простых типов с помощью методов класса Arrays и Collections

В случае примитивных типов понятие натурального порядка вполне понятно, поэтому в Java методы `sort` полностью готовы и их можно использовать в контейнерах простых типов без дополнительной подготовки. Ниже приведен пример сортировки массива простых типов через метод `sort()` класса Arrays.

```
import java.util.Arrays;

public class ArraySortTest {

    public static void main(String[] args) {
        int[] array = { 15, 42, 4, 23, 16, 8 };
        System.out.println(Arrays.toString(array));
        Arrays.sort(array);
        System.out.println(Arrays.toString(array));
    }
}
```

В результате на экране появится:

```
[15, 42, 4, 23, 16, 8]
[4, 8, 15, 16, 23, 42]
```

Далее приведем пример сортировки списка простых типов через метод `Collections.sort()`.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class ListSortTest {

    public static void main(String[] args) {
        List<Integer> list = new ArrayList<Integer>();
        for (int c : new int[] { 15, 42, 4, 23, 16, 8 })
            list.add(c);
        System.out.println(list.toString());
        Collections.sort(list);
        System.out.println(list.toString());
    }
}
```

В результате работы программы будет выведено:

```
[15, 42, 4, 23, 16, 8]
[4, 8, 15, 16, 23, 42]
```

Следует еще раз подчеркнуть, что для хранения простых типов в коллекциях требуется использовать их классы-оболочки.

## Сортировка объектов

В случае объектов классов невозможно заранее определить, что следует понимать под натуральным порядком. Например, если у нас есть два объекта-прямоугольника, то их можно сравнивать как по периметру, так и по площади. Для сортировки объектных типов в Java предусмотрен специальный интерфейс Comparable. Он включает в себя всего один метод compareTo(T o). Метод возвращает тип int — результат сравнения текущего объекта с объектом o. При этом возвращаемое значение является:

- отрицательным целым числом, если this < o;
- нулем, если this = o;
- положительным целым числом, если this > o.

Например, определение натурального порядка для объектов класса Rectangle может выглядеть так:

```
class Rectangle implements Comparable<Rectangle> {

    int w, h;

    public Rectangle(int w, int h) {
        this.w = w;
        this.h = h;
    }

    public int area() {
        return w * h;
    }

    @Override
    public int compareTo(Rectangle arg0) {
        return this.area() - arg0.area();
    }
}
```

Теперь если создать массив или коллекцию из объектов типа Rectangle, его можно сортировать методами sort класса Arrays или Collections соответственно.

```
Rectangle [] rectangles = {new Rectangle(10,15), new Rectangle(10,20), new Rectangle(1,2),
new Rectangle(2,7)};
System.out.println("Массив до сортировки:");
for(Rectangle r: rectangles){
    System.out.println(r.h + " " + r.w);
}
Arrays.sort(rectangles);
System.out.println("Массив после сортировки:");
for(Rectangle r: rectangles){
    System.out.println(r.h + " " + r.w);
}
```

В результате работы программы появится следующее:

Массив до сортировки:

```
15 10  
20 10  
2 1  
7 2
```

Массив после сортировки:

```
2 1  
7 2  
15 10  
20 10
```

Как видно, треугольники упорядочены по возрастанию площади. Если реализовать функцию compareTo следующим образом, то прямоугольники будут сравниваться по периметру:

```
@Override  
public int compareTo(Rectangle arg0) {  
    return 2 * this.w * this.h - 2* arg0.w * arg0.h;  
}
```

Из примера видно, что контейнеры объектов не имеют особых различий от контейнеров примитивных типов с точки зрения применения методов sort.



Обратите внимание, несмотря на то что метод equals не входит в Comparable, его реализация является рекомендуемой. Если методы equals и compareTo будут вести себя несогласованно, то могут появиться неожиданные неприятности с хранением данных в коллекциях (SortedSet, SortedMap и др.). В класс Rectangle необходимо включить метод equals():

```
@Override  
public boolean equals(Object other) {  
    boolean result = false;  
    if (other instanceof Rectangle) {  
        Rectangle that = (Rectangle) other;  
        result = (this.w == that.w && this.h == that.h);  
    }  
    return result;  
}
```

## Интерфейс Comparator

Существует еще один способ сортировки объектов, который подразумевает более одного способа упорядочивания. Для этого можно реализовать несколько различных компараторов («сравнителей»), каждый из которых определял бы необходимый порядок, и использовать в каждом случае свой компаратор. Основной метод интерфейса Comparator определен следующим образом:

```
public interface Comparator<T> {  
    int compare(T o1, T o2);  
}
```

Метод compare() сравнивает два объекта o1 и o2 и, аналогично методу compareTo интерфейса Comparable, и возвращает отрицательное число, если o1 меньше, чем o2, ноль, если o1 равен o2, и положительное число, если o1 больше o2. Далее в примере подробнее рассмотрим работу данного интерфейса.

## Пример 4.22

Рассмотрим простой класс, который хранит данные о человеке:

```
/**  
 * Класс, описывающий конкретного человека. Не позволяет менять содержимое полей после создания.  
 */  
  
public class Person {  
    private final String name;  
    private final String surname;  
    private final int age;  
  
    /**  
     * @return возраст этого человека  
     */  
    public int getAge() {  
        return age;  
    }  
  
    /**  
     * @return имя этого человека  
     */  
    public String getName() {  
        return name;  
    }  
  
    /**  
     * @return фамилия этого человека  
     */  
    public String getSurname() {  
        return surname;  
    }  
  
    @Override  
    public String toString() {  
        return "Person{" +  
            "age=" + age +  
            ", name='" + name + '\'' +  
            ", surname='" + surname + '\'' +  
            '}';  
    }  
  
    /**  
     * Позволяет создать нового человека  
     * @param age возраст  
     * @param name имя  
     * @param surname фамилия  
     */  
    public Person(int age, String name, String surname) {  
        this.age = age;  
        this.name = name;  
        this.surname = surname;  
    }  
}
```

Предположим, в программе, в зависимости от выбора пользователя, нужно упорядочивать людей или по именам, или по фамилиям и именам (иначе говоря, все люди упорядочены по фамилиям, а однофамильцы между собой — по именам), или по возрасту, фамилиям и именам (иначе говоря, все упорядочены по возрасту, те, у кого возраст одинаков, упорядочены между собой по фамилиям, а однофамильцы, у которых возраст одинаков, упорядочены между собой по именам). Это может быть реализовано следующим образом (описание полей, геттеров, конструктора и `toString` опущено, так как они были приведены выше):

```
public class Person {

    /**
     * Экземпляр компаратора, который сравнивает двух людей по именам
     */
    public static final Comparator<Person> NAME_COMPARATOR =
        new Comparator<Person>() {
            @Override
            public int compare(Person o1, Person o2) {
                // возвращаем результат сравнения имен
                return o1.getName().compareTo(o2.getName());
            }
        };

    /**
     * Экземпляр компаратора, который сравнивает двух людей по фамилиям,
     * а в случае равенства фамилий - по именам
     */
    public static final Comparator<Person> SURNAME_NAME_COMPARATOR =
        new Comparator<Person>() {
            @Override
            public int compare(Person o1, Person o2) {
                // сравним фамилии
                int result = o1.getSurname().compareTo(o2.getSurname());
                // если фамилии не одинаковы - вернем результат сравнения
                if (result != 0)
                    return result;
                // для одинаковых фамилий, результат сравнения - сравнение имен
                return o1.getName().compareTo(o2.getName());
            }
        };

    /**
     * Экземпляр компаратора, который сравнивает двух людей по возрасту,
     * в случае равенства возрастов - по фамилиям, а в случае равенства
     * фамилий - по именам
     */
    public static final Comparator<Person> AGE_SURNAME_NAME_COMPARATOR =
        new Comparator<Person>() {
            @Override
            public int compare(Person o1, Person o2) {
                // сравним возраст
                int result = Integer.compare(o1.getAge(), o2.getAge());
                // если возраст не одинаков - вернем результат сравнения
                if (result != 0)
                    return result;

                // сравним фамилии
                result = o1.getSurname().compareTo(o2.getSurname());
                // если фамилии не одинаковы - вернем результат сравнения
                if (result != 0)
                    return result;
            }
        };
}
```

```

        // для одинаковых возрастов и фамилий, результат
        // сравнения - сравнение имен
        return o1.getName().compareTo(o2.getName());
    }
};

}
}

```

В этом коде определены три компаратора, которые доступны через статические поля класса Person. Они позволяют упорядочить список людей в том виде, в котором необходимо.

Приведем пример использования этих компараторов:

```

public static void main(String[] args) {
    List<Person> people = new ArrayList<Person>();
    people.add(new Person(22, "Eikichi", "Onizuka"));
    people.add(new Person(72, "Edsger", "Dijkstra"));
    people.add(new Person(41, "Alan", "Turing"));
    people.add(new Person(41, "Sergey", "Brin"));
    people.add(new Person(41, "Impersonator", "Brin"));
    people.add(new Person(28, "Dmitry", "Karamazov"));
    people.add(new Person(23, "Ivan", "Karamazov"));
    people.add(new Person(16, "Alex", "Karamazov"));

    // отсортируем людей по именам
    Collections.sort(people, Person.NAME_COMPARATOR);
    // выведем их
    System.out.println("By name:");
    for (Person p: people)
        System.out.println(p);
    System.out.println();

    // отсортируем людей по фамилиям, а затем по именам
    Collections.sort(people, Person.SURNAME_NAME_COMPARATOR);
    // выведем их
    System.out.println("By surname and then by name:");
    for (Person p: people)
        System.out.println(p);
    System.out.println();

    // отсортируем людей по возрасту, а затем по фамилиям, а затем по именам
    Collections.sort(people, Person.AGE_SURNAME_NAME_COMPARATOR);
    // выведем их
    System.out.println("By age and then by surname and then by name:");
    for (Person p: people)
        System.out.println(p);
}

```

Результат работы этого примера следующий:

By name:

```
Person{age=41, name='Alan', surname='Turing'}
Person{age=16, name='Alex', surname='Karamazov'}
Person{age=28, name='Dmitry', surname='Karamazov'}
Person{age=72, name='Edsger', surname='Dijkstra'}
Person{age=22, name='Eikichi', surname='Onizuka'}
Person{age=41, name='Impersonator', surname='Brin'}
Person{age=23, name='Ivan', surname='Karamazov'}
Person{age=41, name='Sergey', surname='Brin'}
```

By surname and then by name:

```
Person{age=41, name='Impersonator', surname='Brin'}
Person{age=41, name='Sergey', surname='Brin'}
Person{age=72, name='Edsger', surname='Dijkstra'}
Person{age=16, name='Alex', surname='Karamazov'}
Person{age=28, name='Dmitry', surname='Karamazov'}
Person{age=23, name='Ivan', surname='Karamazov'}
Person{age=22, name='Eikichi', surname='Onizuka'}
Person{age=41, name='Alan', surname='Turing'}
By age and then by surname and then by name:
Person{age=16, name='Alex', surname='Karamazov'}
Person{age=22, name='Eikichi', surname='Onizuka'}
Person{age=23, name='Ivan', surname='Karamazov'}
Person{age=28, name='Dmitry', surname='Karamazov'}
Person{age=41, name='Impersonator', surname='Brin'}
Person{age=41, name='Sergey', surname='Brin'}
Person{age=41, name='Alan', surname='Turing'}
Person{age=72, name='Edsger', surname='Dijkstra'}
```

Практически никогда нет необходимости создавать больше одного экземпляра каждого компаратора, поэтому классы-компараторы объявлены как приватные статические классы внутри класса Person и их единственные экземпляры доступны через статические переменные.

Стоит отметить, что возможность дать переменной-компаратору имя существенно улучшает понятность кода как для вас самих, так и для других людей, которые будут его читать. Предположим, что вы встретили в чужой программе строку:

```
Collections.sort(people);
```

У вас нет никакой информации о том, как именно будет отсортирован список в данном случае. Хорошо, если вы можете заглянуть в исходные коды класса Person и узнать, как был реализован метод compareTo, и выяснить, какой порядок сравнения был принят за натуральный. Но это требует от вас дополнительных действий каждый раз, когда вы встречаете подобную строку. Если же вы встречаете строку:

```
Collections.sort(people, Person.NAME_COMPARATOR);
```

То вам не нужно производить дополнительных действий, вы сразу видите, что здесь автор подразумевал упорядочивание людей по их именам. Когда вы имеете дело не с объектами типа строк или чисел, для которых есть очевидный всем естественный порядок, часто имеет

смысл создать компаратор просто для того, чтобы дать ему имя и повысить читаемость вашего кода.

Полный код упражнения можно посмотреть [здесь](#).

## **4.9. Множества. Хеширование**

Сайт: IT Академия SAMSUNG

Курс: MDev @ IT Академия Samsung

Книга: 4.9. Множества. Хеширование

Напечатано:: Егор Беляев

Дата: Суббота, 18 Апрель 2020, 19:30

# **Оглавление**

- 4.9.1. Множества
- 4.9.2. Множество целых чисел от 0 до 100
- 4.9.3. Хеширование
- 4.9.4. Хеш-таблица
- 4.9.5. Интерфейс Set. Классы HashSet и TreeSet

## 4.9.1. Множества

Важными структурами данных являются **множества**. Обычно множества не могут хранить в себе равных элементов (множества, позволяющие это, называются **мультимножествами**).

Множества выполняют те же операции, что и другие коллекции: добавление, удаление и поиск элементов.

Важной особенностью множеств является быстрый поиск элемента, то есть получение ответа на запрос «есть ли такой элемент во множестве». При этом операции добавления и удаления элементов из множества выполняется тоже быстро.

В этой теме мы преимущественно будем рассматривать реализацию множеств при помощи хеширования, однако существуют и другие способы. Например, библиотечный Java-класс TreeSet использует в реализации красно-черные деревья.

## 4.9.2. Множество целых чисел от 0 до 100

Реализуем множество, в котором будут храниться целые числа от нуля до ста. Для этого можно взять простой массив boolean на сто один элемент и записывать туда true, если элемент есть во множестве, и false, если нет.

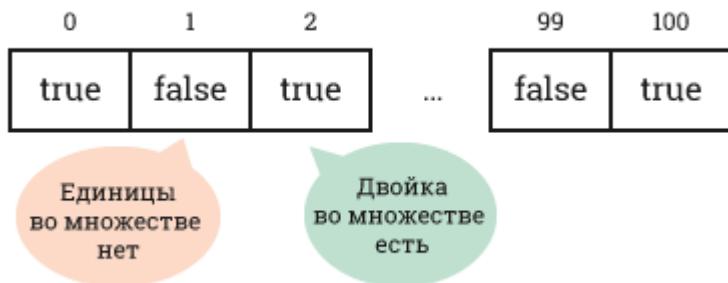


Рис. 4.42.

```
public class Set100int {  
  
    private boolean array[] = new boolean[101];  
  
    public boolean add(int e) {  
        if (e > 100)  
            return false;  
        else {  
            array[e] = true;  
            return true;  
        }  
    }  
  
    public void clear() {  
        Arrays.fill(array, false);  
    }  
  
    public boolean contains(int e) {  
        if (e <= 100)  
            return array[e];  
        else  
            return false;  
    }  
  
    public boolean remove(int e) {  
        if (e > 100)  
            return false;  
        else {  
            array[e] = false;  
            return true;  
        }  
    }  
}
```

Операции `size()` и `clear()` тоже можно реализовать со сложностью  $O(1)$ .

Проблем у этого решения две.

1. Решение не масштабируется на большое количество чисел. В этом случае пришлось бы создавать огромный массив.
2. Оно в принципе не подходит для нецелых чисел, а тем более произвольных объектов.

Обе проблемы решаются при помощи **хеширования**.

### 4.9.3. Хеширование

Хеширование применяют, когда нужно преобразовать некоторую совокупность связанных данных в число (более строго последовательность битов определенной длины). Это преобразование называется хеш-функцией, а результат хеш-кодом. Естественно, с помощью хеш-функций получают не произвольное число, а такое, которое зависит от содержания данных, причем если данные одинаковые, то и число должно получаться равным.

В качестве примера применения хеш-функций для поиска можно привести поиск в словаре.

Поиск слова — процесс долгий: нужно сравнивать каждый символ искомой фразы.

Значительно быстрее сравнивать числа. Если нам известны хеш-коды каждого слова в словаре (их нужно вычислить всего один раз), то можно при поиске нужного слова вычислить его хеш-код и дальше сравнивать числа: при этом будет одно единственное сравнение при проверке каждого слова. Таким образом, хеш-функция значительно ускоряет поиск.

Однако, в этом случае мы можем найти другое слово, если у них окажутся одинаковые хеш-коды. Если хеш-коды совпали, то нужно уже проверить на совпадение символы и в случае несовпадения продолжить поиск.

Такие совпадения называются **коллизиями**.

#### Коллизии

Каким бы не был большим диапазон `int`, понятно, что он не бесконечен и существует вероятность того, что хеш-коды разных объектов могут совпасть. Ситуация, когда у разных объектов одинаковые хеш-коды, называется — коллизией. Вероятность возникновения коллизии зависит от используемого алгоритма генерации хеш-кода.

#### Метод `hashCode()`

Функция хеширования настолько важна, что в Java она реализована в классе `Object`, суперклассе всех остальных.

Все классы наследуют методы класса `Object`, среди которых наиболее известен метод `hashCode()`.

```
String str = "Test string";
int hCode = str.hashCode();
System.out.println(hCode);
```

В результате выполнения программы будет выведено целое число. Это число и есть хеш-код строки. В Java она представлена в виде значения типа `int`. Как следствие, хеш-код ограничен диапазоном значений типа `int`.

Главные свойства хеш-кода:

- если хеш-коды *разные*, то они получены из *разных* входных объектов;
- если *объекты одинаковы*, то их хеш-коды всегда одинаковы, однако обратное неверно.

Таким образом:

- для одного и того же объекта хеш-коды всегда одинаковы (рис. 4.43);

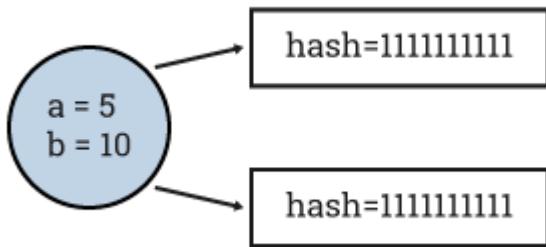


Рис. 4.43.

- для одинаковых и хеши одинаковы (рис. 4.44);

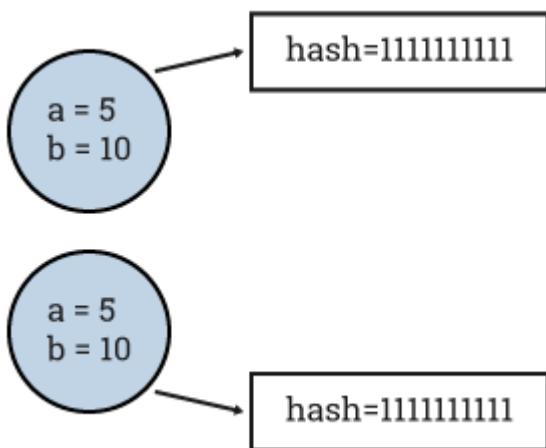


Рис. 4.44.

- если хеш-коды равны, то входные объекты не всегда равны (коллизия) (рис. 4.45);

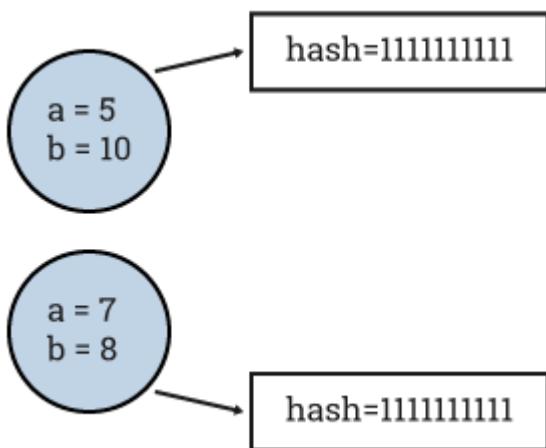


Рис. 4.45.

- если хеш-коды разные, то и объекты разные (рис. 4.46);

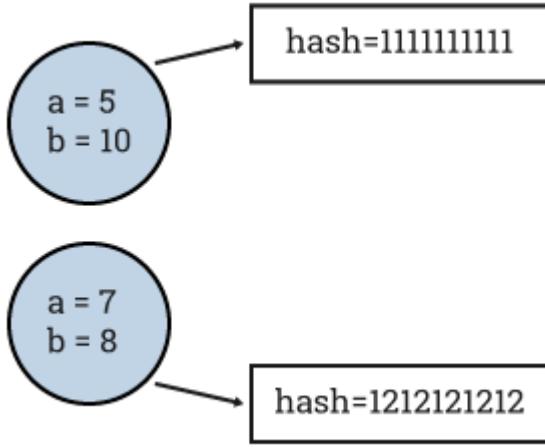


Рис. 4.46.

## Перегрузка метода hashCode()

Реализация метода hashCode() в классе Object, естественно, удовлетворяет требованиям хеш-кодов, но для разных объектов он практически всегда возвращает разные хеш-коды, даже в том случае, если объекты содержат одинаковые данные.

Поэтому, в каждом классе нужно реализовать свою версию хеширования, которая будет основываться на содержании объекта, значениях его полей.

Идеальной хеш-функции не существует. Например, взятие остатка от деления для чисел использовать нехорошо, если мы будем иметь дела только с четными числами: в этом случае половина возможных значений хеш-кодов использоваться не будет, поэтому вероятность появления коллизий сильно возрастет.

Руководящие принципы написания hashCode():

- возвращение одного и того же значения для одного и того же объекта при каждом вызове;
- вычисление хеш-кода должно основываться исключительно на содержимом объекта, но не на его случайной уникальной информации (например, адресе в памяти);
- множество значений, генерируемое функцией hashCode() на множестве объектов контейнера (какой-либо структуры данных), должно быть распределено как можно равномернее, чтобы каждому значению соответствовало примерно равное количество объектов контейнера;
- хорошая хеш-функция должна стремиться сократить вероятность появления коллизий.

Приведем наиболее распространенный алгоритм для получения хеш-кода.

В зависимости от типа каждого поля value вычислите выражение по одному из семи правил (см. табл. 4.21).

№	Тип/значение поля value	Вычисляемое выражение
1	boolean	value ? 0 : 1
2	byte, char, short, int	(int)value
3	long	(int)(value - (value >>> 32))
4	float	Float.floatToIntBits(value)
5	double	(int)Double.doubleToLongBits(value)
6	Ссылка на объект	Вызвать метод hashCode() этого объекта
7	null	0

Табл. 4.21.

Для массивов вычислить по приведенному правилу хеш-код от каждого элемента.

Если в объекте хранится несколько переменных, то объединить полученные значения:

