Программирование на Java

Пособие по курсу

Университет ИТМО
Центр Авторизованного Обучения IT-Технологиям

Оглавление

1. Введение

Это учебное пособие написано специально для курса «программирование на Java». Оно не является учебником, а является набором примеров и вспомогательным средством для практических занятий в рамках курса.

Для глубокого изучения языка рекомендуется воспользоваться специализированной литературой, список которой можно найти в конце пособия.

Также во второй части пособия приведены несколько примеров программ (или частей программ). Часть из них достаточно сложны, чтобы попасть на практические занятия этого курса, но они иллюстрируют возможности и синтаксис языка, использование стандартной библиотеки Java. Некоторые из них встречаются на собеседованиях в IT-компании.

Сегодня Java используется на 97% корпоративных настольных ПК, используется в 3 миллиарда мобильных телефонов, входит в комплект поставки 100% всех проигрывателей дисков Blu-ray. Сегодня 9000000 программистов пишут код на Java. Это больше, чем население Нью-Йорка или Лондона, больше чем суммарное население Санкт-Петербурга и Ленинградской области. Это один из самых популярных и распространённых языков программирования.

Этот курс — введение в огромный мир Java.

2. Что понадобится

Для программирования на Java понадобится:

- Компьютер с операционной системой по вкусу
- DK (Java Developer Kit)
- IDE

2.1. JDK (Java Developer Kit)

Java Developer Kit — это бесплатный набор утилит для разработчика. Строго говоря, для программирования на Java этого достаточно. В набор входят компилятор, виртуальная машина, стандартные библиотеки, документация, профайлер и так далее.

Загрузить его можно с сайта oracle.com/java. В этом пособии используется JDK от компании Oracle, хотя существуют и другие, например OpenJDK (проект по созданию полностью совместимого Java Development Kit, состоящего исключительно из свободного и открытого исходного кода).

2.2. IDE

Конечно, сегодня никто не пишет код в блокноте. Давно стандартом стало использование интегрированной среды разработки (англ. Integrated development environment — **IDE**). Для языка Java самыми популярными IDE являются IntelliJ IDEA, NetBeans и Eclipse.

Такие системы упрощают компиляцию и запуск приложений, отладку, поиск по коду, форматирование. С ними легко можно генерировать код из шаблонов, реструктуризировать его и так далее.

3. Компиляция

Java — компилируемый язык программирования. Исходный код Java компилируются в специальный байт-код, который не может быть выполнен «как есть», в отличие от программ, написанных на C++ или Delphi (и других компилируемых языков). Байт-код исполняется виртуальной машиной, которая для каждой платформы своя. Схематично это изображено на схеме 1:

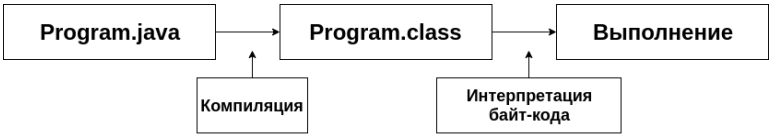

Схема 1: компиляция программы

Это означает, что написанный один раз код, может быть выполнен на любой платформе без перекомпиляции.

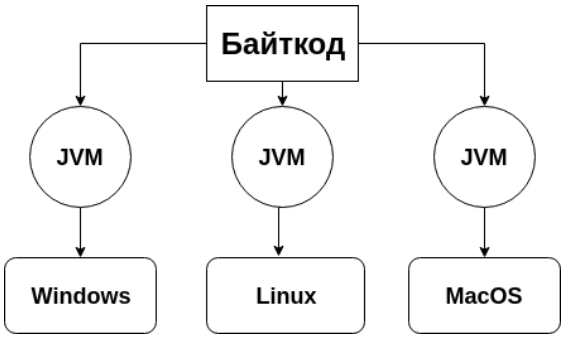

Схема 2: кроссплатформенность

4. Hello, World!

Рассмотрим процесс компиляции и запуска программы на Java без IDE.

Для этого, напишем самую простую программу:

```
public class ExampleProgram {
    public static void main(String[] args) {
        System.out.println("I'm a Simple Program");
    }
}
```

В этой программе объявляется класс ExampleProgram, в котором объявляется один метод ***main***. Это необходимая для компиляции и запуска синтаксическая конструкция. Дальше в пособии, до рассмотрения объектно-ориентированного программирования в Java, предполагается, что весь код пишется внутри метода ***main***.

Сохранив этот код в файле ExampleProgram.java можно выполнить в командной строке:

```
javac ExampleProgram.java
```

javac.exe можно найти в каталоге с JDK.
После запуска компилятор javac сгенерирует файл ExampleProgram.class, содержащий байт-код программы.
Для запуска теперь можно написать:

```
java ExampleProgram
```

После этого на консоль будет выведена надпись «I'm a Simple Program».
Несмотря на простоту компиляции и запуска программы, в процессе изучения языка рекомендуется использовать интегрированную среду разработки, чтобы компилировать и запускать программы нажатием одной кнопки на клавиатуре, и тем самым экономя своё время.

5. Переменные

В Java существуют примитивные типы и объектные. Пока что остановимся на первых. Чтобы объявить переменную необходимо указать её тип и имя:

```
int value;
```

Каждая логическая конструкция должна заканчиваться точкой с запятой.
Можно объявить переменную и сразу присвоить ей значение:

```
int value = 100;
```

5.1. Сильная типизация

Переменные (или, например, возвращаемые значения методов) связаны с типом в момент объявления и тип не может быть изменён позже. То есть, в Java должен быть объявлен тип при объявлении переменной:

```
int value;
```

Переменной **value** нельзя присвоить значение, отличное от **int**.

5.2. Приведение типов

Java сильно типизированный язык, но иногда необходимо приводить одни типы к другим. Например, чтобы привести тип **double** к **int** нужно написать:

```
int value = (int) 10.5;
```

Перед выражением, которое нужно привести к другому типу в скобках написан тип, к которому нужно привести.
Не все типы приводимы. Например, нельзя привезти тип **boolean** к **int** и наоборот. При этом тип **int** к **double** приводится автоматически.

5.3. Примитивные типы

Тип	Длина (в байтах)	Диапазон или набор значений
boolean	1 в массивах, 4 в переменных	true, false
byte	1	−128..127
char	2	0..216−1, или 0..65535
short	2	−215..215−1, или −32768..32767
int	4	−231..231−1, или −2147483648..2147483647
long	8	−263..263−1, или примерно −9.2·1018..9.2·1018
float	4	-(2-2−23)·2127..(2-2−23)·2127
double	8	-(2-2−52)·21023..(2-2−52)·21023

6. Строки

Чтобы работать со строками следует использовать двойные кавычки (в отличие от char):

```
String name = "Вася";
```

Тип **String** написан с большой буквы. **String** это не ключевое слово, это стандартный класс в Java. Об этом пойдёт речь позднее.

7. Управляющие операторы

В Java программа выполняется последовательно (опустим пока что многопоточность). Управляющие операторы позволяют ветвить программу в зависимости от условий.

7.1. If else

Операторы **if** и **else** позволяют выполнять разные операции в зависимости от условий, проверяемых оператором **if**. Например:

```
if (doubleValue < 100) {
    System.out.println("Yes");
} else {
    System.out.println("No");
}
```

После оператора (это касается и циклов) должно быть или одно выражение, или один блок кода. Блок кода — это совокупность выражений, объединённых фигурными скобками {}.

Операторы сравнения:

Оператор	Использование	Возвращает значение "истинно", если...
>	a > b	a больше b
>=	a >= b	a больше или равно b
<	a < b	a меньше b
<=	a <= b	a меньше или равно b
==	a == b	a равно b
!=	a != b	a не равно b
&&	a && b	a и b истинны
||	a || b	a или b истинно
!	!a	a ложно

Пример использования:

```
if (doubleValue < 100 && integer < 100 || longValue <= 100 && bool)
    System.out.println("Yes");
```

Можно переписать пример иначе:

```
boolean isSomething = doubleValue < 100 && integer < 100 || longValue <= 100;

if (isSomething)
    System.out.println("Yes");
```

Если условие if вложено в другой if, то часто можно заменить вложенность на цепочку if-else. Два примера ниже эквивалентны:

```
if (doubleValue < 100) {
    System.out.println("Yes");
} else {
    if (integer < 100) {
        System.out.println("Yes");
    } else {
        System.out.println("No");
    }
}

//.....

if (doubleValue < 100) {
    System.out.println("Yes");
} else if (integer < 100) {
    System.out.println("Yes");
} else {
    System.out.println("No");
}
```

7.2. Циклы

В Java есть три вида циклов: **for**, **while** и **do while**.

for

```
for (int i = 0; i < 100; i++)
    System.out.println(i);
```

В конструкции for сначала пишется произвольное выражение инициализации цикла, затем — условие продолжения и, наконец, выполняемая после каждой итерации цикла некоторая операция (это не обязательно должно быть изменение счётчика; это может быть правка указателя или какая-нибудь совершенно посторонняя операция).

while

Цикл, который выполняется, пока истинно некоторое условие, указанное перед его началом. Это условие проверяется до выполнения тела цикла, поэтому может быть не выполнено ни одной итерации (если условие с самого начала ложно).

```
int i = 0;
while (i < 100) {
    System.out.println(i);
    i++;
}
```

do while

Цикл, в котором условие проверяется после выполнения тела цикла. Отсюда следует, что в таком цикле блок кода выполняется хотя бы один раз.

```
int i = 0;
do {
    System.out.println(i);
    i++;
} while (i < 100);
```

Три примеры выше — эквивалентны.

continue и break

Внутри цикла можно использовать операторы **continue** и **break**.
continue прерывает текущую итерацию чтобы сразу началась следующая, **break** и вовсе прерывает работу цикла. Два примера:

```
for (int i = 0; i < 100; i++) {
    // если делится на 10, то пропускаем.
    if (i % 10 == 0) {
        continue;
    }
    System.out.println(i);
}
```

```
for (int i = 0; i < 100; i++) {
    // если дошли до 10, то завершаем цикл.
    //В итоге будут выведены числа от 0 до 9
    if (i == 10) {
        break;
    }
    System.out.println(i);
}
```

Еще пример: вывод всех чётных чисел от 0 до 100:

```
for(int i = 0; i < 100; i+=2){ // i = i + 2
    System.out.println(i);
}
```

Более сложные примеры можно посмотреть в конце пособия, например вычисление синуса (страница 70) и поиск наибольшего элемента массиве (страница 71).

8. Массивы

8.1. Объявление массива

Объявление массива похоже на объявление переменной (оно и есть), но с добавлением [].

```
int array1[] = new int[10];
```

Можно и так:

```
int[] array2 = new int[10];
```

Два примера выше эквивалентны.
Можно задать массив, сразу перечислив его содержимое:

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

8.2. Работа с массивом[1]

Для доступа к элементу массива надо использовать его индекс. Следует помнить, что нумерация начинается с нуля.

```
array[1] = 10;
System.out.println(array1[0]); // 0 по умолчанию
System.out.println(array1[1]); // 10
```

1. см. пример про поиск наибольшего элемента в массиве на странице 71.

8.3. Двумерные массивы

В Java допустимы двумерные массивы. Двумерный массив — это просто массив массивов.
Объявить двумерный массив можно так:

```
int[][] array = new int[10][10];
```

При объявлении двумерного массива необходимо указать размер только первого массива:

```
int[][] array = new int[10][];
```

В таком случае можно создавать массивы второго уровня динамически:

```
int[][] array = new int[10][];
for(int i = 0 ; i < array.length; i++){
    array[i] = new int[10];
}
```

Принято считать, что в двумерном массиве первый массив это строки, второй — столбцы. Так как двумерные массивы можно создавать динамически, то массив может не быть квадратным:

```
int[][] array = new int[10][];
for(int i = 0 ; i < array.length; i++){
    array[i] = new int[i]; // Размер ступеньками
}
```

Перебрать все элементы такого массива можно с помощью вложенного цикла[2]:

```
for (int i = 0; i < array.length; i++) {
    // array — массив массивов,
    // innerArray — текущий массив второго уровня
    int[] innerArray = array[i];

    // пробегаем по массиву внутреннему
    for (int j = 0; j < innerArray.length; j++) {
        // без перевода на новую строку
        System.out.print(innerArray[j]);
    }

    // после каждой строки - перевод на новую строку
    System.out.println();
}
```

Так как массив в примере не был заполнен, то выведены будут только нули:

```
0
00
000
0000
00000
000000
0000000
00000000
000000000
```

2. см. пример про поиск наибольшего элемента в двумерном массиве на странице 71.

9. for each

Существует ещё одна разновидность цикла for — **for each**. Она позволяет «пробегать» все элементы массива (или коллекции) без использования индексов.

```
for(int item : array){ // array — массив
    System.out.println(item); // item — текущий элемент массива
}
```

Предыдущий пример можно переписать иначе, с использованием индексов:

```
for(int i = 0; i <  array.length; i++){
    System.out.println(array[i]);
}
```

10. break с меткой

Представим, что есть два массива и надо определить, есть ли в них пересекающиеся элементы. Напишем код:

```
for(int item : array1){
    for(int item2 : array2){
        if(item == item2){
            System.out.println("Arrays contain values");
        }
    }
}
```

Очевидно, что в этом коде есть проблема. Она заключается в том, что если массивы очень большие, и при этом в самом начале уже нашёлся элемент, который входит в оба массива, то перебор происходит впустую, так как ответ уже известен. Использование внутри блока if оператора break не решает проблему, так как только вложенный цикл будет завершён. Для решения этой проблемы в C++ можно было использовать оператор goto. Java **не поддерживает goto**, но поддерживает break с метками:

```
myLabel : for(int item : array1){
    for(int item2 : array2){
        if(item == item2){
            System.out.println("Arrays contain values");
            break myLabel;
        }
    }
}
```

Можно присвоить метку любому блоку кода, и с помощью break выйти из него:

```
label:{
     ...
    break label;
}
```

11. Switch

Если необходимо проверить переменную на соответствие не одному а нескольким значениям, то вместо использования нескольких выражений if-else следует использовать оператор **switch**:

```
switch (value) {
    case 1:
        System.out.println(1);
    case 2:
        System.out.println(2);
        break;
    default: // Необязательно
        System.out.println(-1);
}
```

В качестве аргумента switch можно использовать примитивные типы **byte, short, char, int, enum** и **String.**

Стоит обратить внимание на использование оператора **break** внутри **switch**. Когда переменная value в **switch** равна значению, указанному в **case**, операторы следующие за **case** будут выполняться до тех пор, пока не будет достигнут оператор **break**.

В примере выше, если value будет равна 1, то на консоль будут выведены числа 1 и 2, а если value равна 2, то то на консоль будет выведено число 2.

Оператор **default** необязателен в блоке switch.

Ещё пример — определение дня недели по номеру:

```
switch (n){ // switch по номеру
    case 1: s = "Понедельник"; break;
    case 2: s = "Вторник"; break;
    case 3: s = "Среда"; break;
    case 4: s = "Четверг"; break;
    case 5: s = "Пятница"; break;
    case 6: s = "Суббота"; break;
    case 7: s = "Воскресенье"; break;
    default: s = null; // Если номер не из промежутка [1,7]
}
```

12. Комментарии

В Java существует три вида комментариев. Два из них пришли в Java из других языков, таких, как C++. Третий используется для создания документации к программам или библиотекам, а также облегчает навигацию по коду.

```
// - текст вашего комментария начнётся сразу за чертами

/*
В этом комментарии может быть сколько угодно строк.
Сколько угодно.
*/
```

13. Условный оператор

Условный оператор — оператор, который состоит из трёх операндов и используется для оценки выражений типа boolean. Цель условного оператора заключается в том, чтобы решить, какое значение должно быть присвоено переменной.

```
int maxSpeed = isCity ? 60 : 90;
//  (выражение) ? значение if true : значение if false
```

14. Объектно-ориентированное программирование

Объектно-ориентированное программирование — методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определённого класса. В каком-то смысле класс это описание структуры данных. Например, у класса User может быть возраст и имя, и класс может быть просто объединением разных данных. Но классы также могут иметь собственное поведение. Или наоборот, не иметь никаких данных, а только лишь описывать поведение. Работа с объектами позволяет создавать программы на более высоком уровне абстракции, чем при использовании, например, процедурных языков программирования.

15. Классы

Класс — это элемент ПО, описывающий абстрактный тип данных и его частичную или полную реализацию. Класс содержит поля, описывающие структуру данных, и методы, которые задают поведение. Не стоит путать объекты и классы. Если проводить аналогию, то описание автомобиля это класс, а конкретные автомобили, которые ездят по городу — это объекты. Не смотря на то, что такой «класс» предполагает одинаковую сущность для всех автомобилей (мотор, размеры, одинаковое поведение и так далее), каждый экземпляр имеет свой уникальный набор характеристик — конкретный цвет, пробег, количество бензина в баке. Также и с объектами в Java. Класс может пониматься как некий шаблон, по которому создаются объекты — экземпляры данного класса. Все экземпляры одного класса созданы по одному шаблону, поэтому имеют один и тот же набор полей и методов.

Чтобы написать свой первый класс, нужно использовать ключевое слово **class**:

```
class Auto{} // Это класс, хоть и пустой.
```

Класс с двумя полями:

```
class User {
    String name;
    int age;
}
```

15.1. Создание объекта

Для создания объекта используется ключевое слово **new**:

```
class User {
    String name;
    int age;
}

public class Main {
    public static void main(String[] args){
        User user = new User();
    }
}
```

Теперь можно использовать объекты так, как нам нужно:

```
public static void main(String[] args) throws Exception {
    User user1 = new User();
    User user2 = new User();

    user1.name = "Вася";
    user1.age = 10;
    user2.name = "Петя";
    user2.age = 12;

    if(user2.age > user1.age)
        System.out.println(user2.name + " старше чем " + user1.name + " на " +
                                                    (user2.age - user1.age) + " года.");
}
```

В результате будет выведено:

```
Петя старше чем Вася на 2 года.
```

15.2. Указатели

Рассмотрим внимательно следующий код:

```
User user = new User();
```

Ранее уже говорилось об объектных и примитивных типах. В данном случае, когда мы пишем “User user”; объявляется переменная типа User, которая называется user. new User() — это вызов конструктора. О конструкторах речь пойдет позже. В данном случае, вызывает пустой конструктор, и в результате создается объект типа User на который ссылается переменная (указатель) user. Можно присвоить переменной user новое значение, и она будет ссылаться на другой объект или в никуда (тогда переменная равна **null**). И при этом, на созданный объект может ссылаться несколько указателей.

Например:

```
User user1 = new User();
user1.name = "Вася";
User user2 = user1;

// Вася
System.out.println(user1.name);

// Вася, так как user2 ссылается на тот же объект, что и user1
System.out.println(user2.name);
```

15.3. Методы

Метод — фрагмент программного кода (подпрограмма), к которому можно обратиться из другого места программы. В программировании разделяют понятие функции и метода. Метод это функция, принадлежащая какому-то классу. В Java нельзя написать функцию вне класса, поэтому речь идет только о методах.

Метод состоит из имени, списка параметров, типа возвращаемого значения, тела метода. Имя метода вместе с параметрами называется сигнатурой метода. В одном классе не может быть двух методов с одинаковой сигнатурой.

```
int sum(int a, int b){
    return a + b;
}
```

В данном примере int — это тип возвращаемого значения, int a, int b — параметры, и всё, что внутри {} — это тело метода. Этот метод просто возвращает сумму двух чисел.