```
//ПСЕВДОКОД
int result = 17;
для всех значений value переменных объекта
    result = 37 * result + hashCode
```

Числа 17 и 37 выбраны потому, что они простые. Можно доказать, что в этом случае получаются хорошие результаты: хеши распределяются равномерно.

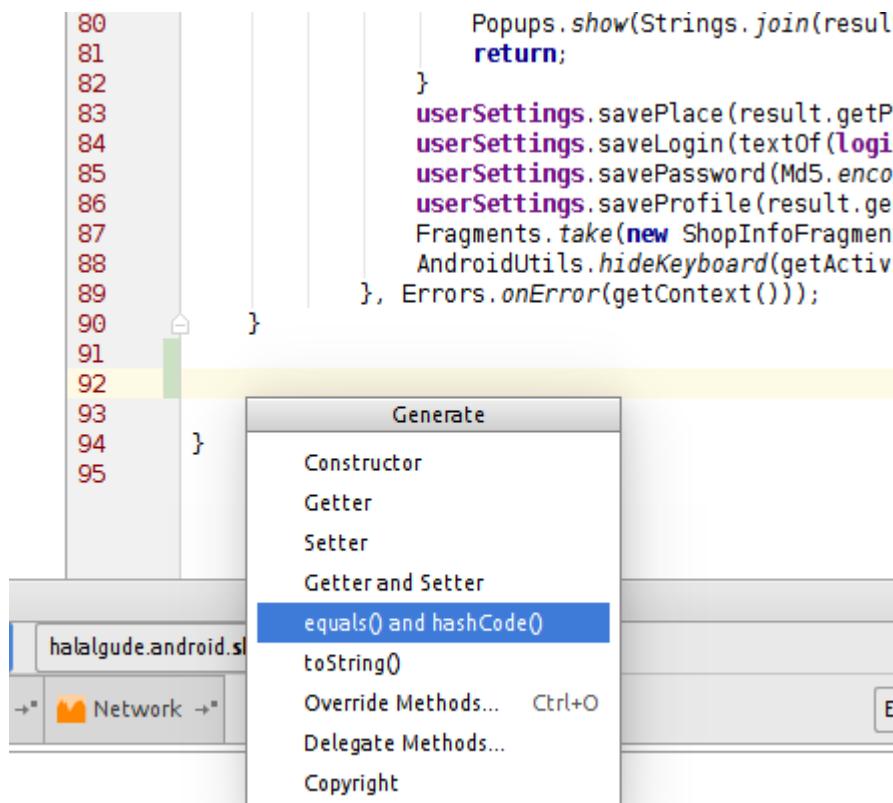


В Eclipse генерацию хеш-кода по этим правилам для своего объекта можно сделать автоматически: для класса в Package Explorer выбрать Source — GenerateHashCode() and equals()...

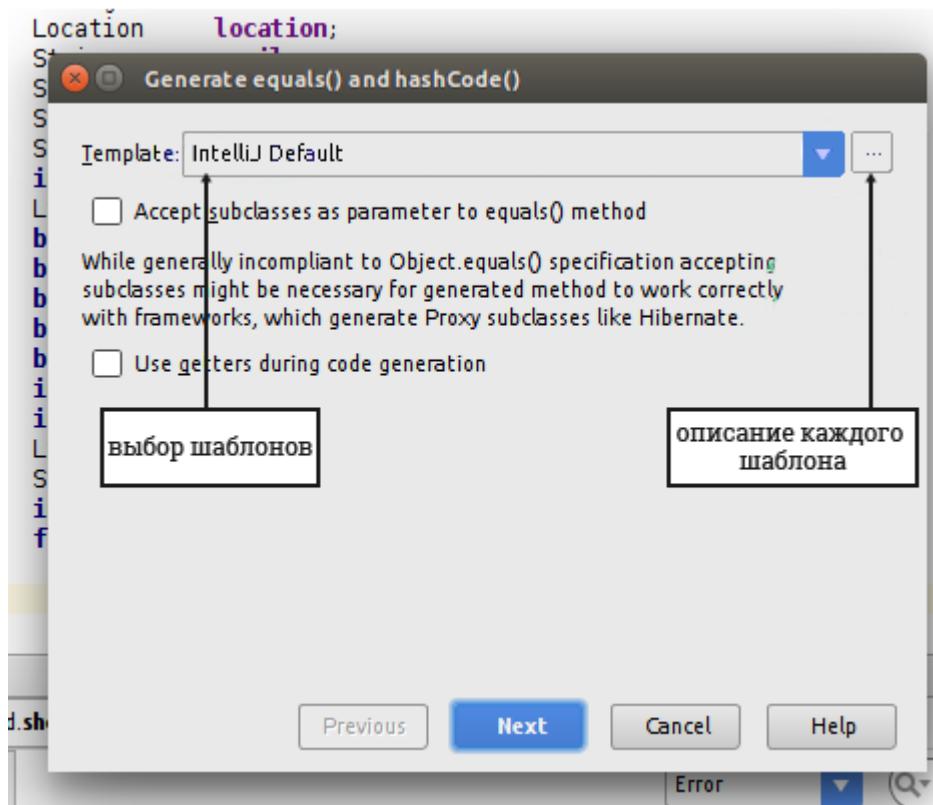


В Android Studio генерацию хеш-кода по этим правилам для своего объекта также можно сделать автоматически: комбинация клавиш Alt+Insert, в открывшемся контекстном меню выбираем equals() and hashCode().

Далее среда предложит шаблоны, по которым можно сгенерировать эти методы, выбор полей, которые будут включены в эти методы.



Далее среда предложит шаблоны, по которым можно сгенерировать эти методы, выбор полей, которые будут включены в эти методы.



Конечно, описанный способ хорошо работает не для всех случаев. Существует достаточно много других алгоритмов для получения хеш-кодов.

Например, для данных строкового типа (наиболее часто хеширование применяется именно для строк) в статье Arash Partow «General Purpose Hash Function Algorithms» приведены восемь вариантов.

#### 4.9.4. Хеш-таблица

От коллизий избавится невозможно. Нужно придумать способ их разрешать, организовать корректную работу в случае их возникновения. Обычно используются два способа: *метод цепочек* и *открытая адресация*.

Рассмотрим первый способ подробно на примере. Для этого модернизируем множество Set100int таким образом, чтобы в нем можно было хранить любое количество произвольных элементов.

Для этого будем хранить в массиве не значения типа boolean, а списки элементов с хеш-кодом равным индексам элементов.

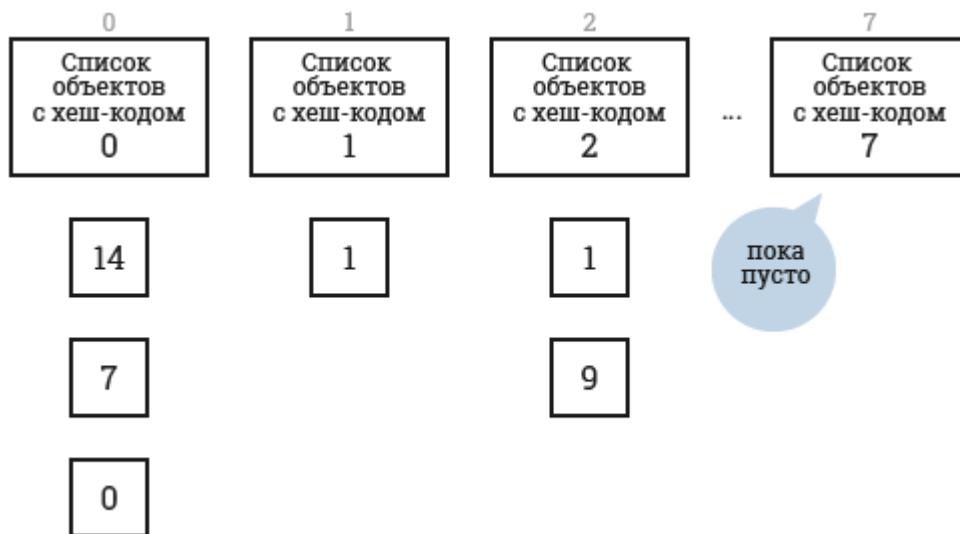


Рис. 4.47.

На рисунке 4.47 пример хеш-таблицы на 8 хеш-индексов. Каждый раз для помещаемых данных вычисляется хеш-код, затем из него вычисляется остаток от деления на 8, то есть получаются значения в диапазоне от 0 до 7 — индекс ячейки массива, куда мы должны поместить значение.

В ячейке массива хранится список (bucket, бакет), в котором и сохраняются все значения, получившие одинаковый индекс. То есть хеш-таблица в этой реализации — это массив списков.

Модернизированный класс может выглядеть так:

```
public class MyHashSet {  
  
    //Если будут частые удаления,  
    //лучше использовать массив LinkedList  
  
    private ArrayList array[];  
    private int capacity;  
  
    // Для инициализации всех Array List  
    // используем конструктор  
    public MyHashSet(int capacity) {  
        this.capacity = capacity;  
        array = new ArrayList[capacity];  
        for (int i = 0; i < capacity; i++)  
            array[i] = new ArrayList();  
    }  
  
    public boolean add(Object e) {  
        // вычисляем хэш-код  
        int hCode = e.hashCode() % capacity;  
        if (array[hCode].contains(e))  
            return false;  
        else {  
            array[hCode].add(e);  
            return true;  
        }  
    }  
  
    public void clear() {  
        for (ArrayList ai : array)  
            ai.clear();  
    }  
  
    public boolean contains(Object e) {  
        // вычисляем хэш-код  
        int hCode = e.hashCode() % capacity;  
        return array[hCode].contains(e);  
    }  
  
    public boolean remove(Object e) {  
        // вычисляем хэш-код  
        int hCode = e.hashCode() % capacity;  
        if (array[hCode].contains(e))  
            return false;  
        else {  
            array[hCode].remove(e);  
            return true;  
        }  
    }  
  
    public String toString()  
}
```

```

{
    StringBuffer res = new StringBuffer();
    for (int i = 0; i < capacity; i++){
        res.append(i + ":" + array[i].toString() + "\n");
    }
    return res.toString();
}

```

Последняя функция позволяет быстро вывести хеш-таблицу.

Добавим 15 элементов в таблицу из 5 списков.

```

int dataSize = 15;
MyHashSet set = new MyHashSet(5);

for (int i = 0; i < dataSize; i++){
    set.add(i);
}

System.out.println(set);

```

Получим:

```

0:[0, 5, 10]
1:[1, 6, 11]
2:[2, 7, 12]
3:[3, 8, 13]
4:[4, 9, 14]

```

Если же будем добавлять значения double, заменим

```
set.add(i);
```

на

```
set.add((double)i);
```

результат будет другим:

```

0:[0.0, 4.0, 10.0]
1:[8.0, 13.0]
2:[3.0, 7.0, 11.0]
3:[1.0, 6.0, 9.0, 14.0]
4:[2.0, 5.0, 12.0]

```

Так происходит потому, что хеш-код для значений double вычисляется по-другому.

Мы видим, что у нас получилось неравномерное распределение значений по бакетам. Очевидно, что в общем случае лучше, если бакетов много и значения по ним раскладываются равномерно, тогда поиск и размещение значений в хеш-таблицу будет очень быстрым. Хорошая хеш-функция та, которая порождает хорошие ключи для распределения элементов по бакетам. Последнее требование минимизирует коллизии и предотвращает случай, когда элементы данных с близкими значениями попадают только в одну часть таблицы.

Конечно, сильно влияет на скорость размер таблицы. Проведите эксперименты.

```
int dataSize = 1000000;
MyHashSet set = new MyHashSet(10000);

long startTime = System.currentTimeMillis();
for (int i = 0; i < dataSize; i++){
    set.add(i);
}
for (int i = 0; i < dataSize; i++){
    set.contains(i);
}
long finishTime = System.currentTimeMillis();
System.out.println("Worktime: " + (finishTime - startTime) + "ms");
```

Попробуйте добавить миллион элементов в таблицу на 10 тысяч списков и на 100 тысяч. Посмотрите на разницу во времени выполнения.

В заключение проведите эксперимент с наивной реализацией множества на базе обычного ArrayList:

```
int dataSize = 100000;
//наивная реализация на базе ArrayList
ArrayList set = new ArrayList();
long startTime = System.currentTimeMillis();
for (int i = 0; i < dataSize; i++){
    //Если элемент есть - не добавляем
    if (!set.contains(i)) set.add(i);
}
for (int i = 0; i < dataSize; i++){
    set.contains((double)i);
}
long finishTime = System.currentTimeMillis();
System.out.println("Worktime: " + (finishTime - startTime) + "ms");
```

Вряд ли вам хватит терпения дождаться окончания работы этого фрагмента, хотя элементов всего 100 тысяч. Здесь работает долго и добавление, и поиск.

Хеш-таблицы часто применяются в базах данных и, особенно, в языковых процессорах типа компиляторов, где они обслуживают таблицы идентификаторов. Для этих случаев хеш-таблица — наилучшая структура данных.

Хеш-таблица обеспечивает константное время (в случае хорошей хеш-функции) выполнения методов add(), contains() и remove() даже для контейнеров с большим количеством элементов.

## 4.9.5. Интерфейс Set. Классы HashSet и TreeSet

Множества в стандартной библиотеке Java реализуют интерфейс Set, который расширяет интерфейс Collection.

Таким образом, множества являются коллекциями, то есть реализуют методы;

- Iterator iterator();
- int size();
- boolean isEmpty();
- boolean contains(Object o);
- boolean add(Object o);
- boolean addAll(Collection c);
- Object[] toArray();
- boolean remove(Object o);
- boolean removeAll(Collection c);
- public boolean retainAll(Collection c) — (retain — сохранить). Выполняет операцию «пересечение множеств»;
- public void clear().

Основными классами, реализующими интерфейс Set, являются HashSet и TreeSet.

HashSet работает быстрее, как следует из названия, он реализован на основе хеш-таблицы. При хранении объектов в их классах должны быть переопределены методы hashCode() и equals(), иначе два логически одинаковых объекта будут считаться разными, так как при добавлении элемента в коллекцию будет вызываться метод hashCode() класса Object, который скорее всего вернет разный хеш-код для этих объектов.

TreeSet реализован на базе красно-черных деревьев, работает несколько медленнее HashSet, но предоставляет расширенные возможности. По сути, он является отсортированным массивом, поэтому в нем есть методы для получения максимального и минимального элементов, итератор обходит TreeSet в порядке возрастания. При использовании TreeSet для хранения объектов нужно в их классах реализовать интерфейс Comparable или передать компаратор.

Рассмотрим эти важные ограничения на примере.

В программе для демонстрации геометрических преобразований нужно хранить одинаковые объекты, например, точки в единственном экземпляре. Для этого будем хранить точки во множествах.

```
class Point{  
    double x;  
    double y;  
    public Point(double x, double y) {  
        this.x = x;  
        this.y = y;  
    }  
    @Override  
    public String toString() {  
        return "Point [x=" + x + ", y=" + y + "]";  
    }  
}
```

Попробуем добавить несколько точек в HashSet и TreeSet.

```
HashSet<Point> hashSet = new HashSet<>();
hashSet.add(new Point(1., 1.));
hashSet.add(new Point(0., 0.));
hashSet.add(new Point(1., 1.));
System.out.println("HashSet: " + hashSet);

TreeSet<Point> treeSet = new TreeSet<>();
treeSet.add(new Point(1., 1.));
treeSet.add(new Point(0., 0.));
treeSet.add(new Point(1., 1.));
System.out.println("TreeSet: " + treeSet);
```

Программа запустится, но выведет:

```
HashSet: [Point [x=1.0, y=1.0], Point [x=1.0, y=1.0], Point [x=0.0, y=0.0]]
Exception in thread "main" java.lang.ClassCastException: Point cannot be cast to java.lang.
Comparable
    at java.util.TreeMap.compare(Unknown Source)
    at java.util.TreeMap.put(Unknown Source)
    at java.util.TreeSet.add(Unknown Source)
...
...
```

В HashSet точка (1.0; 1.0) добавилась дважды.

А TreeSet вообще не смог работать с классом Point.

Это произошло потому, что для корректной работы HashSet нужно добавить в класс Point реализацию методов hashCode() и equals(), а для работы TreeSet реализовать в нем интерфейс Comparable.

Класс Point может быть дополнен методами:

```

class Point implements Comparable<Point> {
    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        long temp;
        temp = Double.doubleToLongBits(x);
        result = prime * result + (int) (temp ^ (temp >>> 32));
        temp = Double.doubleToLongBits(y);
        result = prime * result + (int) (temp ^ (temp >>> 32));
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Point other = (Point) obj;
        if (Double.doubleToLongBits(x) != Double.doubleToLongBits(other.x))
            return false;
        if (Double.doubleToLongBits(y) != Double.doubleToLongBits(other.y))
            return false;
        return true;
    }

    @Override
    public int compareTo(Point o) {
        // по удаленности от центра координат
        return (int)(x * x + y * y - o.x * o.x - o.y * o.y);
    }
    //...
}

```

Методы equals() и hashCode можно быстро сгенерировать в среде разработки.

Метод compareTo() должен возвращать отрицательное, положительное число или ноль в зависимости от того, меньше, больше или равен объект другому объекту. В примере объекты точки сравниваются по расстоянию от центра координат.

Теперь вывод происходит корректно:

```

HashSet: [Point [x=1.0, y=1.0], Point [x=0.0, y=0.0]]
TreeSet: [Point [x=0.0, y=0.0], Point [x=1.0, y=1.0]]

```

В HashSet добавилось всего две точки, как и должно быть, а в TreeSet они плюс к этому расположились в порядке удаленности от центра координат.

У разработчиков возникает вопрос, когда использовать HashSet, а когда TreeSet. Класс HashSet не гарантирует упорядоченности элементов: упорядоченность хеш-кодов в общем случае не гарантирует упорядоченности самих объектов. Поэтому на практике, как правило, выбор

зависит только от того, нужна ли отсортированная последовательность. Если упорядоченность нужна, то используют TreeSet, в остальных случаях — HashSet.

## **4.10. Ассоциативные массивы**

Сайт: IT Академия SAMSUNG

Курс: MDev @ IT Академия Samsung

Книга: 4.10. Ассоциативные массивы

Напечатано:: Егор Беляев

Дата: Суббота, 18 Апрель 2020, 19:31

# **Оглавление**

- 4.10.1. Ассоциативный массив как набор пар «ключ — значение»
- 4.10.2. Интерфейс Map
- 4.10.3. Классы для Map
- 4.10.4. Контейнер HashMap
- 4.10.5. Контейнер TreeMap
- 4.10.6. Синхронизация ассоциативных массивов
- 4.10.7. Хранение данных в Android Preferences

## 4.10.1. Ассоциативный массив как набор пар

### «ключ — значение»

**Ассоциативный массив** — абстрактный тип данных, который предназначен для работы с данными в виде пар: ключ, значение (K,V). Ассоциативные массивы поддерживают операции добавления пары, поиска и удаления пары по ключу.

В ассоциативном массиве не могут храниться пары с одинаковыми ключами.

В паре (K,V) значение V (value) называется значением, ассоциированным с ключом K (key). Семантика и названия вышеупомянутых операций в разных реализациях ассоциативного массива могут отличаться. Некоторые массивы допускают пустые ключи и пустые значения.

Примером ассоциативного массива может служить телефонный справочник. Значением в данном случае является имя абонента, а ключом — номер телефона. Один номер телефона имеет одного владельца, но один человек может иметь несколько номеров. Ассоциативные массивы в профессиональной литературе на английском называют Map (как карточка в картотеке).

В библиотеке Java предусмотрено две основные реализации Map: ассоциативный массив на основе хеш-таблиц HashMap и ассоциативный массив на основе структуры данных дерево TreeMap. Оба класса реализуют интерфейс Map.



В нереляционных базах данных в качестве основной структуры данных выступают многомерные ассоциативные массивы с произвольным набором индексов, также называемые глобалами. Вкратце **глобал** — это постоянный, разреженный, динамический, многомерный массив, содержащий текстовые значения.

## 4.10.2. Интерфейс Map

Интерфейс Map связывает уникальные ключи с значениями. Ключ — это объект, который вы используете для последующего извлечения данных. Задавая ключ и значение, вы можете помещать значения в объект Map. После того как это значение сохранено, вы можете получить его по ключу.

```
interface Map<K, V>
```

В параметре K указывается тип ключей, в V — тип хранимых значений.

Ниже представлены основные методы интерфейса:

- V get(Object k) — возвращает значение, ассоциированное с ключом k. Возвращает значение null, если ключ не найден;
- V put(K k, V v) — помещает элемент в ассоциативный массив, переписывая любое предшествующее значение, ассоциированное с таким ключом. Возвращает null, если ключ ранее не существовал. В противном случае возвращается предыдущее значение, связанное с ключом;
- void clear() — удаляет все пары «ключ — значение» из вызывающего ассоциативного массива;
- boolean containsKey(Object k) — возвращает значение true, если вызывающий ассоциативный массив содержит ключ k. В противном случае возвращает false;
- boolean containsValue(Object v) — возвращает значение true, если вызывающий ассоциативный массив содержит значение v. В противном случае возвращает false;
- boolean equals(Object o) — возвращает значение true, если параметр o — это ассоциативный массив, содержащий одинаковые значения. В противном случае возвращает false;
- int hashCode() — возвращает хеш-код вызывающего ассоциативного массива;
- boolean isEmpty() — возвращает значение true, если вызывающий массив не содержит пар. В противном случае возвращает false;
- void putAll(Map<? extends K, ? extends V> m) — помещает все значения из m в массив;
- V remove(Object k) — удаляет элемент, ключ которого равен k;
- int size() — возвращает количество пар «ключ — значение» в массиве.

Следующие три метода позволяют получить содержимое ассоциативного массива в различных видах: пар «ключ — значение», только ключи или только значения.

1. Set<Map. Entry<K, V>> entrySet() — возвращает Set, содержащий все пары «ключ — значение» ассоциативного массива в виде множества объектов интерфейсного типа Map. Entry. Каждый элемент ассоциативного массива, описываемого интерфейсом Map, имеет интерфейсный тип Map. Entry, который предоставляет три основных метода:
  - getKey() — возвращает ключ элемента;
  - getValue() — возвращает значение элемента;
  - setValue(Object value) — меняет значение элемента.
2. Set<K> keySet() — возвращает Set, содержащий все ключи ассоциативного массива. Каждый элемент множества ключей, которое возвращает метод keySet(), является объектом класса, реализующего интерфейс Set. Например, можно перебрать все ключи массива:

```
Set keys = map.keySet();
for (String key: keys) {
    //действия с ключом
}
```

3. Collection<V> values() — возвращает коллекцию, содержащую все значения ассоциативного массива.

У интерфейса Map есть расширение — интерфейс Sortedmap, который гарантирует, что элементы размещаются в возрастающем порядке значений ключей.

Интерфейс NavigableMap в свою очередь расширяет интерфейс Sortedmap и позволяет искать и извлекать элементы не только по точному совпадению, но и ближайшие к заданному ключу или ключам.

### **4.10.3. Классы для Map**

На сегодняшний момент в Java ассоциативные массивы реализованы в виде следующих основных классов:

- `AbstractMap` — абстрактный класс, реализующий большую часть интерфейса `Map`;
- `EnumMap` — расширяет класс `AbstractMap` для использования с ключами типа `enum`;
- `HashMap` — для использования хеш-таблицы;
- `TreeMap` — для использования дерева;
- `WeakHashMap` — для использования хеш-таблицы со слабыми ключами;
- `LinkedHashMap` — разрешает перебор в порядке вставки;
- `IdentityHashMap` — использует проверку ссылочной эквивалентности при сравнении документов.

Рассмотрим подробнее два класса `HashMap` и `TreeMap`. Первый обеспечивает максимальную скорость выборки, а порядок хранения его элементов не очевиден. Второй — хранит ключи отсортированными по возрастанию.

## 4.10.4. Контейнер HashMap

Класс HashMap использует хеш-таблицу для хранения ассоциативного массива, обеспечивая быстрое время выполнения запросов get() и put() при больших наборах. Ключи и значения в данном случае могут быть любых типов, в том числе и null. При этом все ключи обязательно должны быть уникальны, а значения могут повторяться. Данная реализация не гарантирует порядка элементов.

Общий вид HashMap:

```
// K - это Key (ключ), V - Value (значение)
class HashMap<K, V>
```

Объект класса можно объявить следующим образом:

```
Map<String, Integer> hm = new HashMap<String, Integer>();
// или так
Map<String, String> hashmap = new HashMap<String, String>();
```

Обратите внимание, что следующая строка работать не будет:

```
Map<String, int> hm = new HashMap<String, int>(); //ошибка
```

Дело в том, что примитивные типы в ассоциативном массиве использовать нельзя. Ключ или значение должны быть объектными типами.

По умолчанию при использовании пустого конструктора создается картотека емкостью из 16 ячеек. При необходимости емкость увеличивается автоматически.

Можно указать свои емкость и коэффициент загрузки, используя конструкторы HashMap(capacity) и HashMap(capacity, loadFactor). Максимальная емкость, которую вы сможете установить, равна половине максимального значения int(1073741824).

Добавление элементов происходит при помощи метода put(K key, V value). При добавлении нужно указать ключ и его значение.

```
hashmap.put("Moscow", "Russia");
```

Чтобы узнать размер ассоциативного массива, нужно воспользоваться следующим оператором:

```
hashmap.size();
```

Для проверки ключа и значения на наличие необходимо:

```
hashmap.containsKey("Moscow");
hashmap.containsValue("Russia");
```

Фрагмент следующего кода сначала выбирает все ключи, затем все значения, затем все ключи и значения одновременно:

```
for (String key : hashmap.keySet()) {  
    System.out.println("Key: " + key);  
}  
for (int value : hashmap.values()) {  
    System.out.println("Value: " + value);  
}  
for (Map.Entry entry : hashmap.entrySet()) {  
    System.out.println("Key: " + entry.getKey() + " Value: "  
        + entry.getValue());  
}
```

## 4.10.5. Контейнер TreeMap

Класс TreeMap расширяет класс AbstractMap и реализует интерфейс NavigableMap. Он создает ассоциативный массив, который для хранения элементов применяет дерево.

Объекты хранятся в отсортированном порядке по возрастанию. Время доступа и извлечения элементов достаточно мало, что делает класс TreeMap блестящим выбором для хранения больших объемов отсортированной информации, которая должна быть быстро найдена. Для сортировки используются ключи, а не значения. Общий вид объекта класса схож с объектом класса HashMap.

В классе TreeMap присутствуют следующие конструкторы: TreeMap(), TreeMap(Comparator comp), TreeMap(Map m) и TreeMap(SortedMap sm). Первый конструктор создает ассоциативный массив, в котором все элементы отсортированы в натуральном порядке их ключей. Второй создаст пустой ассоциативный массив, элементы которого будут отсортированы по закону, определенному в передаваемом компараторе. Третий конструктор создаст TreeMap на основе уже имеющегося Map. Четвертый создаст TreeMap на основе уже имеющегося SortedMap, элементы в которой будут отсортированы по закону передаваемой SortedMap.

Для объектов класса TreeMap существует метод SubMap(K fromKey, K toKey), который возвращает часть дерева, начиная с ключа fromKey включительно до ключа toKey исключительно. Прототип метода java.util. TreeMap.subMap() выглядит следующим образом:

```
public SortedMap<K,V> subMap(K fromKey,K toKey)
```

### Пример 4.23

Рассмотрим пример работы двух контейнеров HashMap и TreeMap.

Для начала создадим класс TreeAndHashMapExample. В классе создадим функцию, которая демонстрирует некоторые методы для работы с ассоциативными массивами.

```
public class TreeAndHashMapExample {  
    private static void testMap(Map<String, String> map) {  
    }  
  
    public static void main(String[] args) {  
    }  
}
```

В функции testMap запишем код, который демонстрирует некоторые возможности при работе с ассоциативными массивами:

```

System.out.println(map.getClass());

// Добавление элементов в ассоциативный массив.
map.put("Russia", "Moscow");
map.put("USA", "Washington");
map.put("France", "Paris");

// Проход по всем элементам массива.
System.out.println("All elements:");
for (Map.Entry<String, String> entry : map.entrySet()) {
    System.out.println(entry.getKey() + " -> " + entry.getValue());
}
System.out.println();
// Быстрый поиск элемента по ключу.
System.out.println("Russia:");
System.out.println("Russia -> " + map.get("Russia"));

// Удаление элемента по ключу.
map.remove("Russia");

// Несуществующий элемент обозначается как null.
System.out.println("Russia after remove:");
System.out.println("Russia -> " + map.get("Russia"));
System.out.println();

System.out.println("New map:");
for (Map.Entry<String, String> entry : map.entrySet()) {
    System.out.println(entry.getKey() + " -> " + entry.getValue());
}

System.out.println();
// Возвращаем Россию обратно.
map.put("Russia", "Moscow");

```

Далее в функции main вызовем функцию testMap сначала для HashMap, а затем для TreeMap.

```

testMap(new HashMap<String, String>());
TreeMap<String, String> treeMap = new TreeMap<String, String>();
testMap(treeMap);

```

Следующий фрагмент кода демонстрирует возможность выбора подмножества из TreeMap.

```

// с TreeMap можно брать подмножество по аналогии с treeSet.
SortedMap<String, String> submap = treeMap.tailMap("Germany");
System.out.println("Submap:");
for (Map.Entry<String, String> entry : submap.entrySet()) {
    System.out.println(entry.getKey() + " -> " + entry.getValue());
}

```

В итоге, полностью код класса TreeAndHashMapExample выглядит так:

```
import java.util.*;
import java.util.Map.Entry;

public class TreeAndHashMapExample {
    private static void testMap(Map<String, String> map) {
        System.out.println(map.getClass());

        // Добавление элементов в ассоциативный массив.
        map.put("Russia", "Moscow");
        map.put("USA", "Washington");
        map.put("France", "Paris");

        // Проход по всем элементам массива.
        System.out.println("All elements:");
        for (Map.Entry<String, String> entry : map.entrySet()) {
            System.out.println(entry.getKey() + " -> " + entry.getValue());
        }
        System.out.println();
        // Быстрый поиск элемента по ключу.
        System.out.println("Russia:");
        System.out.println("Russia -> " + map.get("Russia"));

        // Удаление элемента по ключу.
        map.remove("Russia");

        // Несуществующий элемент обозначается как null.
        System.out.println("Russia after remove:");
        System.out.println("Russia -> " + map.get("Russia"));
        System.out.println();

        System.out.println("New map:");
        for (Map.Entry<String, String> entry : map.entrySet()) {
            System.out.println(entry.getKey() + " -> " + entry.getValue());
        }

        System.out.println();
        // Возвращаем Россию обратно.
        map.put("Russia", "Moscow");
    }

    public static void main(String[] args) {
        testMap(new HashMap<String, String>());
        TreeMap<String, String> treeMap = new TreeMap<String, String>();
        testMap(treeMap);

        // с TreeMap можно брать подмножество по аналогии с treeSet.
        SortedMap<String, String> submap = treeMap.tailMap("Germany");
        System.out.println("Submap:");
        for (Map.Entry<String, String> entry : submap.entrySet()) {
            System.out.println(entry.getKey() + " -> " + entry.getValue());
        }
    }
}
```

Результат работы программы:

```
class java.util.Hashtable
All elements:
France -> Paris
USA -> Washington
Russia -> Moscow

Russia:
Russia -> Moscow
Russia after remove:
Russia -> null

New map:
France -> Paris
USA -> Washington

class java.util.TreeMap
All elements:
France -> Paris
Russia -> Moscow
USA -> Washington

Russia:
Russia -> Moscow
Russia after remove:
Russia -> null

New map:
France -> Paris
USA -> Washington

Submap:
Russia -> Moscow
USA -> Washington
```

## 4.10.6. Синхронизация ассоциативных массивов

Если вы обращаетесь к ассоциативному массиву из нескольких потоков, необходимо принять меры, чтобы не повредить информацию в массиве. Это неминуемо произойдет, если, например, один поток будет пытаться включить элемент в хэш-таблицу, а другой — перегенерировать ее.

Вместо того, чтобы реализовать классы наборов данных, обеспечивающих безопасную работу с потоками, разработчики библиотеки предпочли использовать для этого механизм представлений. Например, статический метод synchronizedMap() класса java.util. Collections может преобразовать любой ассоциативный массив в Map с синхронизированными методами доступа.

```
Map synchMap = Collections.synchronizedMap(new HashMap());  
// или  
SortedMap m = Collections.synchronizedSortedMap(new TreeMap(...));
```

Всего вспомогательный класс java.util. Collections обеспечивает следующие синхронизирующие методы:

```
public static Collection synchronizedCollection(Collection c);  
public static List synchronizedList(List list);  
public static Map synchronizedMap(Map m);  
public static Set synchronizedSet(Set s);  
public static SortedMap synchronizedSortedMap(SortedMap m);  
public static SortedSet synchronizedSortedSet(SortedSet s);
```

После синхронизации можно обращаться к объекту Map из различных потоков. Каждый метод должен полностью закончить свою работу перед тем, как другой поток сможет вызвать подобный метод. При разработке необходимо следить, чтобы ни один поток не обращался к структуре данных посредством обычных, не синхронизированных методов.

## 4.10.7. Хранение данных в Android Preferences

Android предоставляет несколько вариантов для вас, чтобы сохранить постоянные данные приложений:

- Shared Preferences — сохранение пар «ключ — значение» для примитивных типов данных;
- внутренняя память — сохранение данных во внутренней памяти устройства;
- внешняя память — сохранение данных на внешней памяти устройства;
- база данных SQLite — сохранение структурированных данных в БД.

Решение по выбору одного из вариантов хранения зависит от конкретных потребностей, таких как, должны ли данные быть доступными только для вашего приложения или нет, сколько места требуется для ваших данных и др.

В данном параграфе рассмотрим способ хранения данных, основанных на хранении ассоциативных массивов — Shared Preferences. Класс SharedPreferences позволяет создавать в приложении именованные ассоциативные массивы типа «ключ — значение», которые могут быть использованы различными компонентами приложения.

Общие настройки поддерживают типы boolean, String, float, long и int, что делает их идеальным средством для быстрого сохранения значений по умолчанию, переменных экземпляра класса, текущего состояния UI или пользовательских настроек.

Чтобы получить экземпляр класса SharedPreferences для получения доступа к настройкам в коде приложения используются три метода:

- getPreferences() — внутри активности, чтобы обратиться к определенному для активности предпочтению;
- getSharedPreferences() — внутри активности, чтобы обратиться к предпочтению на уровне приложения;
- getDefaultSharedPreferences() — из объекта PreferencesManager, чтобы получить общедоступную настройку, предоставляемую Android.

По отношению к введенным данным можно выбрать 3 уровня доступности:

- MODE\_PRIVATE — только это приложение может читать настройки с xml-файла;
- MODE\_WORLD\_READABLE — все приложения могут читать с xml-файла;
- MODE\_WORLD\_WRITEABLE — все приложения могут выполнять запись в xml-файл.
- MODE\_MULTI\_PROCESS — несколько процессов совместно используют один файл *SharedPreferences*.



Все модификаторы, кроме MODE\_PRIVATE, в настоящий момент объявлены deprecated и не рекомендуются к использованию в целях безопасности.

Все эти методы возвращают экземпляр класса SharedPreferences, из которого можно получить соответствующую настройку с помощью ряда методов:

- getBoolean(String key, boolean defValue);
- getFloat(String key, float defValue);
- getInt(String key, int defValue);
- getLong(String key, long defValue);
- getString(String key, String defValue).

Чтобы создать или изменить общие настройки, нужно вызвать метод `getSharedPreferences` в контексте приложения, передав имя общих настроек (имя файла):

```
SharedPreferences sharedPreferences = getSharedPreferences(APP_PREFERENCES, Context.MODE_PRIVATE);
```

## Пример 4.24

Разработаем приложение, в котором покажем пример сохранения строки и целого числа. В приложении будет поля для ввода текста и числа, а также две кнопки — Save и Load. По нажатию на Save мы будем сохранять значения из полей, по нажатию на Load — загружать.

Откроем `activity_main.xml` и создадим поля ввода и кнопки (рис. 4.48).

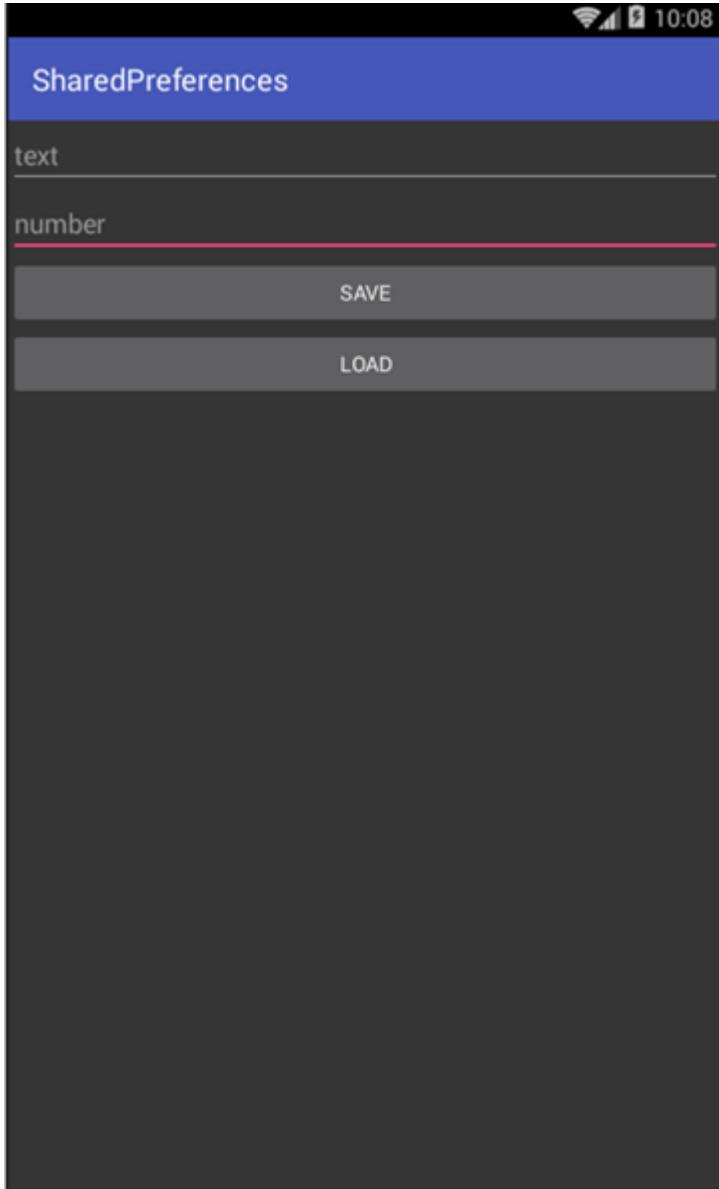


Рис. 4.48.

```
<?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="vertical">

    <EditText
        android:id="@+id/editTextStr"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="text">
    </EditText>

    <EditText
        android:id="@+id/editTextNum"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:ems="10"
        android:hint="number"
        android:inputType="numberDecimal" />

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

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

В *MainActivity.java* пишем следующий код:

```
public class MainActivity extends Activity implements View.OnClickListener {
    EditText editTextStr, editTextNum;
    Button btnSave, btnLoad;
    SharedPreferences sharedpreferences;
    final String SAVED_TEXT = "TEXT";
    final String SAVED_NUM = "NUMBER";

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

        editTextStr = (EditText) findViewById(R.id.editTextStr);
        editTextNum = (EditText) findViewById(R.id.editTextNum);

        btnSave = (Button) findViewById(R.id.save);
        btnSave.setOnClickListener((View.OnClickListener) this);

        btnLoad = (Button) findViewById(R.id.load);
        btnLoad.setOnClickListener((View.OnClickListener) this);
    }

    @Override
    public void onClick(View v) {
        switch (v.getId()) {
            case R.id.save:
                saveData();
                break;
            case R.id.load:
                loadData();
                break;
            default:
                break;
        }
    }

    void saveData() {
        sharedpreferences = getPreferences(MODE_PRIVATE);
        SharedPreferences.Editor editor = sharedpreferences.edit();
        editor.putString(SAVED_TEXT, editTextStr.getText().toString());
        editor.putInt(SAVED_NUM, Integer.parseInt(editTextNum.getText().toString()));
        editor.commit();
        Toast.makeText(this, "Saved", Toast.LENGTH_SHORT).show();
    }

    void loadData() {
        sharedpreferences = getPreferences(MODE_PRIVATE);
        String savedText = sharedpreferences.getString(SAVED_TEXT, "");
        Integer savedNum = sharedpreferences.getInt(SAVED_NUM, 0);
        editTextStr.setText(savedText);
        editTextNum.setText(savedNum.toString());
        Toast.makeText(this, "Loaded", Toast.LENGTH_SHORT).show();
    }
}
```

```
    }  
}
```

После метода `getPreferences()` нужно создать объект `Editor`, который нужен для создания пар «имя — значение», которые будут записаны в xml-файл для сохранения с помощью метода `put()`. Для успешного внесения данных в файл сохранения в конце нужно выполнить команду `commit()`.

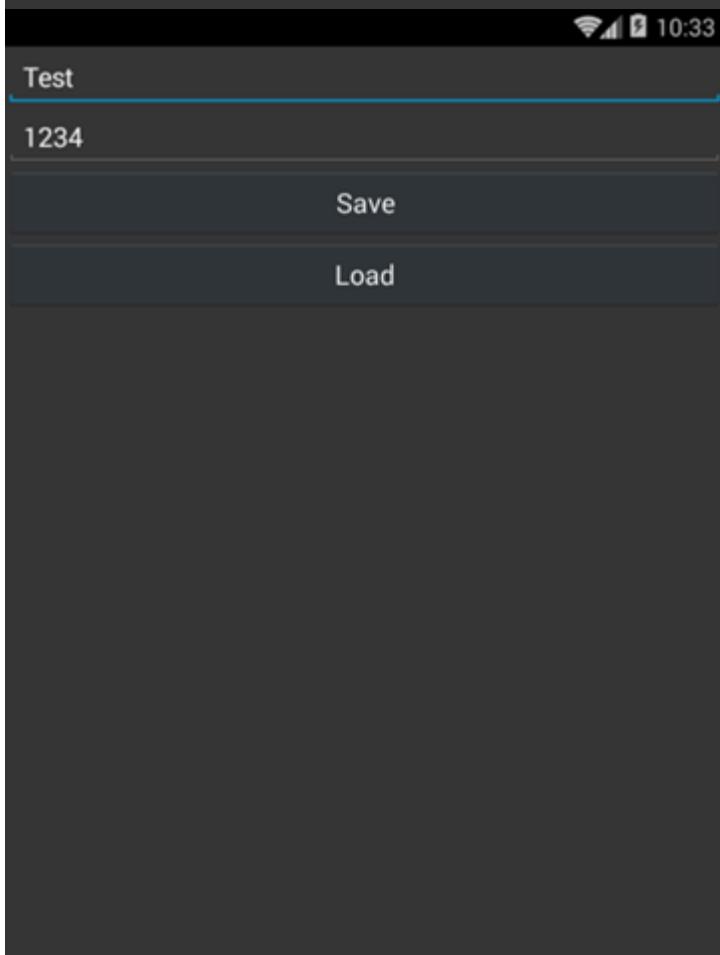
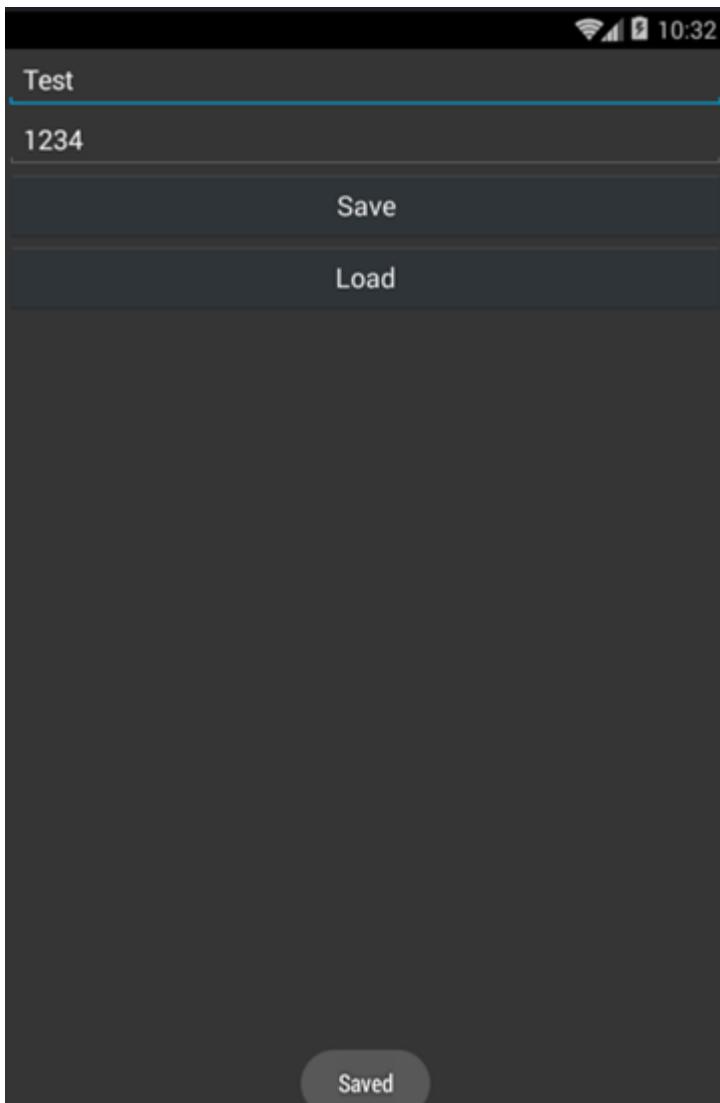
Для того чтобы извлечь сохраненные данные, нужно обратиться к ним с помощью команды `get()`, ссылаясь на необходимые пары-значения.

При сохранении и загрузке сохраненных данных будет высвечиваться `Toast`-сообщение с информацией о выполненной операции.

Убедимся, что будет, если данные не сохранять. Введем любой текст и число в поля ввода и, не нажимая кнопку `Save`, закроем приложение. Затем снова запустим приложение и убедимся, что по нажатию кнопки `Load` ничего не происходит. Введенные нами данные не сохранились и потерялись при новом запуске приложения.

Теперь попробуем сохранить данные. Снова введем значения в поля ввода и нажмем `Save`.

Значения сохранились в системе. Теперь закроем приложение, снова откроем и нажмем `Load`. Значение считалось и отобразилось (рис. 4.49).



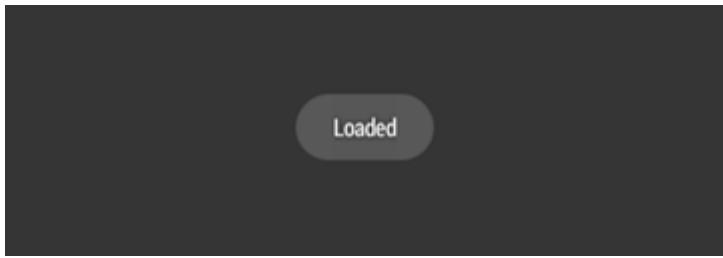


Рис. 4.49.

Чтобы сохранение и загрузка происходили автоматически при закрытии и открытии приложения, метод loadData следует вызывать в onCreate, а метод saveData — в onDestroy.



Preferences-данные сохраняются в файлы, и вы можете посмотреть их. Для этого в Android Studio откройте меню Tools > Android > Android Device Monitor и выберете вкладку File Explorer.

Отобразилась файловая система устройства (или эмулятора).