Оператор **return** завершает выполнение метода, в котором он присутствует. Он также возвращает значение. Если метод имеет тип возвращаемого значения **void**, оператор **return** можно опустить. **void** означает, что метод не возвращает никакого значения (например, System.out.println()).

Рассмотрим другой пример:

```
class User {
    String name;
    int age;

    boolean isTeenager(){
        return age > 9 && age < 20;
    }
}
…
public static void main(String[] args) throws Exception {
    User user = new User();
    user.age = 15;
    System.out.println(user.isTeenager());
}
```

Метод isTeenager() не имеет параметров, но возвращает значение вычисленное с помощью поля age. У каждого объекта результат вызова isTeenager() будет отличаться.

15.4. Методы с переменным количеством аргументов

Можно создавать методы с переменным числом аргументов (параметров). Синтаксис выглядит так:

```
void doSome(int... args) {}
```

Такой метод можно вызвать с произвольным количеством аргументов:

```
doSome(); // и без аргументов
doSome(1);
doSome(1, 2, 3, 4);
doSome(1, 2, 3, 4, 5, 6);
```

У такого метода компилятор заменит список аргументов произвольной длины на массив, а при вызове метода будет автоматически упаковывать в массив переданные аргументы. Поэтому внутри метода доступ к аргументам осуществляется через на массив:

```
void doSome(int... v){
    for(int i = 0; i < v.length; i++){
        System.out.println(v[i]);
    }
}
```

Некоторые ограничения

1. Если в списке аргументов больше одного элемента, то аргумент переменной длины должен указываться последним:

```
void doSome(double value, int oneMoreInt, int... v){…}
```

2. Нельзя объявить в одном методе два аргумента переменной длины.

15.5. public static void main(String[] args)

Выполнение программы начинается с метода *main()*. Про «public» и «static» речь пойдет позже, но сейчас уже известно, что метод main ничего не возвращает, называется main и как параметр принимает массив строк. Входной массив содержит аргументы, которые были переданы программе при запуске. То есть, если запустить программу с аргументом, то args будет содержать один элемент — строку «MyParameter».

```
java ExampleProgram MyParameter
```

15.6. return в блоках if, else, for других

Оператор **return** завершает выполнение метода. Выполнение кода внутри циклов или других управляющих конструкций будет прервано:

```
void func(){
    while (true){
        return; // метод будет завершён на первой итерации
    }
}
```

15.7. Конструкторы

Конструктор — это метод, вызываемый при создании объекта. То есть, при использовании ключевого слова **new**.
Объявление конструктора очень похоже на объявление метода, но с двумя отличиями:

1. Тип возвращаемого значения не указывается.
2. Название совпадает с названием класса.

Также существует два ограничения: конструкторы не могут напрямую вызываться (необходимо использовать ключевое слово **new**) и конструкторы не могут иметь модификаторы *synchronized*, *final*, *abstract*, *native* и *static*.

```
class User {
    String name;
    int age;

    public User(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

Теперь уже нельзя просто написать new User(), требуется явно указать параметры:

```
User user = new User("Вася", 20);
```

Внутри конструктора полям присваиваются переданные значения. Стоит обратить внимание на ключевое слово **this**. Внутри класса **this** ссылается на текущий объект этого класса. В примере выше, когда для объекта типа User память уже выделена, **this** ссылается на объект типа User. Можно считать, что это скрытое поле класса, к которому есть доступ только внутри этого класса, и он содержит ссылку на текущий объект.

15.8. Конструктор по умолчанию

Если конструктор не объявлен (и только в этом случае) считается, что класс имеет конструктор по умолчанию. Это пустой конструктор без параметров. То есть два примера ниже эквивалентны:

```
class User {
    String name;
    int age;
}
```

```
class User {
    String name;
    int age;

    User() {
    }
}
```

15.9. Перегрузка методов

Несколько методов одного класса могут иметь одно и то же имя, отличаясь лишь набором параметров. Примером такого метода является метод print в классе PrintStream (экземпляром которого является System.out):

```
void print(boolean b)
void print(char c)
void print(char[] s)
void print(double d)
void print(float f)
void print(int i)
void print(long l)
void print(Object obj)
void print(String s)
```

Не будет являться перегрузкой изменение типа возвращаемого значения. Перегрузкой является только различие параметров.
При вызове метода, компилятор по переданным параметрам определит, какой из методов должен быть вызван.
Если подходящих вариантов несколько, то компилятор выбирает наиболее подходящий метод:

```
void print(double d){
}

void print(int d){
}
…
// будет вызван метод с int. Если бы метода с параметром типа  // int не было, был бы вызван
метод с double
print(10);
```

15.10. Перегрузка конструкторов

Конструкторы, как и методы можно перегружать. Работает это точно также как и с простыми методами. Основная цель такой перегрузки — дать возможность инициализировать объекты поразному. Например, у класса Date в Java есть конструкторы:

```
Date(); // текущее время
Date(int year, int month, int day);
Date(int year, int month, int day, int hour,
           int minute);
Date(int year, int month, int day, int hour,
           int minute, int second);
Date(long milliseconds);
Date(String string);
```

15.11. Сборка мусора (Garbage Collection)

Специальный процесс, называемый сборщиком мусора (англ. garbage collector), периодически освобождает память, удаляя объекты, которые уже не будут востребованы приложением.

В Java невозможно явное удаление объекта из памяти — вместо этого реализована сборка мусора. Объект не будет удалён сборщиком мусора, пока на него указывает хотя бы одна ссылка.

Существуют также методы для инициации принудительной сборки мусора, но не гарантируется, что они будут вызваны исполняющей средой, и их не рекомендуется использовать для обычной работы.

Стоит учитывать, что ссылка на неиспользуемый объект может сохраниться в другом объекте, который используется и становится своеобразным «якорем», удерживающим ненужный объект в памяти. Таким образом, могут «подвисать» целые цепочки объектов.

Неизвестно точно, когда будет запущена следующая сборка мусора. Это зависит от многих факторов, плюс к этому, разные реализации виртуальной машины Java могут использовать разные алгоритмы. Тем не менее, есть механизм, позволяющий узнать, что объект будет вскоре собран сборщиком мусора. Для этого в классе можно описать метод finalize().

15.12. Метод finalize()

```
class myClass{
    protected void finalize()  {
        // todo
    }
}
```

Когда сборщик мусора посчитает, что объект можно удалить, он вызовет у него метод *finalize()*. В этом методе, например, можно может освободить ресурсы, которые уже не требуются.

Деструкторов в Java не существует, а метод *finalize()* ни в коем случае нельзя считать аналогом деструктора. И не следует вызывать его самостоятельно.

15.13. Getter и setter

Во всех примерах, которые разбирались ранее, доступ к полям класса осуществлялся напрямую. Однако в Java так не принято. В будущем мы увидим, что поля класса обычно помечаются как **private**, а доступ к ним осуществляется через специальные методы, которые называются геттеры и сеттеры:

```
class A {
    private int value;

    public int getValue() {
        return value;
    }

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

Одна из причин — это возможность в будущем поменять логику доступа к полю. Например, значение будет считываться из файла. Нужно будет поменять только содержимое метода, а вызовы останутся прежними.

Стоит обратить внимание, что это не является новой синтаксической конструкцией. Это не более чем поля и методы.

Примеры более сложных классов можно посмотреть во второй части пособия. Например, классы описывающие вектор и связный список (страницы 72и 77, соответственно).

16. Strings

Рассмотрим один из встроенных классов для работы с последовательностью символов, то есть со строками. Стоит обратить внимание, что это не более чем обычный класс, который не отличается от всех других классов за исключением того, что его конструктор можно вызвать неявно:

```
String s = "my String";
```

А также можно использовать оператор «+» для соединения двух строк:

```
String s = "my " + "String";
```

Этот класс предоставляет удобный способ работы со строками: создавать их, менять, модифицировать, делать поиск в строке и так далее. В основе этого класса лежит массив символов, но с помощью класса String можно работать со строкой как с отдельной сущностью. Ниже перечислены некоторые из самых важных методов класса String[3]:

3. Примеры работы с классом String можно посмотреть на страницах 73, 74.

char charAt(int index)	Возвращает char со значением равным символу с индексом index.
int compareTo(String anotherString)	Сравнивает указанную строку. Используется при сортировке.
String concat(String str)	Присоединяет строку str к концу родительской строки.
boolean contains(CharSequence s)	Возвращает true, если строка содержит указанную последовательность символов.
boolean endsWith(String suffix)	Проверяет, заканчивается ли строка с указанной строкой.
boolean equals(Object anObject)	Сравнивает строки по значению с учётом регистра.
boolean equalsIgnoreCase(String anotherString)	Сравнивает строки по значению без учёта регистра.
String format(String format, Object... args)	Возвращает отформатированную строку, используя специальный формат строки и аргументы.
byte[] getBytes()	Возвращает представление строки в виде массива байт.
int indexOf(String str)	Возвращает индекс первого символа подстроки.
boolean isEmpty()	Проверяет, пустая строка или нет.
int length()	Возвращает количество символов в строке.
String replace(CharSequence target, CharSequence replacement)	Заменяет одну подстроку другой.
boolean startsWith(String prefix)	Проверяет, начинается ли строка с указанной строки.
String substring(int beginIndex)	Возвращает подстроку, начиная с beginIndex (включительно) до конца строки.
String substring(int beginIndex, int endIndex)	Возвращает подстроку, начиная с beginIndex (включительно) до endIndex (не включительно).
char[] toCharArray()	Преобразует строку в массив символов.
String toLowerCase()	Конвертирует все символы строки в нижний регистр.
String toUpperCase()	Конвертирует все символы строки в верхний регистр.
String trim()	Удаляет пробелы в начале и конце строки.

Программа, которая подсчитывает количество слов в тексте, будет выглядеть так:

```
String text = …
String words[] = text.split(" ");
int wordsCount = words.length;
```

16.1. equals()

В отличие от примитивных типов, таких как int или double, нельзя сравнивать объекты с помощью оператора «==». Например:

```
String s1 = new String("string");
String s2 = new String("string");
System.out.println(s1 == s2); // false!
```

В приведённом выше примере сравниваются переменные s1 и s2 с помощью оператора «==». Результатом сравнения будет false. Дело в том, что когда используется оператор «==», на самом деле сравниваются ссылки. Таким образом, если две ссылки указывают на один объект, результатом сравнения с помощью оператора «==» будет true. Если ссылки указывают на разные объекты, результатом сравнения будет false.

Для корректного сравнения объектов нужно использовать метод *equals()*. В будущем будет рассмотрено, как определять метод *equals()* правильно. Пока что надо запомнить, что строки всегда сравниваются с помощью оператора *equals()* как в примере ниже:

```
String s1 = new String("string");
String s2 = new String("string");
System.out.println(s1.equals(s2));
```

16.2. String — неизменяемый объект

Существует такой шаблон программирования, как «неизменяемый объект». В английском языке этот шаблон называется «immutable object». Неизменяемым называется объект, состояние которого не может быть изменено после создания. Например:

```
class A {
    private int value;

    public A(int value) {
        this.value = value;
    }

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

В этом примере полю value класса A присваивается значение, переданное в конструктор. Других способов изменить это поле нет. Состояние этого объекта никогда не будет изменено после создания. Класс String устроен таким же образом.

Например, когда вызывается метод *replace()* у объекта типа String, сам объект не меняется. Методы, изменяющие строку, возвращают новый экземпляр строки уже с изменениями. То же самое касается и «склеивания» двух строк: каждая «склейка» порождает новый объект. Отсюда следует, что использовать оператор «+» в цикле неэффективно с точки зрения расхода памяти. Для оптимизации этого существуют классы, которые будут рассмотрены ниже.

16.3. StringBuilder

Для эффективного изменения, или создания строк существует класс **StringBuilder**:

```
StringBuilder stringBuilder = new StringBuilder();
```

Теперь с помощью объекта stringbuilder (экземпляра класса **StringBuilder**) можно, например, «склеивать» строки в цикле эффективно:

```
for(int i = 0; i < 10; i++){
    stringBuilder.append(i);
}

System.out.println(stringBuilder.toString());
```

Результатом будет строка «123456789». Для получения уже собранной строки из объекта stringBuilder, используется метод *toString()*.

Некоторые из самых популярных методов класса **StringBuilder**:

`public StringBuilder append(String s)`	Добавляет boolean, char, int, long, Strings и т.д.
`public StringBuilder reverse()`	Меняет порядок символов на обратный
`public StringBuilder delete(int start, int end)`	Удаляет строку, начиная с начального индекса до конечного индекса.
`public StringBuilder insert(int offset, int i)`	Этот метод вставляет строку в определенную позицию.
`replace(int start, int end, String str)`	Этот метод заменяет символы в подстроке данного StringBuffer строкой str.

Стоит отметить, что класс StringBuilder не является потокобезопасным. Это означает, что если два потока будут модифицировать один и тот же объект **StringBuilder**, это может привести к неожиданному результату. Существует потокобезопасная версия этого класса — **StringBuffer**.

17. Модификаторы доступа

Правила, определяющие возможность или невозможность напрямую получить доступ к полям (методам, классам), называются правилами задания областей доступа. Слова «приватный» и «публичный» в данном случае являются так называемыми «модификаторами доступа». Совместно классы и модификаторы доступа задают область доступа, то есть у каждого участка кода, в зависимости от того, какому классу он принадлежит, будет своя область доступа относительно тех или иных элементов (членов) своего класса и других классов, включая поля, методы, константы и т. д.

Модификаторы Доступом применяется к классам, полям и методам.

Существуют модификаторы:

private (закрытый, внутренний член класса) — обращения к полю или методу допускаются только из методов того класса, в котором этот член определён.

protected (защищённый, внутренний член иерархии классов) — обращения к члену допускаются из методов того класса, в котором этот член определён, а также из любых методов его классов-наследников.

public (открытый член класса) — обращения к члену допускаются из любого кода.

Подробнее это отражено в следующей таблице:

Модификатор	Внутри класса	Внутри пакета	Внутри подклассов	Всем остальным
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

Есть дополнительное ограничение, связанное с модификаторами доступа: код на Java пишется в файлах с расширением .java, и в одном файле может быть только один класс с модификатором **public**. Файл с таким классом должен называться также, как и сам класс.

18. Рекурсия

Рекурсия — вызов метода из самого же себя, непосредственно (простая рекурсия) или через другие функции (сложная или косвенная рекурсия). Например, метод A вызывает метод B, а B — метод A. Количество вложенных вызовов метода называется глубиной рекурсии. Рекурсивная программа позволяет описать повторяющееся вычисление, причём без явных повторений частей программы и использования циклов.

Рассмотрим пример вычисления факториала:

```
int fact(int n) {
    int result;
    if (n == 1)
        return 1;
    result = fact(n - 1) * n;
    return result;
}
```

Дело в том, что факториал числа n это n * (n - 1)!. То есть, чтобы вычислить факториал числа n, необходимо число n умножить на (n - 1)!. А чтобы вычислить (n - 1)! Надо умножить (n-1) на (n - 2)! и так далее, пока не будет достигнута единица. Такой алгоритм и запрограммирован в методе выше.

Более сложным примером использования рекурсии является обход двоичного дерева поиска.

Стоит обратить внимание, что вызов метода рекурсивно не может быть бесконечным. В какой-то момент произойдёт переполнение стека вызовов.

19. Ключевое слово static

До этого рассматривались примеры кода, в которых сначала создаётся объект, а потом вызывается метод. В таком случае метод всегда связан с объектом и может использовать его поля. Но для вызова такого метода необходимо иметь экземпляр класса (объект). Но существуют методы, которые не зависят от состояния объекта, например метод для вычисления факториала, модуля числа, синуса,

косинуса и так далее. Примеров может быть очень много. Для того, чтобы можно было вызывать такие методы (или использовать поля) не создавая экземпляры класса, существует ключевое слово **static**.

Можно пометить поле или метод как **static**, указав этим, что доступ к нему осуществляется непосредственно с помощью класса, без ссылки на объект. Такие поля или методы называются статическими.

```
class MyClass {

    public static int value;

    public static int fact(int n) {
        if (n == 1)
            return 1;
        return fact(n - 1) * n;
    }
}
```

Теперь методы и поля описаны выше можно использовать так:

```
int n = MyClass.value;
int factorial = MyClass.fact(10);
```

Имя класса используется для вызова метода *fact()*. Нет необходимости в экземпляре класса. Однако, такой метод можно вызвать и имея ссылку на объект:

```
MyClass myClass = new MyClass();
int n = myClass.value;
int factorial = myClass.fact(10);
```

В таком случае компилятор на этапе компиляции вместо переменной подставит её тип. Но так делать не рекомендуется. В будущем будет показано, что из-за наследования, выбор компилятором типа может быть неочевидным. Поэтому стоит всегда использовать имя класса при использовании статических полей или методов.

Так как статический метод может быть вызван без создания объекта, внутри таких методов нельзя обращаться к полям, которые не являются статическими. Также из статического метода нельзя вызвать нестатический метод.

20. static-block

По аналогии со статическими методами существует статический блок инициализации. Это похоже на статический конструктор. Он вызывается при первой загрузке класса (класса, а не объекта).

```
class MyClass{

    public static int[] cache;

    static {
        cache = new int[100];
    }
}
```

Внутри статического блока инициализации нельзя обращаться к полям, которые не являются статическими. Также из статического метода нельзя вызвать нестатический метод.

21. Вложенные и внутренние классы

В Java можно объявить класс внутри другого класса или даже метода. Такие классы называются вложенными. Они делятся на статические, внутренние, локальные и анонимные классы.

21.1. Статические вложенные классы

Можно объявить класс внутри другого класса с ключевым словом **static**:

```
public class List{
    public static class Node{
    ...
    }
}
```

Из класса, который является статическим, нельзя обращаться к нестатическим членам внешнего класса. Но при этом, для создания такого статического класса не требуется объект внешнего класса. То есть, чтобы создать экземпляр вложенного статического класса необязательно создание объекта внешнего класса:

```
List.Node node = new List.Node();
```

21.2. Внутренние классы

Немного сложнее обстоит дело с внутренними классами. Внутренний класс от статического отличается отсутствием ключевого слова **static**. Это позволяет из внутреннего класса обращаться ко всем полям и методам внешнего класса. Но экземпляры такого класса нельзя создать без наличия объекта внешнего класса.