Если открыть папку `data/data/package_name/shared_prefs`, то мы увидим там файл `MainActivity.xml`. Следует обратить внимание на то, что в пути к файлу используется наш package. Если его открыть, увидим следующее:

```
<?xml version="1.0" encoding="utf-8" standalone="yes" ?>
<map>
    <int name="NUMBER" value="1234" />
    <string name="TEXT">Test</string>
</map>
```

Кроме метода `getPreferences`, который мы использовали, есть метод `getSharedPreferences`. Он выполняет те же действия, но позволяет указать имя файла для хранения данных. Если в методе `saveData` использовать для получения `SharedPreferences` такой код:

```
sharedPreferences = getSharedPreferences("MyPref", MODE_PRIVATE);
```

то данные сохранились бы не в файле `MainActivity.xml`, а в `MyPref.xml`.

Таким образом, мы научились как в Android выполнять сохранение данных приложения с помощью стандартного интерфейса под названием Shared Preferences.

Проект примера можно найти здесь.

## 4.11. Контент-провайдеры в Android

Сайт: IT Академия SAMSUNG  
Курс: MDev @ IT Академия Samsung  
Книга: 4.11. Контент-провайдеры в Android  
Напечатано:: Егор Беляев  
Дата: Суббота, 18 Апрель 2020, 19:31

# **Оглавление**

- 4.11.1. Введение
- 4.11.2. Использование контент-провайдеров
- 4.11.3. Стандартный контент-провайдер ContactsContract
- 4.11.4. Загрузчики (Loaders)

## 4.11.1. Введение

**Контент-провайдер (Content Provider)** — это оболочка, в которую заключены данные. Если приложение использует базу данных SQLite, то только оно имеет к ней доступ. На практике бывают ситуации, когда данные необходимо сделать общими. Например, загрузить фотографию из галереи. Контент-провайдеры — это единственный способ совместного использования данных между приложениями в Android. Каждый контент-провайдер имеет открытый URI (Uniform Resource Identifier — унифицированный (единообразный) идентификатор ресурса), который уникально идентифицирует его набор данных. Все URI для провайдеров начинаются со строки `content://`. URI обычно формируется следующим образом:

```
content://<домен-разработчика-наоборот>.provider.<имя-приложения>/<путь-к-данным>
```

Классы, реализующие контент-провайдеры, чаще всего имеют статическую строковую константу `CONTENT_URI`, которая используется для обращения к данному контент-провайдеру. Вот, например, URI для таблицы телефонных звонков:

```
android.provider.CallLog.Calls.CONTENT_URI
```

Контент-провайдеры являются единственным способом доступа к данным других приложений и используются для получения результатов запросов, обновления, добавления и удаления данных. Если у приложения есть нужные полномочия, оно может запрашивать и модифицировать соответствующие данные, принадлежащие другому приложению, в том числе данные стандартных БД Android. В общем случае, контент-провайдеры следует создавать только тогда, когда требуется предоставить другим приложениям доступ к данным вашего приложения. В остальных случаях рекомендуется использовать СУБД (SQLite). Тем не менее, иногда контент-провайдеры используются внутри одного приложения для поиска и обработки специфических запросов к данным.

В Android есть следующие встроенные провайдеры, которые находятся в пакете `android.provider`:

- `AlarmClock`: управление будильником;
- `Browser`: история браузера и закладки;
- `CalendarContract`: календарь и информация о событиях;
- `CallLog`: информация о звонках;
- `ContactsContract`: контакты;
- `MediaStore`: медиа-файлы;
- `SearchRecentSuggestions`: подсказки по поиску;
- `Settings`: системные настройки;
- `UserDictionary`: словарь слов, которые используются для быстрого набора;
- `VoicemailContract`: записи голосовой почты.

## 4.11.2. Использование контент-провайдеров

Для доступа к данным какого-либо контент-провайдера используется объект класса ContentResolver, который можно получить с помощью метода getContentResolver контекста приложения для связи с поставщиком в качестве клиента:

```
ContentResolver cr = getApplicationContext().getContentResolver();
```

Объект ContentResolver взаимодействует с объектом контент-провайдера, отправляя ему запросы клиента. Контент-провайдер обрабатывает запросы и возвращает результаты обработки.

Контент-провайдеры представляют свои данные потребителям в виде одной или нескольких таблиц подобно таблицам реляционных БД. Каждая строка при этом является отдельным «объектом» со свойствами, указанными в соответствующих именованных полях. Как правило, каждая строка имеет уникальный целочисленный индекс и именем «`_id`», который служит для однозначной идентификации требуемого объекта.

Контент-провайдеры обычно предоставляют минимум два URI для работы с данными: один для запросов, требующих все данные сразу, а другой — для обращения к конкретной «строке». В последнем случае в конце URI добавляется `/<номер-строки>` (который совпадает с индексом `«_id»`).

### Запросы на получение данных

Запросы на получение данных похожи на запросы к БД, при этом используется метод query объекта ContentResolver. Ответ также приходит в виде курсора, «нацеленного» на результирующий набор данных (выбранные строки таблицы):

```
ContentResolver cr = getContentResolver();
// получить данные всех контактов
Cursor c = cr.query(ContactsContract.Contacts.CONTENT_URI, null, null, null, null);

// получить все строки, где третье поле имеет конкретное
// значение и отсортировать по пятому полю
String where = KEY_COL3 + "=" + requiredValue;
String order = KEY_COL5;

Cursor someRows = cr.query(MyProvider.CONTENT_URI, null, where, null, order);
```

### Изменение данных

Для изменения данных применяются методы `insert`, `update` и `delete` объекта ContentResolver. Для массовой вставки также существует метод `bulkInsert`. Пример добавления данных:

```
ContentResolver cr = getContentResolver();

ContentValues newRow = new ContentValues();
// повторяем для каждого поля в строке:
newRow.put(COLUMN_NAME, newValue);
Uri myRowUri = cr.insert(SampleProvider.CONTENT_URI, newRow);

// Массовая вставка:
ContentValues[] valueArray = new ContentValues[5];

// здесь заполняем массив
// делаем вставку
int count = cr.bulkInsert(MyProvider.CONTENT_URI, valueArray);
```

При вставке одного элемента метод `insert` возвращает URI вставленного элемента, а при массовой вставке возвращается количество вставленных элементов.

Пример удаления:

```
ContentResolver cr = getContentResolver();
// удаление конкретной строки
cr.delete(myRowUri, null, null);

// удаление нескольких строк
String where = "_id < 5";
cr.delete(MyProvider.CONTENT_URI, where, null);
```

Пример изменения:

```
ContentValues newValues = new ContentValues();

newValues.put(COLUMN_NAME, newValue);
String where = "_id < 5";

getContentResolver().update(MyProvider.CONTENT_URI, newValues, where, null);
```

## 4.11.3. Стандартный контент-провайдер **ContactsContract**

В Android разработчикам предоставляется возможность получить всю информацию из базы данных контактов для любого приложения с полномочиями READ\_CONTACTS.

Начиная с версии 2.0 Android (API level 5), для хранения и управления контактами на устройстве необходимо использовать провайдер ContactsContract. Это прежде всего связано с тем, что сейчас контакт на мобильном устройстве это не только имя и номер телефона, но и email, Skype, Facebook-аккаунт. И при этом пользователю удобно, когда все эти данные привязаны к определенному человеку. Этот контент-провайдер позволяет управлять контактами в Android через базу данных со всей информацией, касающейся контактов.

ContactsContract работает не с одной строго определенной таблицей, отвечающей за хранение данных, а с трехуровневой моделью, которая позволяет объединять и связывать данные с конкретным человеком, используя следующие таблицы: contacts, raw\_contacts и data.

**Data.** В исходной таблице каждая строка определяет набор личных данных (например, телефонные номера, адреса электронной почты и т. д.). То, какие именно данные находятся в конкретной строке, определяется с помощью типа MIME, указанного для нее.

Универсальные столбцы способны хранить до 15 различных секций с данными разного типа. При добавлении новых данных в таблицу Data нужно указать объект класса RawContacts для каждого связанного набора данных.



MIME (произносится «майм», англ. *Multipurpose Internet Mail Extensions* — многоцелевые расширения интернет-почты) — стандарт, описывающий передачу различных типов данных по электронной почте, а также, в общем случае, спецификация для кодирования информации и форматирования сообщений таким образом, чтобы их можно было пересыпать по интернету.

**RawContacts.** Каждая строка в таблице RawContacts определяет учетную запись, с которой будут ассоциироваться данные из таблицы Data.

**Contacts.** Эта таблица объединяет все строки из RawContacts, которые относятся к одному и тому же человеку. На деле используют таблицу Data для добавления, удаления или изменения данных, относящихся к существующим учетным записям, RawContacts — для создания и управления самими учетными записями, Contacts и Data — для получения доступа к базе данных и извлечения информации о контактах.

Важным является тот факт, что по умолчанию методы активностей и фрагментов работают в главном потоке (Main Thread) и должны, в идеале, заканчиваться моментально, так как «замораживание» главного потока недопустимо. Для выполнения длительных операций должны использоваться вспомогательные/рабочие потоки (Worker Threads). С другой стороны, результатом запроса к контент-провайдеру является объект типа Cursor, жизненный цикл которого обычно не совпадает с жизненным циклом активности или фрагмента, где этот курсор используется. Для «автоматизации» решения обеих задач (выполнения длительных действий и управления курсорами) в Android широко используются загрузчики (Loaders).

## 4.11.4. Загрузчики (Loaders)

Представленные впервые в API 11 загрузчики упрощают асинхронную загрузку данных в активностях и фрагментах. Они обладают следующими характеристиками:

- доступны в любой активности и фрагменте;
- обеспечивают асинхронную загрузку данных;
- отслеживают источники данных и доставляют новые данные при их изменении;
- автоматически переустанавливают соединение с последним используемым объектом класса Cursor, когда заново создаются при изменении конфигурации, таким образом устраняя необходимость выполнять повторные запросы.

При использовании загрузчиков применяются несколько классов и интерфейсов:

- LoaderManager: абстрактный класс, связанный с активностью или фрагментом, используется для управления объектами класса Loader. Он помогает приложению управлять долговременными операциями в привязке к жизненному циклу активности или фрагмента. Наиболее часто LoaderManager применяется с CursorLoader, но может быть реализован для загрузки любых данных. Для каждой активности или фрагмента может существовать только один LoaderManager, который может использовать несколько объектов Loader;
- LoaderManager. LoaderCallbacks: методы, реализующие клиентский интерфейс для взаимодействия с LoaderManager. Например, метод onCreateLoader() применяется для создания нового загрузчика;
- Loader: абстрактный класс, осуществляющий асинхронную загрузку данных, он является базовым классом для загрузчика. Чаще всего используется подкласс CursorLoader;
- AsyncTaskLoader: абстрактный класс, обеспечивающий AsyncTask для выполнения нужных действий;
- CursorLoader: подкласс AsyncTaskLoader, который делает запрос к ContentResolver и возвращает объект класса Cursor. Этот класс создает AsyncTaskLoader для выполнения запросов в фоновом потоке, не блокируя UI приложения. Использование этого загрузчика является наилучшим способом асинхронной загрузки данных из контент-провайдеров.



В настоящее время загрузчики доступны и в API < 11, если используются библиотеки совместимости (библиотеки поддержки, Support Libraries).

В приложении, применяющем загрузчики, обычно используются следующие объекты:

- активность или фрагмент;
- объект класса LoaderManager;
- CursorLoader, связанный с контент-провайдером, либо собственная реализация подклассов Loader или AsyncTask для загрузки данных из других источников;
- реализация LoaderManager. CallBacks, в которой будут создаваться новые загрузчики и отслеживаться ссылки на уже существующие;
- реализация отображения данных, например, с помощью SimpleCursorAdapter;
- источник данных, такой как ContentProvider, если используется CursorLoader.

### Запуск и перезапуск загрузчика

Обычно Loader инициализируется в методе onCreate() активности или методе onActivityCreated() фрагмента. Делается это примерно так:

```

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

    mAdapter = new SimpleCursorAdapter(this, R.layout.item, null, FROM, TO, 0);

    setListAdapter(mAdapter);
    getLoaderManager().initLoader(0, null, this);
}

```

Метод `initLoader()` обеспечивает создание загрузчика и/или его активизацию: если загрузчик с таким идентификатором уже имеется, он будет повторно использован, в противном случае будет создан новый загрузчик (путем вызова метода `onCreateLoader()` интерфейса `LoaderManager.LoaderCallbacks`).

У метода `initLoader()` на примере выше были использованы следующие аргументы:

- идентификатор загрузчика (0);
- дополнительные аргументы, передаваемые загрузчику при создании (null);
- ссылка на объект, реализующий интерфейс `LoaderManager.LoaderCallbacks` (`this`).

Если требуется обязательно создать новый загрузчик (например, потому что данные уже имеющегося загрузчика могут быть устаревшими), вместо `initLoader()` используется метод `restartLoader()`.

## Интерфейс `LoaderManager.LoaderCallbacks`

Данный интерфейс позволяет клиентам взаимодействовать с объектом `LoaderManager` и содержит три метода: `onCreateLoader()` (создает и возвращает новый загрузчик с заданным идентификатором), `onLoadFinished()` (вызывается, когда загрузчик завершает загрузку) и `onLoadReset()` (вызывается, когда загрузчик сбрасывается и его данные больше не используются и их можно освободить).

Ниже показаны примеры реализации этих методов:

```

@Override
public Loader<Cursor> onCreateLoader(int id, Bundle arg){
    return new CursorLoader(this, ContactsContract.CommonDataKinds.Phone.CONTENT_URI, P
ROJECTION, null, null, null);
}

@Override
public void onLoadFinished(Loader<Cursor> loader, Cursor data) {
    mAdapter.swapCursor(data);
}

@Override
public void onLoaderReset(Loader<Cursor> loader) {
    mAdapter.swapCursor(null);
}

```

Как видно из примеров, методы эти весьма просты, но стоит обратить внимание на метод `swapCursor()` используемого адаптера: с помощью этого метода предотвращается дальнейшее использование старых данных, доступных через курсор (если они имелись), но сам курсор

остается открытым — он будет закрыт загрузчиком класса CursorLoader, который его создал (в конструкторе, вызванном в методе onCreateLoader()) и отслеживает его жизненный цикл, в том числе, уничтожая при необходимости.

Дополнительную информацию, включая примеры, в том числе о других типах загрузчиков, можно найти в описаниях классов:

- Loader;
- AsyncTaskLoader;
- LoaderManager.

### Пример 4.25\*

Создадим приложение, которое позволит сохранять контакты в базу данных устройства и выводить список контактов. Создадим новый Android-проект, назовем его ContactsAdd. В первую очередь, отредактируем AndroidManifest.xml, так как для чтения и записи контактов нужно соответствующее разрешение. Добавим строки в этот файл:

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

В результате его содержимое должно выглядеть примерно так:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="ru.samsung.itschool.book.contacts">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
    <uses-permission android:name="android.permission.READ_CONTACTS"/>
    <uses-permission android:name="android.permission.WRITE_CONTACTS"/>
</manifest>
```

В проекте будет одна основная активность — назовем ее MainActivity. Имеет смысл унаследовать ее от стандартного класса FragmentActivity, чтобы можно было использовать загрузчики. Activity будет содержать два поля ввода для имени и телефона, две кнопки Save и Show для записи и отображения контактов, а также ListView, который и будет использован для отображения списка контактов.

Класс активности должен принять примерно следующий вид:

```
public class MainActivity extends FragmentActivity implements LoaderManager.LoaderCallback<Cursor> {
    private ListView listView;
    private Button show, save;
    private EditText name, phone;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        listView = (ListView) findViewById(R.id.contactsList);

        name = (EditText) findViewById(R.id.name);
        phone = (EditText) findViewById(R.id.phone);

        show = (Button) findViewById(R.id.show);
        show.setOnClickListener(new View.OnClickListener() {

            @Override
            public void onClick(View v) {
                //TODO
            }
        });

        save = (Button) findViewById(R.id.save);
        save.setOnClickListener(new View.OnClickListener() {

            @Override
            public void onClick(View v) {
                //TODO
            }
        });
    }
}
```

Xml-файл разметки для активности должен выглядеть примерно следующим образом:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout 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.contactadd.MainActivity">

    <EditText
        android:id="@+id/name"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentEnd="true"
        android:layout_alignParentLeft="true"
        android:layout_alignParentRight="true"
        android:layout_alignParentStart="true"
        android:layout_alignParentTop="true"
        android:ems="10"
        android:hint="Name"
        android:inputType="textPersonName" />

    <EditText
        android:id="@+id/phone"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentEnd="true"
        android:layout_alignParentLeft="true"
        android:layout_alignParentRight="true"
        android:layout_alignParentStart="true"
        android:layout_below="@+id/name"
        android:ems="10"
        android:hint="Phone"
        android:inputType="textPersonName" />

    <Button
        android:id="@+id/save"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Save"
        android:layout_below="@+id/phone"
        android:layout_alignParentLeft="true"
        android:layout_alignParentStart="true"
        android:layout_alignParentRight="true"
        android:layout_alignParentEnd="true" />

    <Button
        android:id="@+id/show"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Show"
        android:layout_below="@+id/save"
        android:layout_alignParentLeft="true"
```

```
        android:layout_alignParentStart="true"
        android:layout_alignParentRight="true"
        android:layout_alignParentEnd="true" />

    <ListView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:id="@+id/contactsList"
        android:layout_below="@+id/show"
        android:layout_alignParentLeft="true"
        android:layout_alignParentStart="true">
    </ListView>

</RelativeLayout>
```

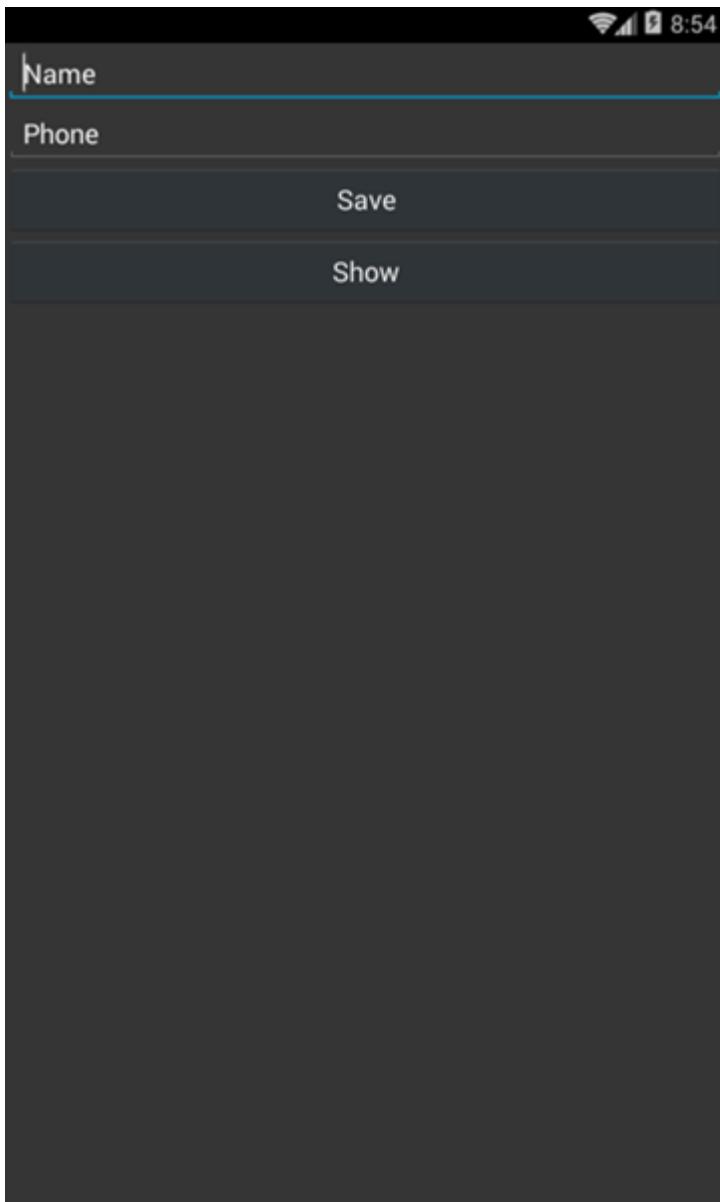


Рис. 4.50.

Чтобы отобразить что-то в ListView, необходим адаптер. В данном случае мы можем использовать SimpleCursorAdapter, который позволяет получать данные из курсора (контент-провайдеры предоставляют доступ к данным именно в виде объекта Cursor — объекта, из которого данные можно прочесть). Объявим поле для адаптера в классе MainActivity, создадим его в методе onClick для кнопки show и назначим его listView:

```
show.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        adapter = new SimpleCursorAdapter(getApplicationContext(),
            android.R.layout.simple_list_item_2, null,
            new String[] { ContactsContract.CommonDataKinds.Phone.DISPLAY_NAME,
            ContactsContract.CommonDataKinds.Phone.NUMBER},
            new int[] { android.R.id.text1, android.R.id.text2 }, 0);

        listView.setAdapter(adapter);

        //TODO: загрузить список контактов в адаптер
    }
});
```

Остановимся более подробно на параметрах конструктора `SimpleCursorAdapter`. Первым параметром передается текущая активность, так как адаптеру нужен Context для создания отдельных view для элементов списка. Следующий параметр — идентификатор ресурса, соответствующего файлу разметки для одного элемента, который будет отображаться в `ListView`. Можно было бы создать свой ресурс со своей разметкой, но в этом упражнении мы воспользуемся встроенным в Android файлом разметки с идентификатором `android.R.layout.simple_list_item_2`. Этот файл разметки содержит два текстовых поля. По сути, он состоит из двух `TextView` с идентификаторами `text1` и `text2` с отличающимися настройками отображения (текст в `text1` крупнее).

Вернемся к конструктору `SimpleCursorAdapter`. Очередной, третий, параметр — это объект класса `Cursor`, из которого адаптер будет читать данные. Так как в момент вызова метода `onCreate` курсора у нас еще нет, он будет получен с помощью загрузчика в отдельном потоке, и его получение может занять ненулевое время, в качестве параметра передан `null`.

Следующие два параметра — это массив `from` и массив `to`. Первый из них — массив строк, который содержит список названий колонок объекта `Cursor` для отображения. Второй массив — это массив идентификаторов `View`, в которых должны быть отображены значения из колонок первого массива. Таким образом, передав в качестве первого массива `new String[] { ContactsContract.CommonDataKinds.Phone.DISPLAY_NAME, ContactsContract.CommonDataKinds.Phone.NUMBER }`, а в качестве второго — `new int[] { android.R.id.text1, android.R.id.text2 }`, можно добиться того, чтобы во `View` с идентификатором `text1` отображалось имя контакта, а во `View` с идентификатором `text2` — номер телефона.

Последний параметр конструктора — это набор дополнительных флагов, значение которых можно узнать в документации. В этом упражнении они нам не нужны, передадим 0 в качестве этого параметра.