```
class List{
    private class Node{
    ...
    }
}
```

Создать экземпляр класса Node можно либо внутри экземпляра класса List, либо использовать ссылку на на экземпляр класса List:

```
List.Node node = new List().new Node();
```

21.3. Локальные классы

Можно объявить локальный класс следующим образом:

```
public void doSome(){

    class Helper{
        int a;

        public Helper(int a) {
            this.a = a;
        }
    }

    Helper helper = new Helper(100);

}
```

То есть, можно объявить класс прямо внутри метода. В методе невозможно использовать ключевое слово static, но в любом случае, доступ к этому классу может быть осуществлён только внутри этого метода.

21.4. Анонимные классы

Для существующего типа можно создать анонимного наследника такого типа (если нет ограничений).
Таким образом создаётся класс без имени, но при этом являющийся наследником начального класса:

```
MyClass myClass = new MyClass(){
    …
};
```

В этом примере создаётся экземпляр класса MyClass, ну тут же открываются фигурные скобки. Это означает, что класс расширяется. При этом, новый класс (наследник MyClass) не имеет имени, то есть он анонимный. Но при этом можно переопределить один или несколько методов в новом классе[4].

4. См. Раздел «22. Наследование» на странице 25.

21.5. Обычный, внутренний, вложенный, локальный или анонимный класс?

Ответ: зависит от задачи.

Если класс будет использоваться повсеместно, то он должен быть обычным публичным классом. Если класс вспомогательный для какого-то класса, то он должен быть внутренним или вложенным. Локальные и анонимные классы нужны только там, где нужно использовать какую-то структуру или поведение единожды.

22. Наследование

Наследование (англ. inheritance) — концепция объектно-ориентированного программирования, согласно которой абстрактный тип данных может наследовать данные и функциональность некоторого существующего типа, способствуя повторному использованию компонентов программного обеспечения. Класс, определённый через наследование от другого класса, называется: производным классом, классом-потомком (англ. derived class) или подклассом (англ. subclass). Класс, от которого новый класс наследуется, называется: предком (англ. parent), базовым классом (англ. base class) или суперклассом (англ. parent class).

Наследование применяется, если нужно добавить или переопределить поведение.

В Java для наследования используется ключевое слово **extends**.

Рассмотрим небольшой пример:

```
class Shape{
    private double area;

    public double getArea() {
        return area;
    }

    public void setArea(double area) {
        this.area = area;
    }
}

class Circle extends Shape{
}
```

В данном примере класс Circle расширяет (наследует) класс Shape. Теперь у объекта типа Circle можно использовать методы объявленные в классе Shape:

```
Circle circle = new Circle();
circle.setArea(3.2);
```

22.1. Модификаторы доступа и наследование

Поля и методы, помеченные как **private**, недоступны для наследников. В примере выше поле area недоступно внутри класса Circle.

protected (защищённый, внутренний член иерархии классов) — обращения к члену допускаются из методов того класса, в котором этот член определён, а также из любых методов его классов-наследников. Например, если в классе Shape объявить поле (или метод) с модификатором **protected**, то такое поле (или метод) будет доступно в классе Circle.

22.2. Переменная суперкласса может ссылаться на объект подкласса

```
Shape circle = new Circle();
```

Так можно делать, потому что Circle это тоже Shape. Если в классе Circle объявить ещё каких-то методов, расширяя класс Shape, и ссылаться на объект типа Circle помощью типа Shape, то методы добавленные в классе Circle вызвать нельзя. Рассмотрим пример:

```
class Shape{
    protected double area;

    public double getArea() {
        return area;
    }

    public void setArea(double area) {
        this.area = area;
    }
}
```

```
class Circle extends Shape{

    protected double radius;

    public double getRadius() {
        return radius;
    }

    public void setRadius(double radius) {
        this.radius = radius;
    }
}

…

Shape circle = new Circle();
int radius = circle.getRadius(); // Так делать нельзя
```

Класс Shape ничего «не знает» о своих наследниках и не может знать что у кого-то из наследников есть метод *getRadius()*.

Возможность переменной суперкласса ссылаться на объект подкласса даёт возможность работать с объектами не заботясь о том, являются ли они объектами суперкласса или подкласса.

22.3. Конструкторы и наследование

Конструктор подкласса обязательно должен вызвать конструктор суперкласса. Конструктор по умолчанию вызывает конструктор по умолчанию суперкласса. Отсюда следует, что если в суперклассе нет конструктора по умолчанию а в подклассе не объявлено никакого конструктора, то это является ошибкой. Конструктор суперкласса можно вызвать с помощью ключевого слова *super()*:

```
class Shape{
    private double area;

    public Shape(double area) {
        this.area = area;
    }
}

class Circle extends Shape{

    public Circle(double area) {
        super(area); // вызываем конструктор суперкласса
    }
}
```

или другой пример:

```
class Shape{
    private double area;

    public Shape(double area) {
        this.area = area;
    }
}

class Circle extends Shape {
    private int radius;

    public Circle(double area, int radius) { // два параметра
        super(area);  // вызываем конструктор суперкласса
        this.radius = radius; // собственная инициализация
    }
}
```

Можно вызвать конструктор суперкласса передав ему конкретное значение. В примере ниже площадь фигуры вычисляется исходя из радиуса:

```
class Shape{
    private double area;

    public Shape(double area) {
        this.area = area;
    }
}

class Circle extends Shape {

    private int radius;

    public Circle(int radius) {
        super(Math.PI * radius * radius);
        this.radius = radius;
    }
}
```

22.4. Порядок вызова конструкторов

Вызов конструктора суперкласса должен быть первым выражением в конструкторе. Из этого следует, что первым выполняется конструктор базового класса, а затем все конструкторы ниже по иерархии наследования.

22.5. Вызов метода с помощью ключевого слова super

По аналогии с **this** работает и ключевое слово **super**. Кроме вызов конструктора с помощью слова **super** можно обращаться к полям и методам суперкласса. Удобно использовать слово **super** при переопределении методов, когда нужно вызвать не переопределённый, а именно родительский метод.

22.6. Переопределение методов

Переопределение метода (англ. method overriding) — одна из возможностей языка программирования, позволяющая подклассу или дочернему классу обеспечивать специфическую реализацию метода, уже реализованного в одном из суперклассов или родительских классов. Реализация метода в подклассе переопределяет (заменяет) его реализацию в суперклассе, описывая метод с тем же названием, что и у метода суперкласса, а также у нового метода подкласса должны быть те же параметры или сигнатура, тип возвращаемого результата, что и у метода родительского класса. Версия метода, которая будет исполняться, определяется объектом, используемым для его вызова. Если вызов метода происходит у объекта родительского класса, то выполняется версия метода родительского класса, если же объект подкласса вызывает метод, то выполняется версия дочернего класса.

Вернёмся к пример с Shape:

```
class Shape{
    private double area;

    public Shape(double area) {
        this.area = area;
    }

    public String toString(){ // Объявили новый метод
        return "I'm shape. My area is " + area;
    }
}

class Circle extends Shape {

    private int radius;

    public Circle(int radius) {
        super(Math.PI * radius * radius);
        this.radius = radius;
    }

    public String toString() { // Объявление с таким же именем — переопределение метода
        return "I'm Circle. My radius is " + radius;
    }
}
```

Создав объекты и вызвав один и тот же метод *toString()*, будет выведен разный результат:

```
Shape shape = new Shape(10);
Shape circle = new Circle(1);
System.out.println(shape.toString());
System.out.println(circle.toString());
```

```
I'm shape. My area is 10.0
I'm Circle. My radius is 1
```

Не стоит путать переопределение и перегрузку. Переопределённые методы имеют одинаковую сигнатуру, в то время как перегруженные разную.

Если нужно переопределить метод для одноразового использования, можно использовать анонимные классы[5].

22.7. Переопределение и статические методы

Переопределить статические методы невозможно. Если в подклассе написать статический метод с такой же сигнатурой, как в суперклассе, то это будет являться не переопределением, а скрытием. Не стоит забывать, что статические методы вызываются с помощью типа, а не объекта и тип всегда указывается явно. Если статический метод вызвать у объекта, компилятор заменит переменную на тип которым она является:

```
MyClass my = new MySubClass();
my.doSome();// Эквивалентно MyClass.doSome();
```

22.8. Скрытие полей

Нет такого понятия как переопределение поля. Если в подклассе объявить поле с таким же именем как в суперклассе, то это является скрытием поля суперкласса.

```
class Shape{
    protected double area;

    public Shape(double area) {
        this.area = area;
    }
}

class Circle extends Shape {

    protected double area;

    public Circle(double area) {
        super(area);
    }

    public String toString() {
        return "My area  = " + area;
    }
}
```

В этом примере в классе Shape объявлено поле area. Класс Circle, который расширяет Shape, тоже имеет поле area. Метод *toString()* класса Circle использует поле, которое скрывает поле суперкласса. Так как поле area класса Circle нигде не инициализируется, то при вызове метода *toString()* поле area равно нулю.

23. Абстрактный класс

Абстрактный класс — базовый класс, который не предполагает создания экземпляров. Абстрактные классы реализуют на практике один из принципов ООП — полиморфизм. Абстрактный класс может содержать (и не содержать) абстрактные методы. Абстрактный метод не реализуется для класса, в котором описан, однако должен быть реализован для его неабстрактных потомков.

Абстрактный класс можно рассматривать в качестве интерфейса к семейству классов, порождённому им.

Например, если вернуться к примеру с фигурой, то понятно, что площадь фигуры вычисляется исходя из других её свойств. Поэтому можно объявить, что у фигуры должна быть площадь, но способ её расчёта должен быть описан в наследниках:

5. См. раздел «21.4. Анонимные классы» на странице 24.

```
abstract class Shape{
    public abstract double area();
}
```

Чтобы объявить класс и метод абстрактным, используется ключевое слово **abstract**. В таком методе отсутствует тело. Если в классе объявлен хотя бы один абстрактным метод, класс должен быть объявлен как абстрактный.

Все подклассы класса Shape должны определить метод *area():*

```
class Circle extends Shape {

    protected double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    public double area() {
        return Math.PI * radius * radius;
    }
}
```

Нельзя создавать экземпляры абстрактных классов.

Абстрактные классы могут содержать обычные методы, поля, конструкторы и так далее.

24. Ключевое слово final

Ключевое слово **final** применяется к классам, методам, полям и переменным.

24.1. final + переменная

Переменные и поля, объявленные с ключевым словом **final**, не могут быть изменены после инициализации. При этом поля класса могут быть объявлены, но инициализированы в конструкторе:

```
class Circle extends Shape {

    private final double radius; // поле

    public Circle(double radius) {
        this.radius = radius;
    }

    public double area() {
        final double area = Math.PI * radius * radius;         // локальная переменная final
        return area;
    }
}
```

После инициализации в конструкторе, переменную с модификатором **final** менять нельзя.

24.2. final + метод

Метод с модификатором **final** нельзя переопределить в подклассах.

24.3. final + класс

Нельзя расширять классы, объявленные как **final**.

24.4. final + abstract

Нельзя использовать модификаторы **final** и **abstract** вместе. Модификатор **abstract** предполагает реализацию, в то время как **final** её запрещает, что делает эту пару лишённой смысла.

25. Константы

Так как в Java нет ключевого слова для объявление констант, то для их объявления используется комбинация из модификаторов **public**, **static** и **final**. Например, такие константы можно встретить в классе **Math**:

```
public static final double PI = 3.14159265358979323846;
```

26. Класс Object

В Java все классы, по умолчанию являются подклассами класса Object. Это означает, что можно написать вот так:

```
Object o1 = "String";
Object o2 = new Circle(34.1);
Object o3 = new StringBuilder();
```

То есть, возможно написание кода, который работает с любыми объектами. Например, метод *println()* объявлен так:

```
public void println(Object x){...}
```

Это позволяет написать:

```
Object o = new Circle(34.1);
System.out.println(o); // любой объект!
```

Метод *println()* получает строковое представление объекта с помощью метода *toString()*, объявленного в Object. То есть, в классе Object есть несколько методов, которые есть и во всех других классах (так как Object суперкласс для всех классов).
Ниже рассмотрены некоторые из них[6].

26.1. toString()

Метод *toString()* возвращает строковое представление объекта. Реализация по умолчанию в классе Object выглядит так:

```
public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

То есть тип объекта вместе с его хеш-кодом (о хеш-кодах речь пойдёт дальше). Рекомендуется всегда переопределять метод *toString()*. Для класса Circle, например, это можно сделать так:

```
public String toString() {
    return "Circle. radius = " + radius;
}
```

Теперь любое преобразование объекта типа Circle в строку даст читаемую строку.

26.2. equals()

Метод *equals()* используется для сравнения двух объектов.
При переопределении этого метода, следует следовать правилам:

- Рефлексивность: для любого объекта x, *x.equals(x)* должен возвращать true.
- Симметрия: для любых объектов x и y, *x.equals(y)* должен возвращать true тогда и только тогда, когда y.*equals(x)* возвращает true.
- Транзитивность: для любых объектов x, y и z, если *x.equals(y)* возвращает true и *y.equals(z)* возвращает true, тогда *x.equals(z)* должен возвращать true.
- Для любых объектов x, y, многочисленный вызов *x.equals(y)* должен давать одинаковый результат, если объекты не менялись.
- Для любого объекта x, *x.equals(null)* должен возвращать false.
- Всегда следует переопределить метод *hashCode()* при переопределении *equals()*.

6. Методы, связанные с многопоточностью, рассмотрены в разделе «38. Многопоточное программирование» на странице 56.

Например:

```
class Circle extends Shape {

    private double radius;
    private String color;


    public boolean equals(Object o) {
        if (this == o) return true; // Рефлексивность

        if (o == null || getClass() != o.getClass()) return false;
        Circle circle = (Circle) o;

        // сравниваем radius
        if (Double.compare(circle.radius, radius) != 0) return false;

        // сравниваем color
        return color != null ? color.equals(circle.color) : circle.color == null;
    }
}
```

Начиная с Java версии 1.7 можно использовать статический метод *equals()* в классе Objects (не Object):

```
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Circle circle = (Circle) o;
    return Double.compare(circle.radius, radius) == 0 && Objects.equals(color, circle.color);
}
```

26.3. hashCode()

Метод *hashCode()* возвращает хеш-код объекта в виде целого числа (int). Он должен быть посчитан на основе полей объекта. Хеш-код объекта активно используется в коллекциях[7].

При переопределении этого метода, следует следовать строгому правилу: если два объекта равны по *equals()* их хеш-коды должны быть одинаковыми.

Именно поэтому, всегда следует переопределить метод *hashCode()* при переопределении *equals()*.

Стоит обратить внимание, что обратное не требуется — объекты с одинаковым хеш-кодом могут быть неравными. То есть, если *hashCode()* возвращает всегда одно и то же число (например 1), то технически это корректно. Однако, коллекции с такими объектами будут работать неэффективно. В идеале, *hashCode()* должен быть равномерно распределён на области значений int.

Один из разработчиков Java, Джошуа Блох, предложил алгоритм вычисления *hashCode()*, который является одним из самых используемых. В том числе его реализацией является метод *hash()* класса Objects (не Object):

```
public int hashCode() {
    return Objects.hash(radius, color);
}
```

Эквивалентный код написанный без Objects.*hash()* выглядит так:

```
int result;
long temp;
temp = Double.doubleToLongBits(radius);
result = (int) (temp ^ (temp >>> 32));
result = 31 * result + (color != null ? color.hashCode() : 0);
return result;
```

26.4. finalize()

О методе *finalize()* уже шла речь на странице 19. Метод *finalize()* тоже является методом класса Object.

7. См. раздел «39. Коллекции» на странице 60.

27. Пакеты

Пакеты позволяют организовать классы в пространства имён. Обычно в пакеты объединяют классы одной и той же категории, либо предоставляющие сходную функциональность.

Классы в Java располагаются в файлах с расширением .java, и файлы могут лежать в различных подкаталогах. Если класс находится в одном из подкаталогов, то первой строкой в классе должно быть объявление пакета:

```
package my.package;

public class Main {…}
```

Класс Main описан в файле Main.java, и находится в каталоге my/package.

Допустимы вложенные пакеты, как в примере выше.

27.1. Пакеты и модификаторы доступа

Классы, определённые без явно заданных модификаторов доступа (**public**, **protected**, **private**), видимы только внутри пакета. С модификатором **protected** — межпакетный доступ только для подклассов. **public** — доступно всем[8].

27.2. Импорт пакетов

Если в каком-то классе необходимо использовать классы из другого пакета, необходимо их импортировать. Для этого существует ключевое слово **import**:

```
package my.package;

import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.Date;
```

Можно импортировать все классы из пакета с помощью «*» :

```
import java.util.*;
```

Есть одно исключение: классы из пакета java.lang импортируются автоматически без явного указания. Примером такого класса является класс String.

28. Интерфейсы

Интерфейс даёт возможность описать поведение класса без его реализации. Он лишь описывает, что должен уметь делать класс, его реализующий. Для описания интерфейсов есть ключевое слово **interface**:

```
interface ThreeDimensionalShape {
    double getVolume();
}
```

Интерфейс ThreeDimensionalShape описывает, что у каждой трёхмерной фигуры, которая реализует интерфейс, есть метод для вычисления объёма. Когда речь идёт об интерфейсах используется слово «реализовать» (**implements**), а не «расширить» или «наследовать».

```
class Sphere implements ThreeDimensionalShape {

    private double radius;

    public Sphere(double radius) {
        this.radius = radius;
    }

    public double getVolume() {
        return 4 / 3 * Math.PI * Math.pow(radius, 3);
    }
}
```

8. См. раздел «17. Модификаторы доступа» на странице 22

В данном примере класс Sphere описывает сферу. Он реализует интерфейс ThreeDimensionalShape и определяет метод *getVolume()* в соответствии со своей реализацией. Аналогично можно написать класс, описывающий цилиндр:

```
class Cylinder implements ThreeDimensionalShape {

    private double radius;
    private double height;

    public Cylinder(double radius, double height) {
        this.radius = radius;
        this.height = height;
    }

    public double getVolume() {
        return Math.PI * radius * radius * height;
    }
}
```

В отличие от переопределения методов, которое необязательно, реализация интерфейса является обязательной. То есть, если объявлено, что класс реализует интерфейс, в нём должны быть реализованы все методы интерфейса.

28.1. Реализация двух и более интерфейсов

В отличие от наследования, можно реализовать несколько интерфейсов. Достаточно после слова implements перечислить все интерфейсы через запятую.

28.2. Использование интерфейса в качестве типа

Ранее уже говорилось, что переменная суперкласса может ссылаться на объект подкласса. Тоже самое и с интерфейсами. Так как класс реализует интерфейс, мы можем ссылаться на экземпляры этого класса с помощью интерфейса:

```
ThreeDimensionalShape shape1 = new Sphere(10);
ThreeDimensionalShape shape2 = new Cylinder(10, 1.5);
System.out.println("Volume = " + shape1.getVolume());
System.out.println("Volume = " + shape2.getVolume());
```

Стоит обратить внимание, что используя переменную shape1 нельзя вызвать никакого метода, кроме *getVolume()*.

28.3. Все методы в интерфейсе — public

Даже если явно не указан модификатор доступа, методы в интерфейсе имеют модификатор **public**.

28.4. Интерфейсы и поля

В интерфейсе не может быть полей в том смысле, в котором они могут быть в классах. Все поля объявленные в интерфейсе по умолчанию имеют модификаторы **public**, **static** и **final**.

28.5. Абстрактные класс или интерфейс

Разница между абстрактным классом и интерфейсом заключается в том, что у интерфейса не может быть никакого состояния, в отличии от класса. Интерфейс лишь описывает поведение.

28.6. Частичные реализации

Если класс, реализующий интерфейс, объявлен как абстрактный, необязательно реализовывать все методы интерфейса. В таком случае эта обязанность перекладывается на потомков этого класса:

```
interface ThreeDimensionalShape {
    double getVolume();
}
```

```
// реализует интерфейс, метод getVolume() не реализован
abstract class ColorThreeDimensionalShape implements ThreeDimensionalShape{

    int color;

    public ColorThreeDimensionalShape(int color) {
        this.color = color;
    }

}


// Расширяет класс ColorThreeDimensionalShape
class Sphere extends ColorThreeDimensionalShape {
    private double radius;

    public Sphere(int color, double radius) {
        super(color);
        this.radius = radius;
    }

    // Так как класс Sphere не абстрактный, то он реализует метод getVolume();
    public double getVolume() {
        return 4 / 3 * Math.PI * Math.pow(radius, 3);
    }
}
```

28.7. Наследование интерфейсом интерфейса

Допустимо наследование одним интерфейсом другого интерфейса. Синтаксис аналогичен синтаксису наследования классов. Если один интерфейс наследует другой, то классы, реализующие этот интерфейс, должны реализовать как методы этого интерфейса, так и его родителя. Например, можно добавить интерфейс для фигуры вращения:

```
interface ThreeDimensionalShape {
    double getVolume();
}

interface SolidOfRevolution extends ThreeDimensionalShape{
    double getRadius();
}
```

28.8. Методы по умолчанию

Начиная с Java версии 1.8, в интерфейсе можно определить методы по умолчанию:

```
interface ThreeDimensionalShape {
    double getVolume();

    default double getWeight(){
        return 0;
    }
}
```

Такой метод можно переопределить, но это необязательно. Кажется, что различий между интерфейсом и абстрактным классом становится ещё меньше. Но главное их отличие остаётся — у интерфейса не может быть состояния.