После того, как адаптер создан и назначен `ListView`, нужно каким-то образом получить данные о контактах в виде курсора, из которого можно читать данные о контактах, и передать этот курсор в адаптер. Для этого используем загрузчик. Менеджеру загрузчиков нужно будет передать `LoaderManager.LoaderCallbacks`, чтобы он оповещал нас о процессе загрузки, поэтому реализуем этот интерфейс в нашем классе активности. В качестве шаблонного типа используем `Cursor`, так как именно его мы собираемся загружать с помощью загрузчика. Реализация методов интерфейса `LoaderManager.LoaderCallbacks` довольно простая:

```

@Override
public Loader<Cursor> onCreateLoader(int id, Bundle args) {
    return new CursorLoader(getApplicationContext(), ContactsContract.CommonDataKinds.P
hone.CONTENT_URI, new String[]{
        ContactsContract.CommonDataKinds.Phone._ID,
        ContactsContract.CommonDataKinds.Phone.DISPLAY_NAME,
        ContactsContract.CommonDataKinds.Phone.NUMBER,
        ContactsContract.CommonDataKinds.Phone.PHOTO_ID},null,null,null);
}

@Override
public void onLoadFinished(Loader<Cursor> loader, Cursor data) {
    adapter.swapCursor(data);
}

@Override
public void onLoaderReset(Loader<Cursor> loader) {
    adapter.swapCursor(null);
}

```

Метод `onCreateLoader` будет вызван при необходимости создать новый курсор. Параметр `id` может быть проигнорирован, так как в этом упражнении используется только один курсор; если бы их было несколько, в зависимости от значения этого параметра нужно было бы создавать нужный курсор. В реализации метода мы просто создаем новый загрузчик — `CursorLoader`, передавая ему контекст, URI для отправки запроса контент провайдеру и «проекцию» — массив с именами полей, которые мы хотим получить в результатах. Стоит отметить, что этот массив полей должен содержать все поля, которые будут отображаться в `ListView` (иначе, у `ListView` не будет каких-то данных для отображения), но он не обязан совпадать с ним полностью; например, в коде выше дополнительно запрашивается поле `ContactsContract.CommonDataKinds.Phone.PHOTO_ID`, которое не отображается в `ListView`. Кроме того, в этот набор обязательно должен входить идентификатор записи (в данном случае `ContactsContract.CommonDataKinds.Phone._ID`, который соответствует строке «`_id`»). Оставшиеся параметры отвечают за фильтрацию списка результатов (например, если мы хотим показать только определенные контакты, а не все) и за их порядок; в этом упражнении они нас не интересуют, поэтому, мы передаем `null` в качестве этих параметров.

Метод `onLoadFinished` вызывается, когда загрузчик закончил свою работу и имеется готовый курсор. Этот курсор просто передается в соответствующий метод адаптера, который уже позаботится о том, чтобы отобразить необходимые данные в `ListView`. Если до этого в адаптере был предыдущий курсор, о его освобождении позаботится загрузчик, поэтому мы используем метод `swapCursor` для замены предыдущего курсора (если он был) на новый и нигде не вызываем метод `close` у курсора — об этом заботится загрузчик.

Метод `onLoaderReset` вызывается, когда загрузчик «сбрасывается», например, потому что менеджер загрузчиков хочет повторно использовать этот загрузчик для какой-то другой задачи. Нужно забыть все ссылки на данные этого загрузчика, так как скоро они станут недоступны, что и реализовано путем передачи `null` в адаптер в качестве курсора. Наконец, необходимо инициализировать создание загрузчика в методе `onClick` для кнопки `Show`, добавив в конец метода строку:

```
getSupportLoaderManager().initLoader(0, null, this);
```

В качестве идентификатора загрузчика мы передаем 0 (реализация метода onCreateLoader игнорирует идентификатор в любом случае), а в качестве последнего параметра передаем this, то есть ссылку на активность, которая и реализует методы onCreateLoader, onLoadFinished и onLoaderReset, чтобы менеджер загрузчиков смог их вызывать в нужный момент.

В результате, если вы запустите приложение на устройстве и нажмете кнопку Show, вы сможете увидеть список контактов с номерами телефонов. Если на вашем тестовом устройстве книга контактов пустая, то ничего не отобразится. Конечно, можно создать список контактов во встроенным приложении «Контакты», однако давайте реализуем возможность сохранения контактов в базу из нашего приложения.

Метод для создания имени и номера телефона контакта может выглядеть следующим образом:

```
private void addContact(String name, String phone) {  
    ArrayList<ContentProviderOperation> ops = new ArrayList<>();  
    int rawContactInsertIndex = ops.size();  
  
    ops.add(ContentProviderOperation.newInsert(ContactsContract.RawContacts.CONTENT_URI  
I)  
        .withValue(ContactsContract.RawContacts.ACCOUNT_TYPE, null)  
        .withValue(ContactsContract.RawContacts.ACCOUNT_NAME, null).build());  
    ops.add(ContentProviderOperation  
        .newInsert(ContactsContract.Data.CONTENT_URI)  
        .withValueBackReference(ContactsContract.Data.RAW_CONTACT_ID, rawContactInse  
rtIndex)  
        .withValue(ContactsContract.Data.MIMETYPE, ContactsContract.CommonDataKind  
s.StructuredName.CONTENT_ITEM_TYPE)  
        .withValue(ContactsContract.CommonDataKinds.StructuredName.DISPLAY_NAME, na  
me) // Name of the person  
        .build());  
    ops.add(ContentProviderOperation  
        .newInsert(ContactsContract.Data.CONTENT_URI)  
        .withValueBackReference(  
            ContactsContract.Data.RAW_CONTACT_ID, rawContactInsertIndex)  
        .withValue(ContactsContract.Data.MIMETYPE, ContactsContract.CommonDataKind  
s.Phone.CONTENT_ITEM_TYPE)  
        .withValue(ContactsContract.CommonDataKinds.Phone.NUMBER, phone) // Number  
of the person  
        .withValue(ContactsContract.CommonDataKinds.Phone.TYPE, ContactsContract.Co  
mmonDataKinds.Phone.TYPE_MOBILE).build()); // Type of mobile number  
  
    try {  
        ContentProviderResult[] res = getContentResolver().applyBatch(ContactsContract.  
AUTHORITY, ops);  
    } catch (RemoteException e) {  
        e.printStackTrace();  
    } catch (OperationApplicationException e) {  
        e.printStackTrace();  
    }  
}
```

Данные могут быть вставлены/обновлены/удалены с использованием традиционных методов insert(Uri, ContentValues), update (Uri, ContentValues, String, String []) и delete (Uri, String, String []), однако существует более новый механизм, основанный на классе ContentProviderOperation.

Метод `applyBatch` является предпочтительным. Он вставляет контакт и его данные за одну транзакцию базы данных. Более подробно о классе `ContentProviderOperation` можно прочитать в официальной документации.

В методе `onClick` для кнопки `Save` считаем данные из текстовых полей и вызовем метод `addContact`:

```
save.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        String contactName = name.getText().toString();
        String contactPhone = phone.getText().toString();
        addContact(contactName, contactPhone);
        Toast.makeText(getApplicationContext(), "Contact added", Toast.LENGTH_LONG).show();
    }
});
```

Теперь можно сохранять контакты в базу данных устройства.

Сохраним несколько контактов и нажмем кнопку `Show`. Получим примерно следующий результат (см. рис. 4.51).

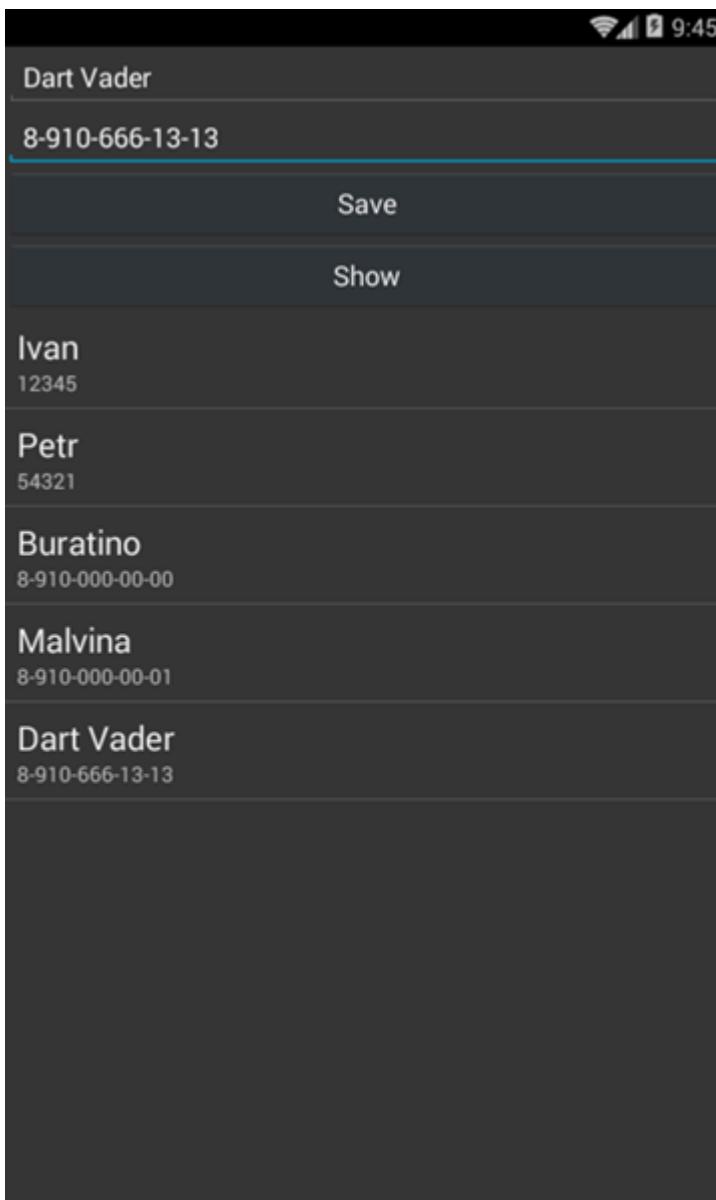


Рис. 4.51.

Теперь откроем встроенное приложение «Контакты» и убедимся, что введенные нами значения сохранились.

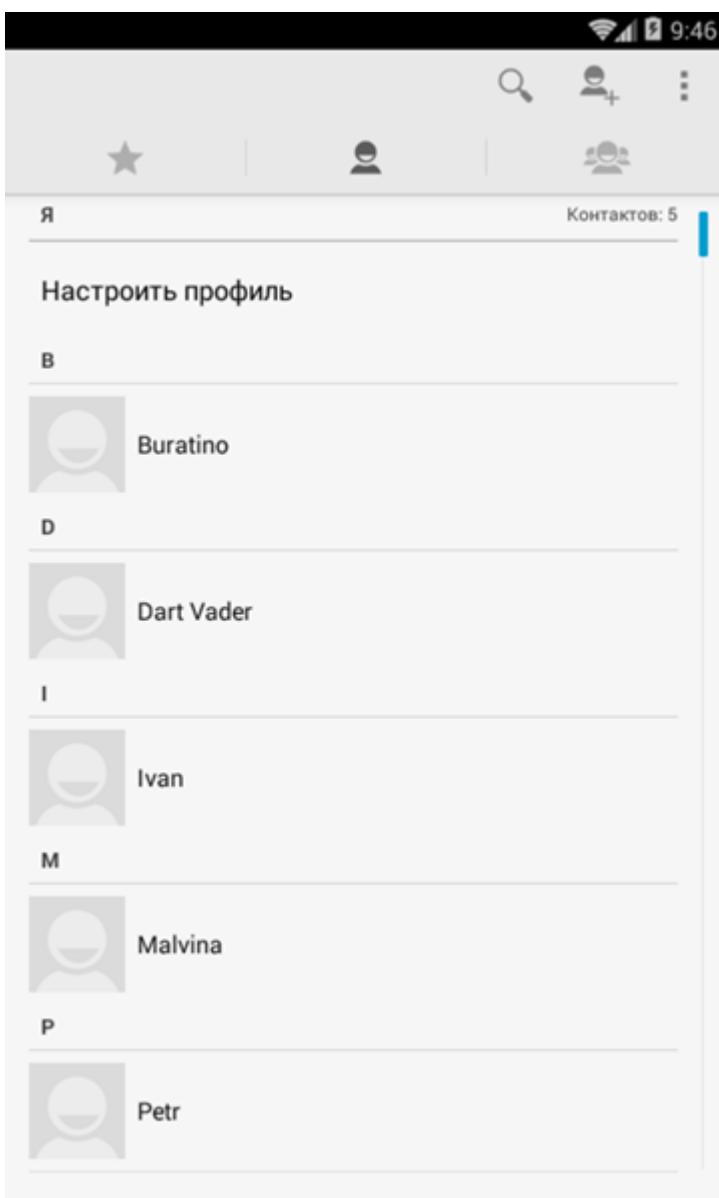


Рис. 4.52.

Полностью код проекта можно посмотреть [здесь](#).

#### Пример 4.27\*

В Android разработчик имеет возможность работать со всем медиаконтенте на мобильном устройстве. Для этого используется класс `MediaStore`, который является мощным провайдером данных и содержит централизованную базу данных мультимедиа (аудиофайлы, видеофайлы и изображения), размещенных во внутренней памяти устройства или на внешнем носителе (SD-карте).

Разработаем приложение для отображения списка миниатюр изображений, размещенных на устройстве. Для этого необходимо будет воспользоваться провайдером `MediaStore`. `Images.Thumbnails`.

Создадим новый Android-проект с названием `MediaStoreProviderSample`. Снова нужно отредактировать `AndroidManifest.xml`, так как для работы с файлами на диске нужно соответствующее разрешение. Добавим строку в этот файл:

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

В результате его содержимое должно выглядеть примерно так:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="ru.samsung.itschool.book.mediasotreprovidersample">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
    <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
</manifest>
```

В проекте снова будет одна основная активность — `MainActivity`, унаследованная от стандартного класса `FragmentActivity` для использования загрузчиков. `Activity` будет содержать один `ListView`, который и будет использован для отображения списка.

Приведем примерное содержимое класса активности (`MainActivity.java`) и соответствующего ей файла разметки (`activity_main.xml`):

```
public class MainActivity extends FragmentActivity {

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

        ListView listView = (ListView) findViewById(R.id.imageList);

        // TODO: реализовать отображение списка миниатюр
    }
}

<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout 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.mediasotreprovidersample.MainActivity">

    <ListView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:id="@+id/imageList">
    </ListView>

</RelativeLayout>
```

В этом примере создадим свой файл разметки для элемента списка, который будет содержать ImageView для показа миниатюры изображения, и TextView с информацией о пути к данной миниатюре. Для этого создадим файл с именем *my\_item.xml* в каталоге *layout* (рядом с файлом *activity\_main.xml*), в котором и опишем уже упомянутые ImageView и TextView. Вы можете расположить эти View иначе или изменить их размеры. Так или иначе, содержимое файла *my\_item.xml* должно стать примерно следующим:

```

<?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">
    <ImageView
        android:layout_margin="2dp"
        android:layout_width="140dp"
        android:layout_height="match_parent"
        android:id="@+id/imageView"
        android:contentDescription="image preview" />

    <TextView
        android:paddingLeft="8dp"
        android:gravity="center_vertical"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:id="@+id/textView"
    />

</LinearLayout>

```

Теперь можно перейти к созданию адаптера в методе onCreate активности. Создание адаптера аналогично примеру 4.11.1 за тем исключением, что мы используем свой ресурс для разметки элемента (my\_item) и свои view, указанные в этом ресурсе (imageView и textView), для отображения данных.

```

public class MainActivity extends FragmentActivity {

    private SimpleCursorAdapter adapter;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ListView listView = (ListView) findViewById(R.id.imageList);

        adapter = new SimpleCursorAdapter(this, R.layout.my_item, null,
            new String[] {
                MediaStore.Images.Thumbnails.DATA,
                MediaStore.Images.Thumbnails.DATA
            },
            new int[] {
                R.id.imageView,
                R.id.textView
            }, 0);
        listView.setAdapter(adapter);
        // TODO: добавить загрузчик, который получит данные о
        // миниатюрах изображений на устройстве, и передать
        // эти данные в адаптер
    }
}

```

Будем отображать одно и то же содержимое в обоих view — непосредственно URI данной миниатюры (колонку с именем MediaStore.Images.Thumbnails.DATA). Класс SimpleCursorAdapter правильно покажет это URI в виде изображения при отображении на ImageView и в виде текста при отображении на TextView (SimpleCursorAdapter не поддерживает других типов View, кроме TextView и ImageView, но можно написать свою реализацию адаптера, если необходимо отображать данные более сложной структуры).

Далее остается лишь закончить проект, добавив загрузчик, который будет получать необходимые данные и передавать их адаптеру. Этот процесс подробно описан в примере 4.11.1. Стоит отметить, что в качестве URI для запроса мы используем MediaStore.Images.Thumbnails.EXTERNAL\_CONTENT\_URI, а в качестве «проекции» (набора читаемых колонок) колонки с идентификаторами MediaStore.Images.Thumbnails.\_ID и MediaStore.Images.Thumbnails.DATA. Приведем окончательный вид класса MainActivity:

```
public class MainActivity extends FragmentActivity implements LoaderManager.LoaderCallback<Cursor>{

    private SimpleCursorAdapter adapter;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ListView listView = (ListView) findViewById(R.id.imageList);

        adapter = new SimpleCursorAdapter(this, R.layout.my_item, null,
            new String[] {
                MediaStore.Images.Thumbnails.DATA,
                MediaStore.Images.Thumbnails.DATA
            },
            new int[] {
                R.id.imageView,
                R.id.textView
            }, 0);
        listView.setAdapter(adapter);

        getSupportFragmentManager().initLoader(0, null, this);
    }

    @Override
    public Loader<Cursor> onCreateLoader(int id, Bundle args) {
        return new CursorLoader(this,
            MediaStore.Images.Thumbnails.EXTERNAL_CONTENT_URI,
            new String[] {
                MediaStore.Images.Thumbnails._ID,
                MediaStore.Images.Thumbnails.DATA
            }, null, null, null);
    }

    @Override
    public void onLoadFinished(Loader<Cursor> loader, Cursor data) {
        adapter.swapCursor(data);
    }

    @Override
    public void onLoaderReset(Loader<Cursor> loader) {
        adapter.swapCursor(null);
    }
}
```

В результате, если у вас имеется несколько фотографий на устройстве, после запуска приложения вы сможете увидеть их миниатюры в виде списка (рис. 4.53).

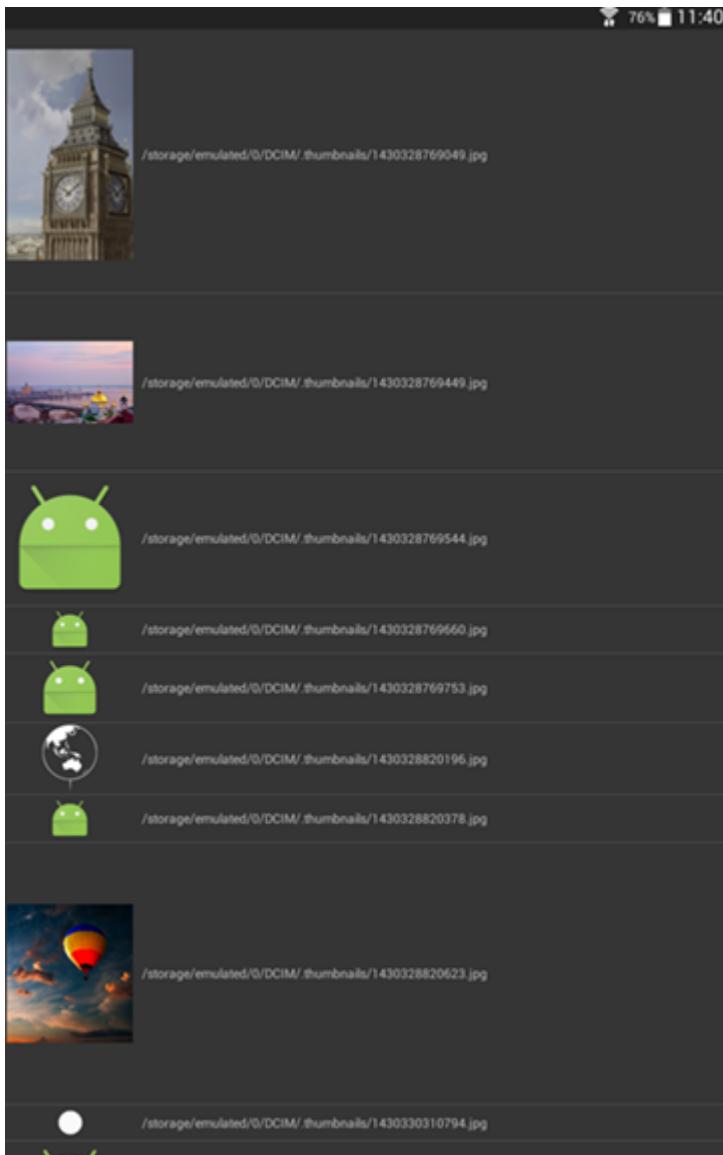


Рис. 4.53.

Проект разобранного примера можно посмотреть здесь.

## **4.12.\* Введение в криптографию и криptoанализ**

Сайт: IT Академия SAMSUNG

Курс: MDev @ IT Академия Samsung

Книга: 4.12.\* Введение в криптографию и криptoанализ

Напечатано:: Егор Беляев

Дата: Суббота, 18 Апрель 2020, 19:31

# **Оглавление**

- 4.12.1. Введение
  - 4.12.2. Шифры подстановки
  - 4.12.3. Шифры перестановки
  - 4.12.4. Шифрование и ЭВМ
  - 4.12.5. Криптоанализ
- Задание

## 4.12.1. Введение

**Шифрование** — это обратимое преобразование текста (информации) с целью сокрытия информации от посторонних.



Наивные шифры чаще всего основаны на том, что постороннее лицо не знает метода шифрования.

Современное шифрование основано на трудности расшифровки даже при опубликованном методе шифрования.

На протяжении всего процесса развития цивилизации интерес к сокрытию информации всегда был высок. Вариантов было очень много. Например, разного рода тайнопись, то есть способ сделать текст невидимым для постороннего: писать секретные сведения молоком на бумаге между строк; или татуировка на обритой голове раба (впоследствии выросшие волосы маскировали сообщение). Однако гораздо эффективнее были шифры, так как раскрытие информации было практически невозможным без алгоритма и/или ключа к шифру.

## 4.12.2. Шифры подстановки

*Простой подстановочный шифр* (или простой замены, моноалфавитный шифр) — класс методов шифрования, которые сводятся к созданию по определенному алгоритму таблицы шифрования, в которой для каждой буквы открытого текста существует единственная сопоставленная ей буква шифр-текста. Само шифрование заключается в замене букв согласно таблице. Для расшифровки достаточно иметь ту же таблицу, либо знать алгоритм, по которой она генерируется.

К шифрам простой замены относятся многие способы шифрования, возникшие в древности или средневековье, как, например, Атбаш или Шифр Цезаря. Для вскрытия подобных шифров используется частотный криптоанализ.

### Шифр Цезаря

Шифр Цезаря — один из самых простых и наиболее широко известных методов шифрования. Активно использовался императором Гаем Юлием Цезарем в переписке со своими военачальниками в Древнем Риме (50 г. до н. э.).

**Шифр Цезаря** — это вид шифра подстановки, в котором каждый символ в открытом тексте заменяется символом, находящимся на некотором постоянном числе позиций левее или правее него в алфавите.

Это постоянное число позиций сдвига при шифровании называется ключом шифра Цезаря.

Например, для русского алфавита в шифре со сдвигом вправо с ключом 3 (именно такой ключ применял Цезарь), А будет заменена на Г, Б станет Д, и так далее (рис. 4.54).

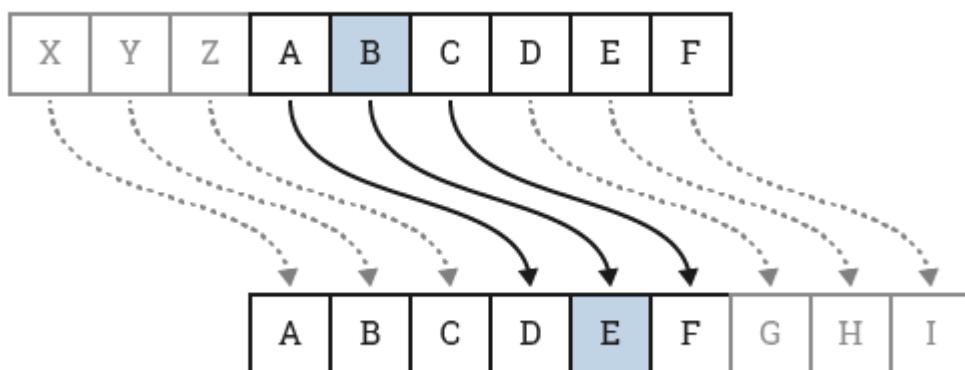


Рис. 4.54.

Шифр Цезаря со сдвигом на 3:

А заменяется на Д

В заменяется на Е

...

Z заменяется на С

Итак, подведем итог. Получается, что для дешифрования сообщения, закодированного шифром Цезаря, нужно знать примененный:

- алфавит;
- ключ;
- направление сдвига (вправо или влево).

## ROT13

Современная разновидность шифра Цезаря — ROT13 (от англ. «*rotate*»). Число 13 в наименовании шифра означает, что ключ равен 13.

Этот шифр подстановкой широко используется в интернет-форумах как средство для скрытия спойлеров, основных мыслей, решений загадок от случайного взгляда. ROT13 был охарактеризован как «сетевой эквивалент того, как в журналах печатают ответы на вопросы викторин — перевернутыми буквами».

При алгоритме ROT13 каждый буквенный символ английского алфавита заменяется на соответствующий ему со сдвигом на 13 позиций. А цифры, пробелы и все остальные символы остаются без изменений.

Почему в качестве ключа выбрано 13? Потому что он разрабатывался для английского алфавита, а там всего 26 букв, а  $26 = 2 \times 13$ . Такой ключ позволяет получить так называемый «*взаимный шифр*» — когда исходное и закодированное сообщение можно получить друг из друга, применив один и тот же алгоритм.

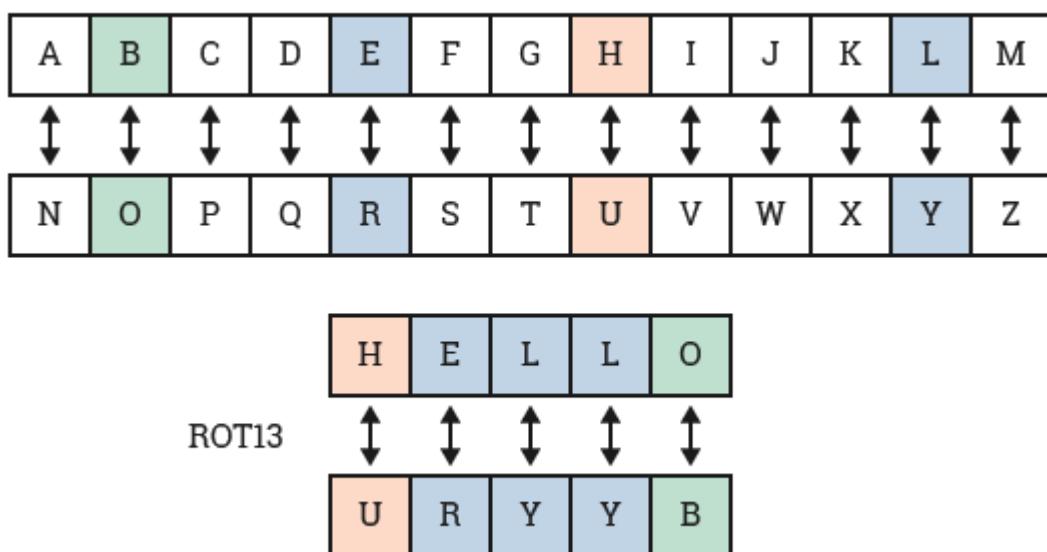
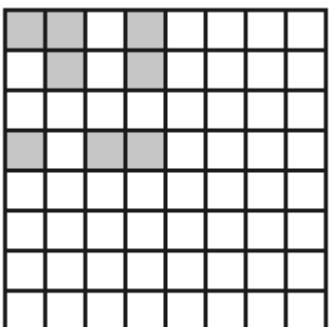


Рис. 4.55. ROT13 — шифровка и расшифровка происходит одинаково

### 4.12.3. Шифры перестановки

В качестве альтернативы шифрам подстановки можно рассматривать перестановочные шифры. В них элементы текста переставляются в ином от исходного порядке, а сами элементы остаются неизменными. Тогда как в шифрах подстановки, элементы текста не меняют свою последовательность, а изменяются сами.

# Решетка Кардано



Этот способ шифрования был придуман итальянским математиком Джероламо Кардано. Он представлял собой трафарет с прорезанными окошками, через которые на лист бумаги последовательно записывался текст. После заполнения всех окошек трафарет поворачивали на 900. И так три раза, после чего решетку Кардано убирали и клетки, оставшиеся пустыми, заполняли «мусором»: различными буквами, знаками препинания, цифрами (в произвольном порядке, чем хаотичнее, тем лучше). Например, давайте зашифруем сообщение «шифрование с помощью решетки Кардано» (рис. 4.56).

ш	и	ф	и		е
р	о			с	п
		о			
в	а	н	м	о	щ
и	к	а	ь	ю	р
		р			
д	а		е		ш
н		о	е	т	к

ш	и	к	ф	и	т	с	е
м	р	в	о	п	а	с	п
в	и	р	а	о	ц	ф	ю
в	я	а	н	м	п	о	щ
и	к	ц	а	ь	ю	э	р
и	с	х	р	г	л	е	ч
д	а	ю	ж	е	ы	ш	з
н	ч	й	о	е	й	т	к

Рис. 4.56.

## Конь Эйлера

Задача, на основе которой и составлен шифр, заключается в нахождении маршрута шахматного коня, проходящего через все поля доски только один раз. Этот маршрут и является порядком заполнения текстом квадратной матрицы  $8 \times 8$ . Затем выписывается текст

слева-направо. Максимальная длина сообщения — 64 символа.

## Шифр Сцитала

Очень удачным примером шифра перестановки является шифр Сцитала, использовавшийся еще во времена Древней Спарты. Ключом такого шифра была цилиндрическая палочка, а шифрование выполнялось следующим образом:

- узкая пергаментная лента наматывалась по спирали на цилиндрическую палочку;
  - шифруемый текст писался на пергаментной ленте по длине палочки, после того как длина палочки оказывалась исчерпанной, она поворачивалась и текст писался далее, пока либо не заканчивался текст, либо не исписывалась вся пергаментная лента. В последнем случае использовался очередной кусок пергаментной ленты.



Расшифровка выполнялась с использованием палочки такого же диаметра. Таким образом, длина блока определялась длиной и диаметром палочки, а само шифрование заключалось в перестановке символов исходного текста в соответствии с длиной окружности палочки.

Например, используя палочку, по длине окружности которой помещается 4 символа, а длина палочки позволяет записать 6 символов, исходный текст «это шифр Древней Спарты» превратится в шифrogramму «эфвптрнао ер дйтшр ыиес». Длина блока  $n = 23$ , а вектор  $t$ , указывающий правило перестановки, для этого шифра может быть записан следующим образом:  $t = \{1, 7, 13, 19, 2, 8, 14, 20, 3, 9, 15, 21, 4, 10, 16, 22, 5, 11, 17, 23, 6, 12, 18\}$ .

#### 4.12.4. Шифрование и ЭВМ



Рис. 4.57.

История использования вычислительных машин восходит к времени Второй мировой войны. В это время Германия использовала передовую по тем временам электрическую шифровальную машину «Энigma» для обмена сообщениями в войсках (рис. 4.57). На тот момент без учета настройки положения колец (нем. Ringstellung), количество различных ключей составляло 1016.

В течение 1920–1930 гг. группа польских математиков искала способ дешифровки сообщений и достигла некоторых успехов. В частности, они догадались использовать машины подобные Энигме для расшифровки сообщений. Чуть позже Аллан Тьюринг создал дешифровальную машину Bombe на основе этого прототипа.

Дальнейшая работа по взлому была организована в главном шифровальном подразделении Великобритании — Правительственной школе кодов и шифров Блетчли-парк (центр «Station X»). В разгар деятельности Блетчли-парк насчитывал 12 тысяч человек, но, несмотря на это, немцы не узнали о нем до самого конца войны! Сообщения, расшифрованные центром, имели гриф секретности «Ultra» — выше, чем использовавшийся до этого «Top Secret». Англичане предпринимали повышенные меры безопасности, чтобы Германия не догадалась о раскрытии шифра.

Начиная с этого времени, вся работа по шифрованию/расшифровыванию и криптоанализу ведется только с применением ЭВМ. Конечно, изменились алгоритмы, подходы, матаппарат, но не изменилось роль шифрования в жизни стран и людей.



Ярким эпизодом деятельности Блетчли-парк является случай с бомбардировкой Ковентри 14 ноября 1940 года, о которой премьер-министру Великобритании Уинстону Черчиллю было известно заранее благодаря расшифровке приказа. Однако Черчилль, опираясь на мнение аналитиков о возможности Германии догадаться об операции «Ультра», принял решение о непринятии мер к защите города и эвакуации жителей.

«Война заставляет нас все больше и больше играть в Бога. Не знаю, как бы я поступил...», — говорил президент США Франклин Рузельт о бомбардировке Ковентри.

Для СССР существование и даже результаты работы «Station X» секрета не представляли. Именно из результатов сообщений, дешифрованных в «Station X», СССР узнал о намечающемся «реванше» Гитлера за Сталинградскую битву и смог подготовиться к операции на Курском направлении, получившей название «Курская дуга».

Посмотреть на работу Энигмы можно здесь <http://enigmaco.de/enigma/enigmahtml>.

## 4.12.5. Криптоанализ

**Криптоанализ** — наука о методах расшифровки зашифрованной информации без предназначенного для такой расшифровки ключа. Первоначально методы криптоанализа основывались на лингвистических закономерностях естественного текста и реализовывались с использованием только карандаша и бумаги. Со временем в криптоанализе нарастает роль чисто математических методов, для реализации которых используются специализированные криптоаналитические компьютеры.

Попытку раскрытия конкретного шифра с применением методов криптоанализа называют **криптографической атакой** на этот шифр. Криптографическую атаку, в ходе которой раскрыть шифр удалось, называют **взломом** или **вскрытием**.

Хотя понятие криптоанализ было введено сравнительно недавно, некоторые методы взлома были изобретены десятки веков назад. Первым известным письменным упоминанием о криптоанализе является «Манускрипт о дешифровке криптографических сообщений», написанный арабским ученым Ал-Кинди еще в 9 веке. В этом научном труде содержится описание метода частотного анализа.

**Частотный анализ** — основной метод атаки на большинство классических шифров перестановки или замены. Этот метод основывается на предположении о том, что каждый символ алфавита встречается в тексте неодинаковое количество раз и в открытом тексте, и в шифротексте. При этом при условии достаточно большой длины шифрованного сообщения шифры, основанные на одном алфавите, легко поддаются частотному анализу: если частота появления буквы в языке (смотри ниже таблицы частотности) и частота появления некоторого присутствующего в шифротексте символа приблизительно равны, то в этом случае с большой долей вероятности можно предположить, что данный символ и будет этой самой буквой.

Применение высокопроизводительных вычислительных систем сделало возможным вскрытие слабых шифров **методом простого перебора** (*brute force*), так, например, был взломан шифр CSS — системы защиты цифрового медиаконтента на DVD-носителях.

Совершенствование математического аппарата криптоанализа позволило, например, создать программное обеспечение, которое даже на не особо производительных вычислительных системах, доступных частным лицам, может атаковать протоколы шифрования WiFi WEP/WPA2PSK. Например, персональный компьютер с четырьмя видеокартами AMD HD7770 способен перебирать несколько миллиардов паролей WPA в час при помощи алгоритма радужных таблиц (rainbow tables) ПО Pyrit.

Как следствие большинство современных алгоритмов шифрования (AES, 3DES, Blowfish, ГОСТ 28147–89) имеют достаточно высокую криптостойкость (порядка 2224 вариантов ключа), для того чтобы их взлом стал слишком длительным (годы или сотни лет) и слишком дорогим (потребуется суперкомпьютер).

Кроме криптоанализа есть и иные способы атак на шифры и протоколы. Это, например, похищение ключа человеком или вирусом, или нахождение дефекта в алгоритме, который может резко снизить сложность взлома.

Также государствами законодательно регулируется использование средств шифрования, что дает возможность спецслужбам контролировать передачу информации частных лиц. Например, система *Carnivore* в США или СОРМ-2 в России, установленная у всех интернет-провайдеров и операторов связи, позволяет отслеживать весь интернет-трафик. Кроме того, в России Федеральный закон № 40 «О Федеральной службе безопасности» в статье 13 ограничивает производство и ввоз оборудования с сильными алгоритмами шифрования.

Русский язык		English	
буква	частотность	Letter	frequency
о	0,10983	a	0,8167
е	0,08483	b	0,01492
а	0,07998	c	0,02782
и	0,07367	d	0,04253
н	0,07367	e	0,12702
т	0,06318	f	0,02228
с	0,05473	g	0,02015
р	0,04746	h	0,06094
в	0,04533	i	0,06966
л	0,04343	j	0,00153
к	0,03486	k	0,00772
м	0,03203	l	0,04025
д	0,02977	m	0,02406
п	0,02804	n	0,06749
у	0,02615	o	0,07507
я	0,02001	p	0,01929
ы	0,01898	q	0,00095
ъ	0,01735	r	0,05987
г	0,01687	s	0,06327
з	0,01641	t	0,09056
б	0,01592	u	0,02758
ч	0,0145	v	0,00978
й	0,01208	w	0,0236
х	0,00966	x	0,0015
ж	0,0094	y	0,01974
ш	0,00718	z	0,00074
ю	0,00639		
ц	0,00486		
щ	0,00361		
э	0,00331		
ф	0,00267		
ъ	0,00037		
ё	0,00013		

Табл. 4.22. Таблицы частотности для русского и английского языков

## Пример криptoанализа

### Расшифровка, вариант I

Возьмем какое-либо email-сообщение и зашифруем его шифром Цезаря. Это можно сделать, например, при помощи онлайн-утилиты шифрования <http://planetcalc.ru/1434/ROT3>. Например, зашифрованный текст:

Жсдуюм жзря. Ргтсплргб, ъхс кгехуг 10ёс врегув ц рғф жстсорлхзоярсз кгрвхлз тс ССТ л ъцхя-  
ъцхя тс тусзnhгп. Нхс псизх тулшсхлхз н 14:00 Ф цегизрлзп Ж.Е.Вщзрнс

Получив такое сообщение, попробуем проанализировать и получить исходное сообщение. Для начала сделаем несколько допущений.

- Язык оригинала — русский.
- Поскольку явно прослеживается структура текста — сообщение, скорее всего, не подвергалось сжатию или другой обработке.
- Судя по внешнему виду, мы, скорее всего, имеем дело с алгоритмом простой подстановки. Исходя из вышесказанных допущений и зная, что перед нами письмо электронной почты (а такого рода информацию далеко не всегда можно получить), можно предположить, что письмо начинается с приветствия. Какие два слова (6 и 4-е буквы) могут означать приветствие? Наверняка *Жсдуюм жзря*. == *Добрый день*. Итак, мы получили первые 9 пар подстановки:

Ж с д у ю м з р я  
Д о б р ы й е н ь

Обратите внимание, что все подстановочные буквы имеют сдвиг от оригинала на 3 позиции в алфавите. То есть, скорее всего, это сдвиговый шифр Цезаря со сдвигом 3. Заменим все буквы в исходном сообщении на буквы со сдвигом назад на 3 и получим оригинал сообщения:

*Добрый день. Напоминаю, что завтра 10го января у нас дополнительное занятие по ООП и чуть-чуть по проектам. Кто может приходите к 14:00 С уважением Д. В. Яценко*

Обратите внимание, что в первом варианте расшифровки мы знали, что перед нами письмо. И еще нам очень повезло, что в исходном письме остались пробелы, знаки препинания и большие буквы. А вот если бы ничего этого не было?

### Упражнение 4.12.1

Зашифруйте шифром Цезаря какое-либо сообщение. Проведите криptoанализ зашифрованного сообщения. Пример зашифрованного сообщения:

жээмпфдсржеютсорвзхкгжгрлзефхгпдцозфтгфгвийфхнлмжлфнфсфлфнспгёэрхседулхгфрнсмугкезж нлтсжтулнуюхлзпесеуззвфшегхнлфргипрлнсптгхулфсплепгрлтзррлгтгурлшгдсржтстулнгкцпфх узовзхергипрлнгрстстгжгхедсржггёэрхтгжгзхфпсфхгескзусеплбзёсфъльхгбхтсёлдылпргипрлнц фнсоякгзхфжлфнспъузкрзnsхсусеузпвлбцкргбхсдцхзънзлрчсупгщлшгнзуюеногжюегбхлпэр гелрхзурзхсдзьгвфссдъгхянгйжцбржзоббрецеузшлпзресеуззвфхузыллпёгузхгпаоосулёогеюсдээ жлриррсёсугкезжюегхзоярсёснсплхзхгёгузхфхгелхтсжфспрзлзинсптзхзрхрсфхялтузжогёгзхт съихрцбсхфхгенцсхнгкюегзхфвжгегвтсрвхяъхсжсойртзузжшжсптулезфхлжзогетсувжнфугкц тсфозефхузылехднегухлузплбсуёгрлксегрекуюетулезжылмнпрсёсълфозррюпизухегфуэжлфсху цжрлнсеуугкезжнлерзкгттрсегрёолбескеуగъзхфвеюлыеылмдсржгрголфхцгшллтснгкюегзхъхсезу свхрззефзэскгхззусулфхълзфнлпгхглфхслхдюеымфсхуцжрлнплбдржкгрсесфжгиханкгзррж стцфннугкезжюегхзоярсмугдсхзрзфпсчхуvrгхсъхжэмпфрзтусшжланкгпзрпргтугеовзхэёсргрс есзкгжгрлзгёэрхкргнсплхфврфссеюпнлтсоцъгзхфргуийзрлзфтзшлгоярюмтлфхсозхлугжлстзозрёгх судсржсчтugeовзхфвеыгршмёжеюфозийлегзхтгхулфглцдлгзхзёссхтгхулфгозжезжихннлдзухзу усуулфхцхявёсусужулеозфцдюеызпцгёэрхцплблжгерзпцкргнспспцтсфозцфтырсмстзугшллпгнгслр гсфхусезебирснлхгмфнсппсузфлюеццжгихфвкгжэуигхялжсфхгелхяеосржсрешжзкгжэуигрлвфлю ергрпзнгзхдсржцсхспыхспрздзузийхфеслшгёэрхселзёсфгпсёстсжфхгелолугжлеютсорзрлвегирсм стзугшллгфздвлдсржфугерлегзхфнуофсдсвпргжстусфзфлюеугуффнгкюегзххсдцжцьлетозрцтю хгофвтснрълхяфсдсмтуснцфленгтфцоцфцлгрлжспрсвжхояяфтуесесщлусегосфхзсплзолхезушрэмь зобфхллозесмфнцосесмнсфхлеузкцияхгхэзёссреюрцйжзррсфлхяефхгерцбъзобфхяфпзхгоользфнс мфнцосмрзмefнсузтuzфхцтrlнццжгихфвдэйгхятуvпслкыгхднегухлуюплблцмхлъзуктсжкзпрюмш скжолрллпзхустузфозжцвзёсдсржтсрлпгзхфлюеагтсфхсвррсстузйгзхзёсдогёсжгувхспцхнсрх усолуцзхнсптиябхзурцбфзхядулхгфрнсмугкезжнлефиетосхяжстозрлваозпзрхютогрфлюеагтюгх зхфвсфцъзфхелхятснцызрлзгрилкряпесеузпвкгфзжгрглвтгуогпзрхфнсёснсплхзхгдсржетзузфхуз нэфтгфзхплцесклхзниеусжесзтспзфхяэдсржсееысхогржлсфсdrvнfнгмчкофлюеэрпзуэррссфх геовбхфозженсптиябхзурсмфзхлплб

Тут расшифровать, как в первом варианте, угадыванием слов не получится. Что же делать? Если мы догадываемся, что шифр сдвиговый, то можно попробовать по очереди расшифровать, заменяя буквы со сдвигом, на 1, потом на 2, потом на 3 и т. д. Это режим грубой силы (brute force). Однако он применим, если шифр сдвиговый, а если он просто с фиксированными парами (по случайно выбранной таблице)?

В этом случае придется применить частотный анализ. Для этого посчитаем частоту появления каждого символа в сообщении и поместим в таблицу 4.23.

**Частотность = (кол-во вхождений буквы)/(длина сообщения)**

с	0,0962	п	0,0378	ё	0,0147	ш	0,0049
з	0,0908	ж	0,0358	я	0,0142	щ	0,0039
г	0,0839	т	0,0334	ю	0,0137	ь	0,0020
х	0,0726	о	0,0319	ъ	0,0108	ч	0,0010
р	0,0653	ц	0,0236	й	0,0088	э	0,0005
ф	0,0579	в	0,0186	б	0,0074		
е	0,0545	к	0,0177	и	0,0069		
у	0,0486	д	0,0167	ы	0,0059		

Табл. 4.23.

Сравним полученную таблицу частотности с таблицей частотности русского языка. Особенно в начале таблиц заметно соответствие:

c -> o	z -> e	g -> a	l -> i
--------	--------	--------	--------

То есть как раз сдвиг на три позиции в алфавите. Пробуем расшифровать текст шифром ROT3 и получаем:

джеймсбондвыполняетзаданиевстамбулеспасаяжёсткийдискоспискомагентовбританскойразведкиподприкрытиемввремясхваткинаёмникомпатристомивминипенниапарницафондапоприказумстремлятьвнаёмниканопопадаетвбондаагентпадаетсмоставозеромибегосчитаютпогибшимнаёмникускользаетсдискомчерезнекотороевремявибузнаютобутечкеинформациихакерывыкладываютименавинтернетобещаясообщатькаждуюнеделю5новыхименввремявстречигаретамэллориглавыюбъединённогоразведывательногокомитетагаретставитподсомнениеёкомпетентностьипредлагаетпочётнуюотставкумотказываетсядаваяпонятьчто должна передуходомпривестидалпорядоксразупослевстречивштабквартиремиborgанизованвзрывприведшийкмногочисленнымжертвамсредисмотрудниковразведкинезапнованглиювозвращаетсясыжившийбонданализситуациипоказываетчтовероятнеевсегоизтеррористическимиактамистоитбывшийсотрудникибондановосдаётэкзаменнадопускразведывательнойработенесмотрянаточтоджеймснепроходитэкзаменнаправляетегонавоезданиеагентзнакомитсясновымкиполучаетснаряжениеспециальныйпистолетирадиопеленгаторбондотправляетсявшанхайгдеявлеживаетпатрисаиубиваетегоотпатрисасследуетккибертерпористутьягородригесубывшемуагентумидавнемузнакомумпослеуспешнойоперациивмакаинаостровевюжнокитайскомморесильвуудаётсязадержатьидоставитьвлондонвходедержаниясильванамекаетбондуоттомчтоинебережётсвоихгентовиегосамогоподставилирадивыполненияважнойоперацииасебяибондасравниваетскрысобояминадопросесильварассказываетчтобудучивпленупыталсяпокончитьсс собойпрокусивкапсулуцианидомноядлишьспровоцировалостеомиелитверхнейчелюстиилевойскуловойкостиврезультатечегоонвынужденноситьвставнуючелюстьсметаллическоискулойнанейвскорепреступникуудаётсябежатьпрямоизштабквартирымибиутическиерезподземныходилинииметропреследуяегобондпонимаетсильвапостоянноопережаетегоблагодарятомучтоонролируеткомпьютернуюсетьбританскойразведкивсёвплотьдоплененияэлементыпланасильвалытаетсяосуществитьпокушениенажизньмввремязаседанияпарламентскогокомитетабондперестрелкеспасаетмиувозитеёвродовоепоместьебондовшотландииособняксскойфоллсильвенмереннооставляютследкомпьютернойсетимиб

Профессионалы есть не только в MI6!

### Упражнение 4.12.2

Разработайте программу, которая осуществляет замену в тексте одного символа на другой. То есть символы, которые не указаны в подстановке, останутся без изменений. Таким образом, у вас будет одно большое поле ввода (EditText) для ввода исходного текста, а также два маленьких поля ввода (также EditText) для ввода заменяемой буквы и буквы, на которую заменяем. Кроме того, необходима кнопка для запуска процедуры замены (см. рис. 4.58). Здесь желательно вспомнить основные принципы разработки программ под Android. Помним, что приложение падает при неправильном вводе (например, пустой символ для замены). Догадайтесь, как исправить эту ошибку.

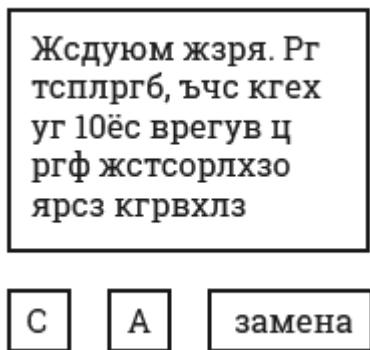


Рис. 4.58.

Приведем код приложения.

1. Макет — /res/layout/activity\_main.xml

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="${relativePackage}.${activityClass}" >
    <EditText
        android:id="@+id/editText"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:lines="10"
        android:hint="введите текст"
        android:background="#cccccc"
        android:ems="10" >
        <requestFocus />
    </EditText>
    <EditText
        android:id="@+id/from_char"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_below="@+id/editText"
        android:layout_marginLeft="28dp"
        android:layout_marginTop="22dp"
        android:background="#cccccc"
        android:ems="10"
        android:width="40dp" />
    <EditText
        android:id="@+id/to_char"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_alignTop="@+id/from_char"
        android:layout_toRightOf="@+id/from_char"
        android:layout_marginLeft="100dp"
        android:width="40dp"
        android:background="#cccccc"
        android:ems="10" />
    <Button
        android:id="@+id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignTop="@+id/from_char"
        android:layout_toRightOf="@+id/to_char"
        android:layout_marginLeft="50dp"
        android:onClick="replace"
        android:text="заменить" />
</RelativeLayout>
```

## 2. Класс активности: MainActivity.java

```

public class MainActivity extends Activity {
    TextView from_charTV,to_charTV;
    EditText edit_textET;
    String from_char,to_char,edit_text;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        from_charTV=(TextView)findViewById(R.id.from_char);
        to_charTV=(TextView)findViewById(R.id.to_char);
        edit_textET=(EditText)findViewById(R.id.editText);
    }
    public void replace(View v){
        from_char=from_charTV.getText().toString();
        to_char=to_charTV.getText().toString();
        edit_text=edit_textET.getText().toString();
        edit_text=edit_text.replaceAll(from_char, to_char);
        edit_textET.setText(edit_text);
    }
}

```

### Упражнение 4.12.3

Расшифруйте сообщение, зашифрованное подстановочным шифром, используя программу, разработанную в примере 4.12.1. Для более удобного криptoанализа зашифрованного сообщения добавьте в вашу программу еще одно текстовое поле (TextView), в которое выведите частотность встречающихся в первом текстовом поле букв. Подсказка — не забудьте из статистики убрать все символы не буквы (пробелы, символы, цифры и т. д.), иначе у вас исказятся частотности. Зашифрованный текст должен быть каким-то широкоизвестным, например, строфа из «Евгения Онегина» или стихотворение в прозе «Русский язык» И. С. Тургенева. Работа выполняется в комбинированном режиме — часть работы выполняется вручную (замена), часть — в автоматизированном (анализ частотности) режиме. Если в течение урока текст не удастся окончательно расшифровать, завершение работы можно задать в качестве домашнего задания.

Для решения этой задачи нужно добавить в предыдущее приложение дополнительную функцию расчета частотности, результат действия которой выводить в дополнительное TextView:

```

<TextView
    android:id="@+id/average"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_alignRight="@+id/from_char"
    android:layout_below="@+id/button1"
    android:layout_marginTop="15dp"
    android:lines="33"
    android:width="250dp"
    android:text="Таблица частотности(рус)" />

```

И дополнительный метод:

```
public void calculate(View v){  
    String s=edit_textET.getText().toString();  
    s = s.toLowerCase().replaceAll("[^a-я]+", "");  
    averageTV.setText("");  
    for (int i = 0; i < s.length(); i++) {  
        char a = s.charAt(i);  
        if (s.substring(0, i).indexOf(a) != -1)  
            continue;  
        int ch = 0;  
        int k = i;  
        for (int j = k; j < s.length(); ch++) {  
            int ii = s.indexOf(a, j);  
            if (ii == -1)  
                break;  
            j = ii + 1;  
        }  
        averageTV.setText(averageTV.getText().toString() + "\n" + a + " " +  
String.format(" %.4f", (double) ch / s.length()));  
    }  
}
```

## **Задание**

Доработать приложение из предыдущего упражнения, чтобы оно показывало уже введенные замены, а также частоту пяти наиболее часто встречающихся букв (\*двухбуквенных слов), для которых еще нет замены.

## **4.13.\* Ключи шифрования**

Сайт: IT Академия SAMSUNG

Курс: MDev @ IT Академия Samsung

Книга: 4.13.\* Ключи шифрования

Напечатано:: Егор Беляев

Дата: Суббота, 18 Апрель 2020, 19:31

# **Оглавление**

- 4.13.1. Введение
  - 4.13.2. Симметричный алгоритм шифрования
  - 4.13.3. Методы шифрования, основанные на алгоритме DES
  - 4.13.4. Асимметричный алгоритм шифрования
  - 4.13.5. Алгоритм RSA
- Задание 4.13.1
- Задание 4.13.2
- Литература

## 4.13.1. Введение

В повседневной жизни почти для всего существуют неформальные протоколы: заказ товаров по телефону, игра в покер, голосование на выборах. Никто не задумывается об этих протоколах, они вырабатывались в течение длительного времени, все знают, как ими пользоваться, и они работают достаточно хорошо.

### Брюс Шнайер

Сегодня все больше людей общаются не лично, а через компьютерную сеть. Честность и безопасность многих протоколов человеческого общения основаны на личном присутствии. Разве вы доверите незнакомцу деньги, чтобы он купил для вас автомобиль? Или сидете играть в покер с тем, кто жульничает при сдаче карт? А быть может, пошлете свой избирательный бюллетень, не удостоверившись, что он дойдет и дойдет невскрытым? И если человек при личной встрече может определить уровень доверия, то компьютеру нужны определенные правила.

Изучением вопросов шифрования данных занимается наука — криптология, включающая криптографию и криptoанализ.

**Криптография** изучает методы и алгоритмы шифрования данных (правила построения и использования шифров), направленные на то, чтобы сделать эти данные бесполезными для противника.

Методы криптографии также используются для подтверждения подлинности источника данных и контроля целостности данных.

**Криptoанализ** — это наука о раскрытии исходного текста зашифрованного сообщения без доступа к ключу.

Процесс шифрования информации с точки зрения программиста можно представить следующим образом:

Открытый текст -> алгоритм шифрования (ключ 1) -> шифрованное сообщение.

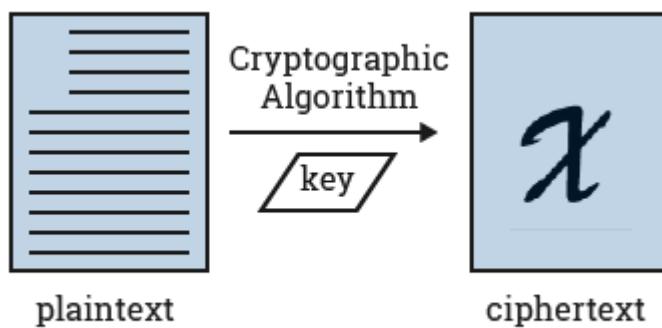


Рис. 4.59.

А процесс расшифровывания:

Шифрованное сообщение -> алгоритм дешифрирования (ключ 2) -> открытый текст.

**Открытый текст** — информация, которую надо скрыть (в общем случае она совсем не обязательно является текстом, состоящим из символов какого-либо языка; это может быть набор данных, компьютерная программа — что угодно).

**Шифрованное сообщение** — это информация, скрытая с помощью системы шифрования. Термин «сообщение» применяется, поскольку чаще всего шифрование используется для обмена секретной информацией между двумя сторонами, но шифрованное сообщение вовсе

не обязано быть куда-либо переданным, оно может быть, например, записано на жесткий диск.

**Алгоритм шифрования/десифрирования** — метод, которым пользуются для сокрытия и восстановления информации, а **ключ** — некоторая специальная, обычно строго секретная и регулярно изменяющаяся последовательность символов, необходимая для того, чтобы правильно применить алгоритм.

Например, для шифра Цезаря алгоритмом кодирования является замена буквы исходного сообщения на другую букву алфавита, сдвинутую на некоторое число от исходной, а ключом является это число.

Однако может возникнуть вопрос: почему мы говорим о секретности ключа, но не упоминаем о секретности алгоритма? Ведь скрыв реализацию функции « $F(\text{открытое сообщение}) \rightarrow \text{шифрованное сообщение}$ », мы могли бы не менее надежно защитить секрет. Действительно, ранее считалось, что одним из важнейших способов сохранения тайны является сокрытие метода шифрования, а желательно и наличия шифрованного сообщения (вспомните истории о секретных чернилах, маскировке шпионских донесений под обычную обывательскую корреспонденцию). Что и говорить, даже десятилетие назад утеря во время войны или крушение судна аппаратуры шифрования (которая и реализовывала алгоритм) считались достаточным основанием для замены всего подобного оборудования на всех объектах. Однако именно этот пример и показывает, что принцип «Безопасность через сокрытие (неясность)», или более привычно — «Security through obscurity», имеет много отрицательных моментов. Утеря техники, разглашение в открытой прессе принципов работы и даже просто то, что количество алгоритмов принципиально ограничено, привело к тому, что в современной криптографии считается, что сами алгоритмы шифрование/десифровки могут быть открытыми (и это не должно влиять на стойкость крипtosистемы), а вот ключи должны быть секретными.

Ключи, используемые для шифрования и десифрирования информации, могут быть как одинаковыми — в этом случае говорят о симметричном шифровании, так и различными — в случае использования асимметричных алгоритмов шифрования.

## 4.13.2. Симметричный алгоритм шифрования

Симметричный алгоритм шифрования предполагает, что и для прямого преобразования сообщения (из открытого текста в шифрованный), и для обратного используется один и тот же ключ, который, очевидно, должен храниться в секрете обеими сторонами. Если две стороны хотят обмениваться зашифрованными сообщениями в безопасном режиме, то обе стороны должны иметь одинаковые симметричные ключи.

Пусть Алиса и Боб хотят безопасно обменяться информацией (см. рис. 4.60). Для осуществления обмена должен быть определен протокол действий.

1. Алиса и Боб выбирают систему шифрования.
2. Алиса и Боб выбирают ключ.
3. Алиса шифрует открытый текст своего сообщения с использованием алгоритма шифрования и ключа.
4. Алиса посыпает шифрованное сообщение Бобу.
5. Боб дешифрует сообщения с использованием алгоритма шифрования и ключа.

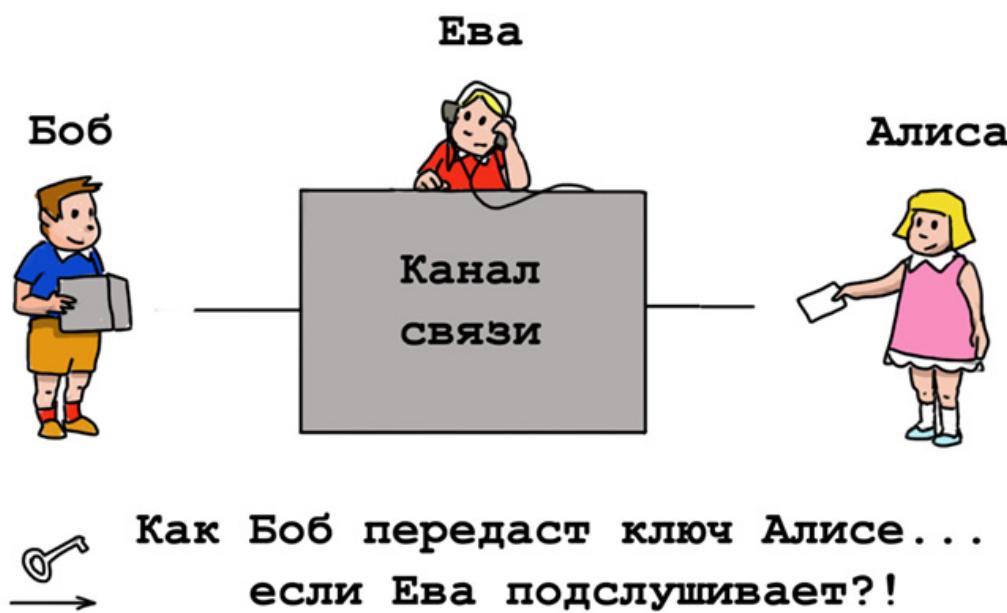


Рис. 4.60.

Что может узнать Ева, находясь между Алисой и Бобом и подслушивая протокол? Может подслушать передачу сообщения (этап 4). Тогда ей придется подвергнуть сообщения криptoанализу. Так же Ева может послушать этапы 1 и 2. Тогда ей, как и Бобу, станут известны алгоритм и ключ, и когда она перехватит сообщение, сможет его дешифровать.

Кроме этого, Ева может попытаться нарушить линию связи так, чтобы сообщение, отправленное Алисой, до Боба не дошло. Или же может перехватить сообщение Алисы, а Бобу отправить свое. А так как Ева знает алгоритм и ключ, то Боб не сможет догадаться, что сообщение отправлено не Алисой.

Шифрование симметричным ключом обычно используется для шифрования большого объема данных, так как этот процесс проходит быстрее, чем при асимметричном шифровании. Обычно используются алгоритмы DES (Data Encryption Standard — стандарт шифрования данных), 3-DES (тройной DES), RC2, RC4 и AES (Advanced Encryption Standard — современный стандарт шифрования).

## XOR-шифрование

Одной из простейших реализаций симметричного алгоритма является XOR-шифрование.

Предположим, что Алиса (**A**) хочет переслать Бобу (**B**) конфиденциальное сообщение **X**, являющееся некоторой последовательностью байт. При этом у обоих сторон есть известный только им идентичный ключ **Y** (последовательность байт длиной не менее длины сообщения). Тогда Алиса, применяя побайтово операцию **XOR (X, Y)**, получит **Z** — зашифрованное сообщение, которое сможет переслать Бобу по открытому каналу связи. Для получения исходного сообщения **X** Бобу надо только провести операцию **XOR** над сообщением **Z** и ключом **Y**.

В криптографии такой метод (проведение операции «исключающее или» над сообщением и ключом) называется **гаммированием**, а персонажи с именами Алиса, Боб и выступающие обычно в качестве злодеев Ева и Чак используются для анализа и иллюстрирования работы алгоритмов.



Математически доказано (К. Шэннон, 1948), что при условии одноразового использования «статистически хороших» ключей длиной не менее длины входящего сообщения этот простой шифр (гаммирование) будет абсолютно стойким. Это означает, что противник никогда не сможет извлечь никакой полезной информации из перехвата любого количества зашифрованных сообщений.

Однако использование такой крипtosистемы технически довольно трудоемко: для обмена сообщениями между двумя сторонами их надо обеспечить большим количеством секретных идентичных наборов длинных последовательностей случайных чисел. Это обходится довольно дорого, поэтому для практических целей обычно используют алгоритмы с ограниченной длиной ключа и неоднократным его использованием. В этом случае для осложнения раскрытия шифра методом частотного анализа в алгоритмах шифрования используются многократно повторяющиеся методы перестановки частей сообщения и гаммирования с ключом.

Так устроен наиболее известный (но уже устаревший) симметричный алгоритм шифрования — DES. На смену ему пришел алгоритм AES (Advanced Encryption Standard) — симметричный алгоритм блочного шифрования, принятый в качестве стандарта шифрования правительством США в 2002 году.

### **4.13.3. Методы шифрования, основанные на алгоритме DES**

Метод DES (Data Encryption Standard) был разработан компанией IBM по заказу правительства США в 1974 году и затем утвержден в качестве стандарта. DES является примером так называемого «блочного шифра», то есть такого, при применении которого открытый текст разбивается на блоки, состоящие из равного количества бит (обычно 64–256), которые затем обрабатываются раздельно.

В DES существует несколько режимов (фактически разновидностей алгоритма), наиболее простой из них это — ECB (Electronic Code Book — «электронная кодовая книга») — режим простой замены. С особенностями реализации прочих режимов, а именно:

- CBC (Cipher Block Chaining) — режим сцепления блоков;
- CFB (Cipher Feed Back) — режим обратной связи по шифротексту;
- OFB (Output Feed Back) — режим обратной связи по выходу;

Можно разобраться самостоятельно, зная алгоритм ECB.

#### **Алгоритм ECB**

Первым шагом при шифровании сообщения является разбиение его на блоки фиксированной длины. Далее в рассмотрении алгоритма будем считать, что длина блока и длина ключа ( $k$ ) — 64 бита. Особенностью алгоритма ECB является то, что кодирование полученных блоков осуществляется абсолютно независимо, и в отличие от прочих режимов DES, результат шифрования предыдущих блоков не используется при обработке всех последующих.

Вторым шагом алгоритма является перестановка отдельных битов блока в соответствии с некоторой (стандартной для данного алгоритма) таблицей перестановки. Цель такой перестановки (как собственно и перестановок, осуществляемых в методе далее) — скрыть избыточность, регулярность языка и осложнить частотный анализ.

Далее 64-битный блок разбивается на два 32-битных блока L и R, над которыми проводится последовательно 16 раундов преобразований, каждый из которых состоит из следующих действий:

- перестановка субблоков;
- операция «исключающего или» над одним из субблоков и некоторой функцией  $f()$ , которая называется функцией Фейстеля и в качестве аргумента получает ключ кодирования и преобразуемый субблок: результат равен XOR (блок,  $f(\text{блок}, \text{ключ})$ ).

Сама функция Фейстеля также является последовательностью операций:

- расширения входных данных (удвоения части данных — введение избыточности);
- перестановки отдельных бит;
- XOR результата с ключом;
- повторной перестановки бит.

Эти перестановки функции Фейстеля осуществляются в соответствии со стандартными таблицами алгоритма (называемыми S-блоками).

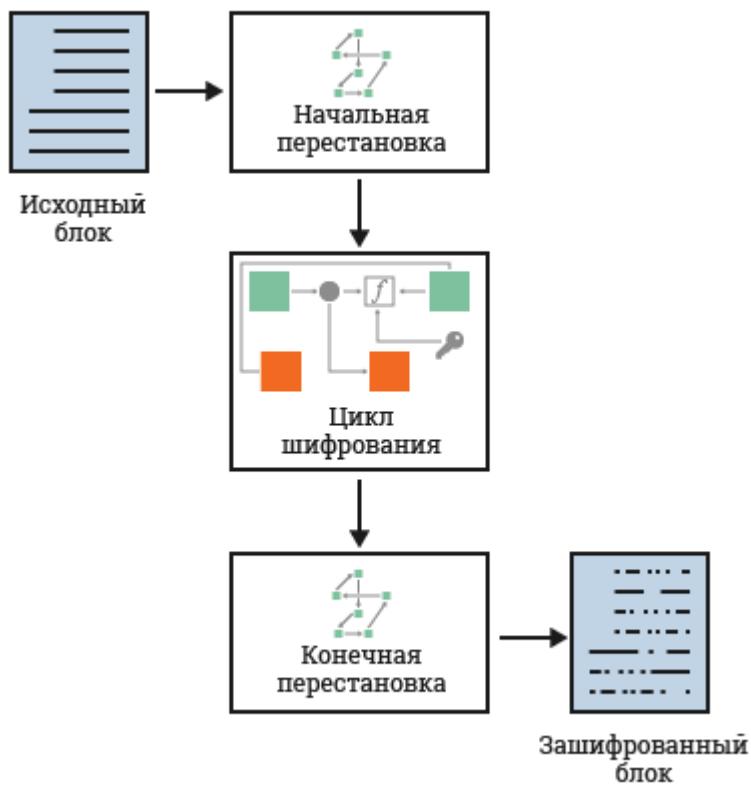


Рис. 4.61.

Создатели метода, отчасти опытным, отчасти аналитическим путем подобрали такие таблицы перестановки (и соответствующие функции Фейстеля), что данный алгоритм с одной стороны был достаточно быстрым, а с другой стороны надежным.

Так для большинства ключей длиной 64 бита (исключая откровенно слабые, типа 111... или симметричные типа 01FE-01FE-...) вскрытие шифрованного сообщения было практически невозможной задачей вплоть до 90-х годов прошлого века. В 1998 году, используя суперкомпьютер стоимостью 250 тыс. долларов, сотрудники RSA Laboratory «взломали» утвержденный правительством США алгоритм шифрования данных (DES) менее чем за три дня.

В дальнейшем развитие компьютеров привело к тому, что взлом подобных шифров методом полного перебора стал возможен за время порядка нескольких дней вычислений. В связи с этим метод DES был усовершенствован — появились методы с увеличенной длиной ключа (до 112 и 168 бит), применением последовательно трех шифрований с разными ключами и еще другие варианты.

### Упражнение 4.13.1

В Java (и Android) для шифрования применяются классы пакета `javax.crypto`. Данный пакет входит в состав JDK (и Android SDK). Класс `javax.crypto.Cipher` реализует все базовые функции всех распространенных криптографических алгоритмов для шифрования данных, включая AES, RSA, DES и нескольких других.

Для того чтобы создать экземпляр такого класса, используется статистический метод `Cipher.getInstance`. Данный метод в качестве параметра получает имя криптографического шифрующего алгоритма, например:

```
Cipher cipher = Cipher.getInstance("DES");
```

После этого нужно настроить экземпляр класса `javax.crypto.Cipher` с указанием, в каком режиме он должен работать: в режиме дешифрования или шифрования: `cipher.init(Cipher.ENCRYPT_MODE, key);`

или

```
cipher.init(Cipher.DECRYPT_MODE, key);
```

Параметр key — это ключ шифрования криптографического алгоритма DES.

Параметр имеет тип javax.crypto.SecretKey и в общем случае его можно создать следующим образом:

```
javax.crypto.KeyGenerator SecretKey key = KeyGenerator.getInstance("DES").generateKey()
```

В алгоритме DES используется симметричное шифрование, а потому для всех операций применяется один и тот же ключ или «master key». Настройка ключа происходит следующим образом.

На основе ключа в виде текста формируется байтовый массив, который может быть использован алгоритмом DES для шифрования данных:

```
DESKeySpec dks = new DESKeySpec(key.getBytes());
```

Класс DESKeySpec предоставляет исходные данные для генерации ключа DES на основе параметра key — секретного ключа алгоритма DES.

После данной операции на основе байтов ключа происходит создание объекта класса SecretKeyFactory, который представляет фабрику для создания секретных ключей:

```
SecretKeyFactory skf = SecretKeyFactory.getInstance("DES");
```

После этого объект класса SecretKeyFactory затем используется для создания объекта класса SecretKey:

```
SecretKey desKey = skf.generateSecret(dks);
```

который затем передается в объект класса javax.crypto.Cipher, который, в свою очередь, непосредственно производит шифрование или же дешифрование информации.

Класс CipherInputStream оборачивает входной поток (InputStream) и объект класса Cipher так, чтобы методы read() возвращали данные, которые были прочитаны из базового входного потока (InputStream) и обработаны шифром.

Например, если объект класса Cipher был создан для дешифрования информации, то CipherInputStream прочитает данные и попытается расшифровать их прежде, чем вернуть.

```

private String code(String input, String key) throws IOException,
        InvalidKeyException, NoSuchAlgorithmException,
        InvalidKeySpecException, NoSuchPaddingException {
    // пароль для DES шифрования должен быть как минимум 8 символов.
    // Обработайте ошибку так, чтобы пользователь мог использовать
    // и более короткие пароли.
    DESKeySpec dks = new DESKeySpec(key.getBytes());
    SecretKeyFactory skf = SecretKeyFactory.getInstance("DES");
    SecretKey desKey = skf.generateSecret(dks);
    // DES/ECB/PKCS5Padding for SunJCE
    Cipher cipher = Cipher.getInstance("DES");
    cipher.init(Cipher.ENCRYPT_MODE, desKey);
    CipherInputStream cis = new CipherInputStream(new ByteArrayInputStream(
            input.getBytes()), cipher);
    ByteArrayOutputStream output = new ByteArrayOutputStream();
    byte[] buffer = new byte[64];
    int numBytes;
    while ((numBytes = cis.read(buffer)) != -1) {
        output.write(buffer, 0, numBytes);
    }
    // Приводим все байты в кодировку, в которой используются только
    // латинские буквы, цифры и некоторые знаки препинания.
    return Base64.encodeToString(output.toByteArray(), Base64.DEFAULT);
}

private String decode(String input, String key) throws IOException,
        InvalidKeyException, NoSuchAlgorithmException,
        InvalidKeySpecException, NoSuchPaddingException {
    // Выделите создание секретного ключа в отдельную
    // функцию для уменьшения дублирования кода.
    DESKeySpec dks = new DESKeySpec(key.getBytes());
    SecretKeyFactory skf = SecretKeyFactory.getInstance("DES");
    SecretKey desKey = skf.generateSecret(dks);
    // DES/ECB/PKCS5Padding for SunJCE
    Cipher cipher = Cipher.getInstance("DES");
    cipher.init(Cipher.DECRYPT_MODE, desKey);
    CipherInputStream cis = new CipherInputStream(new ByteArrayInputStream(
            Base64.decode(input, Base64.DEFAULT)), cipher);
    // Выделите копирование содержание потока в
    // строку в отдельный метод.
    ByteArrayOutputStream output = new ByteArrayOutputStream();
    byte[] buffer = new byte[64];
    int numBytes;
    while ((numBytes = cis.read(buffer)) != -1) {
        output.write(buffer, 0, numBytes);
    }
    return output.toString();
}

public void processCypher(View context) throws InvalidKeyException,
        NoSuchAlgorithmException, InvalidKeySpecException,
        NoSuchPaddingException, IOException {
    String input = ((EditText) findViewById(R.id.input_text)).getText()
}

```

```
        .toString();
Log.d("processCypher", "Coding: " + input);
String password = ((EditText) findViewById(R.id.password)).getText()
        .toString();
Log.d("processCypher", "It is not good idea to log password."
        + " Any application can get access to it.");
TextView output = ((TextView) findViewById(R.id.secret_text));
String secretText = code(input, password);
output.setText(secretText);
if (!input.equals(decode(secretText, password))) {
    // Расшифровка текста приводит к другому результату.
    output.setText("Internal error while coding.");
}
}
```

Программа требует ввода пароля как минимум из 8 символов. Исправьте самостоятельно возникающую ошибку при вводе меньшего числа символов.

#### **4.13.4. Асимметричный алгоритм шифрования**

Начало асимметричным шифрам было положено в работе «Новые направления в современной криптографии» Уитфилда Диффи и Мартина Хеллмана, опубликованной в 1976 году.

Находясь под влиянием работы Ральфа Меркле о распространении открытого ключа, они предложили метод получения секретных ключей, используя открытый канал. Этот метод экспоненциального обмена ключей, который стал известен как обмен ключами Диффи—Хеллмана, был первым опубликованным практическим методом для установления разделения секретного ключа между заверенными пользователями канала. В 2002 году Хеллман предложил называть данный алгоритм «Диффи—Хеллмана—Меркле», признавая вклад Меркле в изобретение криптографии с открытым ключом. Эта же схема была разработана Малькольмом Вильямсоном в 1970-х, но держалась в секрете до 1997 года. Метод Меркле по распространению открытого ключа был изобретен в 1974 и опубликован в 1978 году, его также называют загадкой Меркле.

Как уже было сказано, при использовании асимметричных алгоритмов шифрования для преобразования открытого сообщения в закрытое и обратно используются разные ключи.

Один из ключей используется в качестве открытого. Как правило, специальный (сертификационный) центр публикует открытый ключ в специальном документе (сертификате владельца). Закрытый ключ держится в тайне и никогда никому не открывается. Эти ключи работают в паре: один ключ используется для запуска противоположных функций второго ключа. Так, если открытый ключ используется, чтобы шифровать данные, то расшифровать их можно только закрытым ключом. Если данные шифруются закрытым ключом, то открытый ключ должен это расшифровывать. Такая взаимосвязь позволяет:

- любому пользователю получить открытый ключ по назначению и использовать его для шифрования данных, расшифровать которые может только пользователь, у которого есть закрытый ключ;
- если данные шифруются с использованием закрытого ключа, каждый может расшифровать данные, используя соответствующий открытый ключ.

Пусть Алиса и Боб хотят безопасно обменяться информацией. Для осуществления обмена должен быть определен протокол действий (см. рис. 4.62).

1. Алиса и Боб согласовывают крипtosистему с открытыми ключами.
2. Боб посыпает Алисе свой открытый ключ.
3. Алиса шифрует свое сообщение и отправляет его Бобу.
4. Боб дешифрует сообщение Алисы с помощью своего закрытого ключа.

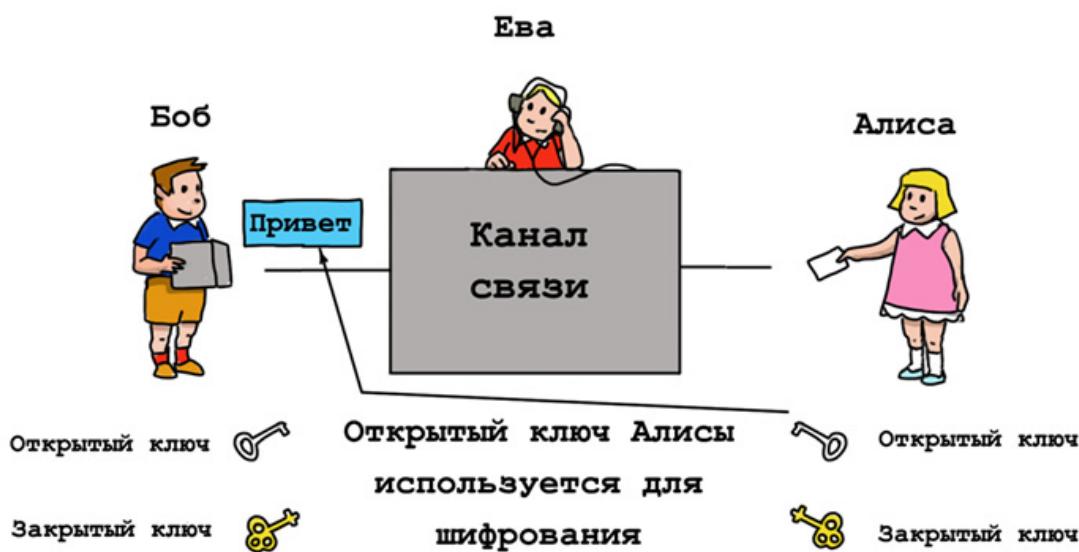


Рис. 4.62.

Теперь, в отличие от симметричного шифрования, Алисе и Бобу не нужно тайно договариваться о ключе и решать проблему его передачи. Боб публикует свой открытый ключ, например, в газете, а Ева, подслушивающая весь протокол, уже не может расшифровать сообщение, так как у нее нет закрытого ключа Боба.

Самый распространенный алгоритм, который используется при шифровании с асимметричными ключами, — **RSA** (назван в честь разработчиков Rivest, Shamir, Adleman).

Алгоритмы крипtosистемы с открытым ключом можно использовать:

- как самостоятельное средство для защиты передаваемой и хранимой информации;
- как средство распределения ключей (обычно с помощью алгоритмов крипtosистем с открытым ключом распределяют ключи, малые по объему, а саму передачу больших информационных потоков осуществляют с помощью других алгоритмов);
- как средство аутентификации пользователей.

Преимущества асимметричных шифров перед симметричными:

- не нужно предварительно передавать секретный ключ по надежному каналу;
- только одной стороне известен ключ шифрования, который нужно держать в секрете (в симметричной криптографии такой ключ известен обеим сторонам и должен держаться в секрете всеми);
- в больших сетях число ключей в асимметричной крипtosистеме значительно меньше, чем в симметричной.

## 4.13.5. Алгоритм RSA

Однако встает вопрос: как создать такой алгоритм и такую пару ключей, чтобы:

- с помощью открытого ключа 1 можно было надежно зашифровать любое сообщение;
- расшифровать его можно было только с помощью закрытого ключа 2;
- никаким образом нельзя было получить ключ 2 из ключа 1.



Для дальнейшей работы нам потребуется вспомнить некоторые термины теории чисел.

**Делитель:** если положительное целое число X делится нацело на целое N, то N — делитель числа X. Так, делителем числа 9 являются 1, 3 и 9. Вообще любое число (кроме 1) всегда имеет минимум два делителя — себя и единицу, однако абсолютное большинство чисел имеет больше делителей.

Здесь и далее мы рассматриваем только натуральные, то есть целые положительные числа.

**Простое число:** число Y будет простым, если не имеет иных делителей, чем себя самого и 1. Пример простых чисел: 3, 5, 7, 11.

**Взаимно простые числа:** два числа будут называться взаимно простыми, если они не имеют общих делителей, кроме 1. Понятно, что два различных простых числа взаимно прости. А вот пример взаимно простых чисел, не являющихся при этом простыми: 9 и 8, 55 и 21.

И, на всякий случай, напоминаем, запись  $X = p \pmod{N}$  или  $X \bmod N = p$  означает, что при делении нацело X на N остаток будет равен p.

Для решения такой задачи используют так называемые «однонаправленные» (one-way) функции, то есть такие, для которых  $y=f(x)$  вычисляется достаточно быстро и легко, а вот найти x, зная значение y и саму функцию  $f()$  — невозможно. Математики пока сомневаются, что идеальная однонаправленная функция вообще существует, но для практических целей в качестве такой функции пока вполне можно применять, например, функцию дискретного возведения в степень:  $y = a^x \pmod{p}$ . Если чуть подробнее, то это возведение в степень x достаточно большого целого числа a с последующим получением остатка от целочисленного деления на p. Даже без дополнительных условий и математического обоснования понятно, что

- небольшие изменения x в такой функции приведут к существенным изменениям y;
- определить исходное x по y или очень сложно или вообще невозможно.

Действительно, на текущий момент неизвестен (возможно, вообще не существует) алгоритм быстрого вычисления обратной функции (она называется дискретным логарифмом), особенно при дополнительных ограничениях, накладываемых на значения a, x, p: a должно быть больше 1 и меньше p;  $1 < x < (p-1)$ ; a p должно быть большим простым числом. Но если нет быстрого алгоритма решения обратной задачи  $y = a^x \pmod{p}$ , то, может быть, вычисление x по y можно решить простым перебором? Оказывается, что нет: для достаточно больших a и p это требует слишком больших вычислительных мощностей и слишком много времени — порядка нескольких лет для современных суперкомпьютеров.

Однако пока непонятно, как пусть даже идеальная однонаправленная функция может помочь в шифровании. Некоторые задачи такая функция действительно успешно решает. Представим, например, задачу по идентификации пользователей при входе в операционную систему компьютера по паролю. Первоначально — при появлении нового пользователя — мы просим его придумать пароль, преобразуем его с помощью однонаправленной функции и в дальнейшем храним в служебном файле только результат вычислений. При следующей

попытке входа пользователь вновь введет пароль, который вновь будет преобразован нашей функцией. Если свежеполученный результат совпадет с данными, хранящимся в файле, пароль верен. То есть фактически, с помощью односторонней функции, мы сумели создать секретный хеш-код для хранения пароля. Причем даже прямой доступ злоумышленника к файлу с хеш-кодами никак не сможет ему помочь, так как получить из результата выполнения функции ее параметр он не сможет (функция односторонняя), а для входа нужен только пароль — хеш-код не годится.

Такое применение весьма полезно, но задачу безопасного обмена сообщениями между Бобом и Алисой, видимо, описанная выше односторонняя функция решить не может. И действительно, для шифрования применяется модифицированная односторонняя функция, так называемая «односторонняя функция с потайным ходом» (trapdoor function).

Односторонней функцией с потайным ходом называют такую функцию  $y = fz(x)$ , что: 1) зная  $y$ , невозможно (или технически очень трудно) получить  $x$ ; 2) зная значение  $z$ , получить из известного значения  $y$  значение  $x$  возможно.

Вариантов односторонних функций с потайным ходом, как ни странно, уже найдено довольно много. Ни одна из них не является идеальной, так как, используя неограниченные вычислительные мощности неограниченное время, простым перебором можно найти все решения для любой из этих функций. Однако, используя обычные ограничения на сложность исходных параметров, можно добиться того, что задача поиска исходного значения решалась бы за неприемлемо большое время (годы вычислений). Первым предложенным вариантом данного класса функций является уже упомянутый выше алгоритм RSA.

В качестве односторонней функции для преобразования исходного сообщения в зашифрованное используют все ту же функцию дискретного возведения в степень:  $y = xe \pmod{N}$ . Используется здесь эта функция чуть иначе, чем ранее: исходное сообщение здесь само возводится в степень, а не находится в показателе степени, но остается неясным, как в этой функции создан «потайной вход»? Оказывается, он сформирован с помощью специального подбора параметров  $e$  и  $N$ .

Рассмотрим реализацию алгоритма.

1. Возьмем достаточно большие простые числа  $p$  и  $q$ .
2. Вычислим произведение этих чисел:  $n = p \times q$ .
3. Для этого произведения вычислим функцию Эйлера  $f(n)$ :  $m = (p - 1) \times (q - 1)$ .
4. Найдем число  $d$ , взаимно простое с  $m$ .
5. Найдем число такое, что  $e \times d = 1 \pmod{m}$ .

Пара  $e$  и  $n$  будут открытым ключом алгоритма, а пара  $d$  и  $n$  — закрытым ключом.

Процедура шифрования будет заключаться в:

- разбиении исходного сообщения на блоки такой длины, чтобы блоку можно было сопоставить (преобразовать в) число  $x < n$ ;
- последовательном применении к блокам функции шифрования  $E(x)$  (Encryption):  $y = xe \pmod{n}$ .

Для дешифрования достаточно применить к полученному закодированному сообщению ту же функцию, но с показателем  $d$  —  $D(y)$  (Decryption):  $x = yd \pmod{n}$ .



Покажем, что алгоритм будет работать, последовательное выполнение  $y = E(x)$  и  $D(y)$  даст нам действительно  $x$ , то есть:  $D(E(x)) = x$ , или, используя конкретные функции  $x = (xe \pmod{n})d \pmod{n}$ . Для доказательства необходимо сослаться на несколько определений и теорем

из теории чисел.

**Функцией Эйлера** от числа  $n$   $f(n)$  называется количество натуральных чисел, меньших, чем  $n$  и взаимно простых с ним. Понятно, что для простого числа  $X$  функция Эйлера будет равна  $X-1$ , ведь все числа, меньшие  $X$ , будут взаимно просты с ним. Также можно доказать, что функция Эйлера имеет такое свойство, как мультипликативность: если  $X = M * N$ , то функция Эйлера от  $X$ :  $f(X) = f(M) * f(N)$ ;

а для простого числа  $p$  и натурального  $n$ :  $f(pn) = pn - pn-1 = pn-1(p-1)$ .

**Малая теорема Ферма** (в отличие от большой теоремы Ферма, эта может быть доказана и даже сравнительно несложно) звучит так: если  $p$  — простое число, а  $X$  — целое число, не делящееся на  $p$ , то  $X(p-1)-1$  будет нацело делится на  $p$ . Иными словами:  $X(p-1) \equiv 1 \pmod{p}$ .

**Лемма Евклида:** если  $a, b$  и  $d$  — натуральные числа и  $d$  делится нацело на произведение  $a$  и  $b$ , то  $d$  делится на  $a$  или на  $b$ . Следствие леммы: если  $a \equiv M \pmod{p}$  и  $a \equiv M \pmod{q}$ , то  $a \equiv M \pmod{p*q}$ .

А теперь давайте рассмотрим, как именно все это работает в алгоритме RSA.

Первыми шагами алгоритма, как описано выше, является подбор двух больших простых числа  $p$  и  $q$  и вычисление их произведения  $N = p * q$ . Для этого произведения, которое в дальнейшем называется «модулем», вычисляем функцию Эйлера  $f(N) = M = f(p)*f(q) = (p-1)*(q-1)$  (см. свойства функции Эйлера для простых чисел и свойство мультипликативности).

На следующем шаге мы должны выбрать число  $e$ , которое должно быть  $> 1$ , но меньше  $M$  и взаимно простое с  $M$ , и дополнительно найти такое число  $d$ , что для него  $e*d \equiv 1 \pmod{M}$ . (Отметим, что этот шаг кажется сложным, но для него существует сравнительно простой алгоритм — немного измененный алгоритм Евклида.)

Теперь докажем, что  $x \equiv (xe \pmod{n})d \pmod{N}$ .

Сначала покажем, что  $xed \equiv x \pmod{p}$  и  $xed \equiv x \pmod{q}$ . Рассмотрим два взаимодополняющих варианта: а)  $x \equiv 0 \pmod{p}$  и б)  $x \not\equiv 0 \pmod{p}$ .

а) Если  $x \equiv 0 \pmod{p}$ , значит  $x = i * p$ , где  $i$  — это некоторое целое число. Тогда  $xed = (i*p)ed = p * xed - 1 * ied$ . Значит  $xed \equiv 0 \pmod{p}$ , как и  $x \equiv 0 \pmod{p}$ . То есть  $xed \equiv x \pmod{p}$ .

б) Если  $x \not\equiv 0 \pmod{p}$ . Это означает,  $x$  не делится нацело на  $p$  и, значит, согласно малой теореме Ферма:  $x(p-1) \equiv 1 \pmod{p}$ . Преобразуем исходное выражение  $xed \pmod{p}$ . Поскольку  $e$  и  $d$  подобраны так, чтобы  $e*d \equiv 1 \pmod{M}$ , то  $e*d-1 \equiv M*k$ , где  $k$  — некоторое целое число, или, иначе говоря, вспомнив, что  $M$  — это функция Эйлера для предварительно выбранных нами  $p$  и  $q$ :  $e*d-1 \equiv M*k \equiv (p-1)*(q-1)*k$ . Значит,  $xed$  можно преобразовать  $xed \equiv xed - 1 * x \equiv x(p-1)*(q-1)*k * x \equiv (x(p-1))(q-1)*k * x$  или по модулю  $p$ :  $xed \pmod{p} \equiv (x(p-1))(q-1)*k \pmod{p} * x \pmod{p}$ .

Но, как выше было отмечено, мы рассматриваем вариант  $x \not\equiv 0 \pmod{p}$ , откуда согласно малой теореме Ферма:  $x(p-1) \equiv 1 \pmod{p}$ . То есть выражение  $(x(p-1))(q-1)*k \pmod{p} * x \pmod{p}$  можно преобразовать к  $(1)(q-1)*k \pmod{p} * x \pmod{p} = x \pmod{p}$ .

Таким образом, мы доказали, что  $xed \equiv x \pmod{p}$ . Аналогично  $xed \equiv x \pmod{q}$ . Следовательно, в соответствии со следствием из леммы Евклида:  $xed \equiv x \pmod{p*q}$ .

Таким образом, мы доказали, что  $x \equiv (xe)d \pmod{n} = (xd)e \pmod{N}$ , то есть описанный выше алгоритм RSA действительно верный.

#### Упражнение 4.13.2

Зашифруем и расшифруем сообщение «DAD» по алгоритму RSA.

1. Выберем  $p = 11$  и  $q = 13$ .
2. Определим  $n = 11 * 13 = 143$ .

3. Вычислим функцию Эйлера  $f(n)$ :  $m = (p - 1) * (q - 1) = 10 * 12 = 120$ .
4. Следовательно,  $d$  будет равно, например, 7 ( $120 \bmod 7 = 1$ ).
5. Выберем число  $e$  по следующей формуле:  $(e * 7) \bmod 120 = 1$ . Значит  $e$  будет равно, например, 103 ( $(103 * 7) \bmod 120 = 1$ ).
6. Представим шифруемое сообщение как последовательность чисел в диапазоне от 0 до 32 (не забывайте, что кончается на  $n-1$ ). Буква, A = 1, B = 2, C = 3.

DAD

4 1 4

Теперь зашифруем сообщение, используя открытый ключ  $\{e, n\} = \{103, 143\}$

$$C1(D) = (4^{\wedge} 103) \bmod 143 = 108$$

$$C2(A) = (1^{\wedge} 103) \bmod 143 = 1$$

$$C3(D) = (4^{\wedge} 103) \bmod 143 = 108$$

Теперь расшифруем данные, используя закрытый ключ  $\{d, n\} = \{7, 143\}$ .

$$M1 = (108^{\wedge} 7) \bmod 143 = 171382426877952 \bmod 143 = 4 (D)$$

$$M2 = (1^{\wedge} 7) \bmod 143 = 1 (A)$$

$$M3 = (108^{\wedge} 7) \bmod 143 = 4 (D)$$

## **Задание 4.13.1**

Самостоятельно добавить в упражнение 4.13.1 еще одну кнопку для дешифрования сообщения.

В качестве игрового элемента учащихся можно разбить на пары, один из участников которой составляет и отправляет зашифрованное сообщение, а другой на своем рабочем месте расшифровывает его; затем участники пары меняются ролями. Ученики заинтересованы добавить эту кнопку, чтобы расшифровать полученное сообщение.

## **Задание 4.13.2**

Изменить программу из упражнения таким образом, чтобы использовалось RSA-шифрование. Функция кодирования и декодирования может быть объединена в одну, так как процессы шифрования и дешифрования в RSA абсолютно симметричны.

# Литература

1. Fermat's Little Theorem and RSA Algorithm:  
[\(https://exploringnumbertheory.wordpress.com/2013/07/08/fermats-little-theorem-and-rsa-algorithm/\)](https://exploringnumbertheory.wordpress.com/2013/07/08/fermats-little-theorem-and-rsa-algorithm/)
2. What is the relation between RSA & Fermat's little theorem?
3. Алгоритм шифрования RSA  
<http://www.e-nigma.ru/stat/rsa/>
4. Википедия:
  - RSA <https://ru.wikipedia.org/wiki/RSA>
  - Функция Эйлера
  - Малая теорема Ферма
5. Асимметричное шифрование. Как это работает? <http://intsystem.org/1120/asymmetric-encryption-how-it-work/>
6. RSA алгоритм шифрования с открытым ключом  
<http://www.paveldvlip.ru/algorithms/rsa.html>