28.9. Статические методы в интерфейсе

Начиная с Java версии 1.8, в интерфейсе можно определить статические методы. Их объявление и использование не отличается от статических методов в классах[9].

9. См. раздел «19. Ключевое слово static» на странице 22.

29. instanceof и приведение типов

Ранее уже упоминалось, как приводить одни примитивные типы к другим[10]. С объектами можно делать то же самое:

```
ThreeDimensionalShape shape = new Sphere(10);
Sphere sphere = (Sphere) shape;
```

В данном примере переменная shape приводится к типу Sphere. Однако это не всегда безопасно. В примере ниже будет ошибка во время выполнения:

```
ThreeDimensionalShape shape = new Cylinder(10, 5);
Sphere sphere = (Sphere) shape; // Cylinder не Sphere!
```

Чтобы приведение типов сделать безопасным, можно сначала проверить тип на соответствие. Для этого существует ключевое слово **instanceof**:

```
// Проверяется, является ли shape типом Sphere.
if(shape instanceof Sphere){
    Sphere sphere = (Sphere) shape;
}
```

Стоит обратить внимание, это не проверка на строгое равенство типов. То есть, можно проверять, используя супертип:

```
if(shape instanceof ThreeDimensionalShape){…}
```

30. Обработка исключений

Рассмотрим такой код:

```
int array[] = new int[]{1, 2, 3};
System.out.println(array[100]);
```

Так как в массиве всего три элемента, невозможно получить сотый элемент. Программа завершится с ошибкой:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 100
```

В таком случае говорят, что программа «выбросила» или сгенерировала исключение (Exception). Рассмотрим другой пример:

```
public static int getMax(int[] array){
    ...
}

int max = getMax(null);
```

Что должен вернуть метод *getMax()*, если вызвать его с аргументом передать null? Вывернуть какое-либо значение будет некорректно, так как это можно трактовать как найденный максимум. В таком случае будет уместно «бросить» исключение.

Обработка исключений — механизм, предназначенный для описания реакции программы на ошибки времени выполнения и другие возможные проблемы (исключения), которые могут возникнуть при выполнении программы и приводят к невозможности (бессмысленности) дальнейшей отработки программой её базового алгоритма.

Для обработки исключительных ситуаций используются ключевые слова **try**, **catch** и **finally**.

Для обработки исключений код помещается внутрь блока **try**. Если этот код «бросит» исключение, то оно будет поймано и обработано. Небольшой пример:

```
try {
    int  array[] = new int[]{1, 2, 3};
    System.out.println(array[100]);
} catch (Exception e){
    System.out.println("Что-то не так");
}
```

10. См. раздел «5.2. Приведение типов» на странице 8.

В данном случае при попытке обратиться к сотому индексу программа «выбросит» исключение. Выполнение кода внутри **try** будет прервано и управление перейдёт к блоку кода внутри **catch**.

Дальше механизм исключений рассмотрен подробно.

30.1. Типы исключений

Исключения в Java это классы, которые являются потомками (может и не прямыми) родительского класса **Throwable**. У Throwable есть два важных подкласса — **Exception** и **Error**. А также у класса Exception есть специфичный подкласс **RuntimeException**. Иерархия исключений показана на схеме 3:

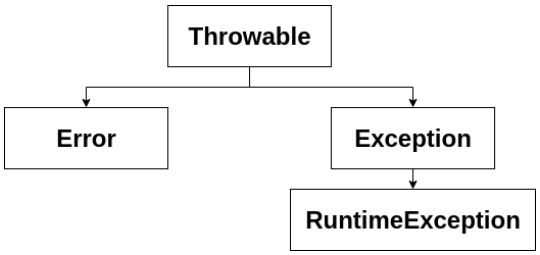

Схема 3: Иерархия исключений в Java.

В зависимости от того, чьим наследником является исключение, оно может быть одним из трёх видов:

- Ошибка внутри виртуальной машины
- Проверяемое исключение
- Непроверяемые исключения

Error

Исключения, которые являются подклассами класса Error, генерируются в случае системных ошибок таких как: VirtualMachineError, OutOfMemoryError, StackOverflowError. Такие исключения, как правило, не обрабатываются.

Exception

При объявлении метода можно указать, что он может «бросить» исключение. При вызове такого метода обязательно наличие обработки соответствующего исключения. Такое исключение называется проверяемым (**checked**).

RuntimeException

Это частный случай класса Exception, который является непроверяемым исключением. Оно может быть выброшено в любом месте кода.

30.2. catch

В блоке **catch** указано, какое именно исключение будет быть обработано:

```
try {
    int  array[] = new int[]{1, 2, 3};
    System.out.println(array[100]);
} catch (ArrayIndexOutOfBoundsException e){
    System.out.println("Что-то не так");
}
```

Можно написать несколько блоков catch для разных вариантов развития событий:

```
try {
    int  array[] = new int[]{1, 2, 3};
    System.out.println(array[100]);
} catch (ArrayIndexOutOfBoundsException e){
    System.out.println("Вышли за границу массива");
} catch (Exception e){
    System.out.println("Что-то не так");
}
```

При объявлении нескольких catch надо следовать правилу: перехват исключений подклассов должен следовать до перехвата суперклассов. Именно поэтому в примере выше сначала перехватывается ArrayIndexOutOfBoundsException, а потом уже Exception. Иначе код будет недостижим.

30.3. Генерирование исключений

Для генерирования исключения используется ключевое слово **throw**.

```
public static int getMax(int[] array) {
    if (array == null) {
        throw new NullPointerException("Массив не должен быть равен null");
    }
    ...
}
```

В данном примере, в случае, если переменная array равна null, бросается исключение. Так как NullPointerException является наследником RuntimeException, у NullPointerException есть несколько перегруженных конструкторов. Один из них принимает строку, описывающую исключение.

NullPointerException является наследником RuntimeException, то есть является непроверяемым исключением. Генерация проверяемых исключений происходит аналогично, но есть нюансы.

30.4. throws и проверяемые исключения

Рассмотрим метод одного из встроенных классов в языке Java:

```
public Date parse(String source) throws ParseException{
    …
    if (pos.index == 0)
        throw new ParseException(...);
    …
}
```

Если что-то пошло не так, в методе *parse()* генерируется исключение ParseException. Но это исключение не является подклассом RuntimeException. Это исключение должно либо быть сразу обработано, либо должно быть отмечено, что метод может «бросить» такое исключение. Для этого используется ключевое слово **throws** при объявлении метода. Таким образом, объявляется, что метод может «бросить» исключение ParseException. Можно перечислить несколько исключений через запятую. При использовании метода с объявлением **throws** обязательно нужно обработать возможное исключение:

```
try {
    SimpleDateFormat simpleDateFormat = new SimpleDateFormat();
    simpleDateFormat.parse(string);
} catch (ParseException e) {
    e.printStackTrace();
}
```

30.5. throws и RuntimeException

Не смотря на то, что непроверяемые исключения не обязательно обрабатывать, их можно указать после слова **throws**. Это не накладывает никаких ограничений и не является ошибкой. Иногда это может улучшить читаемость кода.

30.6. finally

После блока try-catch может быть объявлен блок **finally**. Код в блоке **finally** будет выполнен сразу после блока try-catch и до следующего кода:

```
try {
    int  array[] = new int[]{1, 2, 3};
    value = array[100];
} catch (ArrayIndexOutOfBoundsException e){
    value = -1;
} finally {
    System.out.println(value);
}
```

Независимо от того, будет ли «брошено» исключение или нет, блок кода внутри finally будет выполнен.

Блок finally использовать не обязательно.

30.7. try-finally

Допустимо использование блока try с блоком finally без использования блока catch:

```
try {
    int  array[] = new int[]{1, 2, 3};
    value = array[100];
}  finally {
    System.out.println(value);
}
```

30.8. Создание собственного исключения

Вернёмся к методу, который ещт в массиве наибольший элемент:

```
public static int getMax(int[] array) {
    if (array == null) {
        throw new NullPointerException("Массив не должен быть равен null");
    }
    ...
}
```

В методе уже есть проверка переменной array на значение null, но остаётся открытым вопрос: что возвращать, если массив пуст? Любое число может быть трактовано как найденный результат. Для этого можно создать своё исключение:

```
class ArrayIsEmptyException extends RuntimeException{
    public ArrayIsEmptyException(String message) {
        super(message);
    }

    public ArrayIsEmptyException() {
    }
}
```

Для создания собственного исключения достаточно расширить класс Exception (или RuntimeException). В данном случае расширяется RuntimeException, чтобы исключение было непроверяемым.

Теперь метод можно переписать следующим образом:

```
public static int getMax(int[] array) {
    if (array == null) {
        throw new NullPointerException("Массив не должен быть равен null");
    }

    if (array.length == 0) {
        throw new ArrayIsEmptyException("Массив не должен быть пустым");
    }
    ...
}
```

Стоит отметить, что в большинстве случаев можно использовать стандартные исключения Java. Например, вместо собственного исключения ArrayIsEmptyException можно было бы использовать стандартное IllegalArgumentException.

Некоторые встроенные исключения:

ArithmeticException	Арифметическая ошибка
ArrayIndexOutOfBoundsException	Выход за пределы массива
ClassCastException	Неверное приведение типов
IllegalArgumentException	Неверный аргумент
IllegalStateException	Неверное состояние
IndexOutofBoundsException	Выход за допустимые границы

IOException	Ошибка ввода-вывода
NoSuchElementException	Не найден элемент
NullPointerException	Попытка вызвать метод или обратиться к полю переменной, которая равна null
NumberFormatException	Неверный формат числа
UnsupportedOperationException	Неподдерживаемая операция

30.9. Ещё один вариант обработки нескольких исключений

Начиная с версии 1.7 языка Java вместо объявления нескольких блоков catch, можно перечислить несколько исключений внутри одного блока:

```
try {
    ...
} catch (ArrayIndexOutOfBoundsException | NullPointerException e){
    System.out.println("Вышли за границу массива");
}
```

В такой конструкции допустимо перечислять только те исключения, которые находятся на разных ветках иерархии классов.

30.10. Try-with-resources

В некоторых случаях ресурсы, которые больше не нужны после использование необходимо явно закрыть. Например, потоки ввода-вывода (чтение из файла, сокета) необходимо явно закрывать после использования с помощью метода *close()*. Начиная с версии 1.7 языка Java, если объект реализует интерфейс AutoCloseable, то метод *close()* будет вызван автоматически при использовании конструкции try-with-resources:

```
// у inputStream будет автоматически вызван метод close()
try (InputStream inputStream = new FileInputStream("PFATH_TO_FILE")){
    ...
} catch (Exception e) {
    e.printStackTrace(); // В случае ошибки выводим стектрейс
}
```

31. Классы-обёртки

Раньше уже говорилось, что в Java есть примитивные типы (int, long, short, byte, float, double, boolean, char) и объектные (например, String). Но иногда, нам нужно использовать примитивный тип как объект. Например, для того, чтобы использовать их в коллекциях[11]. Для этого существуют классы-обёртки вокруг примитивов: Byte, Short, Integer, Long, Float, Double, Boolean, Character.

Создавать их можно таким образом:

```
Integer integer = Integer.valueOf(10);
```

```
Integer integer = Integer.parseInt("10");
```

Получить примитив из объекта-обёртки тоже просто:

```
int value = integer.intValue();
```

31.1. Автоупаковка и автораспаковка

Автоупаковка и **автораспаковка** (**autoboxing** и **unboxing**) это автоматическое приведение примитива к объекту-обёртке и наоборот.

Автоупаковка происходит:

- При присвоении значения примитивного типа переменной соответствующего класса-обёртки (и наоборот).

11. См. раздел «39. Коллекции» на странице 60.

- При передаче примитивного типа в параметр метода, ожидающего соответствующий ему класс-обёртку (и наоборот).

Примеры выше можно переписать так:

```
Integer integer = 10;
int value = integer;
```

Никакой «магии» здесь нет, компилятор просто приведёт наш код к такому на этапе компиляции:

```
Integer integer = Integer.valueOf(10);
int value = integer.intValue();
```

Использовать «обёртки» с авто упаковкой и авто распаковкой намного удобнее.

Классы Byte, Double, Float, Integer, Long, Short расширяют более общий класс — Number. Таким образом, методы *byteValue()*, *doubleValue()*, *floatValue()*, *intValue()*, *longValue()* и *shortValue()* можно вызвать у любого из них.

32. Date и Calendar

Кассы Date и Calendar помогают работать с датой и временем.

32.1. UNIX-время

UNIX-время — система описания моментов во времени, принятая в UNIX и других POSIX-совместимых операционных системах. Определяется как количество миллисекунд, прошедших с полуночи (00:00:00 UTC) 1 января 1970 года (четверг); время с этого момента называют «эрой UNIX» (англ. Unix Epoch).

Получить UNIX-время можно с помощью

```
long time = System.currentTimeMillis();
```

UNIX-время обычно используется, если не нужно «человеческое» форматирование времени или даты. Работать с таким форматом удобнее и его удобно хранить — это просто число типа long. Часто в базах данных хранят UNIX-время, а потом уже форматируют его в читаемый вид.

Вот другой пример использования UNIX-времени:

```
long time = System.currentTimeMillis();
...
long result = System.currentTimeMillis() - time;

// выполнение заняло result миллисекунд
System.out.println(result);
```

32.2. Date

Для задач, которые связаны с днями недели, месяцами, годами и так далее существует класс Date и Calendar. Рассмотрим сначала класс Date:

```
Date date = new Date(); // сейчас
```

```
Date date = new Date(System.currentTimeMillis() - 1000*60*60*24); // сутки назад
```

После того, как был создан объект класса Date, он представляет собой некую дату. Прошлую или будущую.

Есть несколько полезных методов в классе Date:

```
long unixTime = date.getTime(); // обратно в UNIX-время
boolean after = date.after(new Date()); // сравнивает две даты
boolean before = date.before(new Date());
```

Если посмотреть документацию класса Date, можно увидеть, что там намного больше методов, но большинство из них помечены как устаревшие. Взамен предлагается использовать класс Calendar.

32.3. Calendar

Calendar — это замена части функциональности класса Date. Работать с ним намного удобнее. Для получения экземпляра класса Calendar нужно воспользоваться статическим методом getInstance():

```
Calendar calendar = Calendar.getInstance();
```

Метод getInstance() возвращает объект Calendar, соответствующий текущему моменту времени. Для замены значения времени на другое, существует несколько методов:

```
// принимает тип Date
calendar.setTime(date);

// UNIX-время
calendar.setTimeInMillis(System.currentTimeMillis() -  1000 * 60 * 60 * 24);
```

Методы *set()* и *get()* позволяют устанавливать и получать различные значения, такие как секунды, минуты, дни и так далее:

```
calendar.set(WHAT, VALUE);
calendar.get(WHAT);
```

Например:

```
calendar.set(Calendar.SECOND, 25);
calendar.set(Calendar.MINUTE, 15);
calendar.set(Calendar.HOUR_OF_DAY, 19);
calendar.set(Calendar.YEAR, 2017);

int yearNow = Calendar.getInstance().get(Calendar.YEAR);
```

Все имена полей, которые можно изменить или получить, заранее объявлены в этом классе как константы. Например, Calendar.SECOND.

Имена некоторых полей, которые можно менять в объекте Calendar:

Поле	**Что меняет/получает**
DAY_OF_MONTH	День месяца
DAY_OF_WEEK	Порядковый день недели
DAY_OF_YEAR	Номер дня в году
HOUR_OF_DAY	Час дня
DATE	День месяца
MILLISECOND	Количество миллисекунд
MINUTE	Количество минут
MONTH	Месяц
SECOND	Секунды
WEEK_OF_MONTH	Номер дня в месяце
WEEK_OF_YEAR	Номер недели в году
YEAR	Год

Также есть константы для месяцев и дней недели, например:

```
calendar.set(Calendar.MONTH, Calendar.JANUARY);

if(calendar.get(Calendar.DAY_OF_WEEK) == Calendar.MONDAY){
    System.out.println("Сегодня понедельник!");
}
```

32.4. UNIX-время — Date — Calendar

Несмотря на наличие различных способов работы со временем, все они легко приводятся друг другу:

Из UNIX-времени:

```
long time = System.currentTimeMillis();

Date date = new Date(time);
Calendar calendar = Calendar.getInstance();
calendar.setTimeInMillis(time);
```

Из объекта Date:

```
long time = date.getTime();
Calendar calendar = Calendar.getInstance();
calendar.setTime(date);
```

Из объекта Calendar:

```
Date date = calendar.getTime();
long time = calendar.getTimeInMillis();
```

33. SimpleDateFormat

Класс SimpleDateFormat создан для форматирования даты в читаемый вид. Для работы с этим классом необходим формат даты и сама дата. Код для форматирования даты выглядит так:

```
Date date = new Date();
SimpleDateFormat simpleDateFormat = new SimpleDateFormat("dd.MM.yyyy");
System.out.println(simpleDateFormat.format(date));
```

Некоторые из полей, которые можно использовать в шаблоне:

Символ	**Что означает**	**Пример**
yy	год (последние 2 цифры)	12
yyyy	год (4-х значное число)	2012
M	номер месяца без лидирующих нулей	2
MM	номер месяца (с лидирующими нулями)	02
MMMM	полное название месяца	Февраль
d	день месяца без лидирующих нулей	7
dd	день месяца с лидирующими нулями	07
E	день недели (сокращение)	Вт
EEEE	день недели (полностью)	вторник
HH	часы в 24-часовом формате с лидирующим нулём	06
hh	время в 12-часовом формате с лидирующим нулём	06
mm	минуты с лидирующим нулём	32
ss	секунды с лидирующим нулём	11
S	миллисекунды	109

Например, можно создавать такие шаблоны:

Шаблон даты и времени	Результат
dd.MM.yyyy	16.11.2017
yyyy.MM.dd G 'at' HH:mm:ss z	2017.11.16 AD at 12:55:26 MSK

SimpleDateFormat можно использовать и наоборот — чтобы получить дату из строки, если известен её формат. Для этого есть метод *parse()*:

```
String dateString = "16.11.2017";
SimpleDateFormat simpleDateFormat = new SimpleDateFormat("dd.MM.yyyy");

Date date = simpleDateFormat.parse(dateString);
```

34. Перечисления (enums)

Перечисления (перечисляемый тип) определяется как набор идентификаторов, с точки зрения языка играющих ту же роль, что и обычные именованные константы, но связанные с этим типом. Описание типа-перечисления выглядит следующим образом:

```
public enum UserRole {
    Admin, Customer, Seller, Support
}
```

Здесь производится объявление типа данных UserRole, значениями которого может быть любая из четырёх перечисленных констант. Переменная типа UserRole может принимать одно из значений Admin, Customer, Seller, Support. Допускается сравнение значений на равенство или неравенство, а также использование их в операторе switch в качестве значений, идентифицирующих варианты.

В Java перечисление представляет собой полноценный класс, в который можно добавлять произвольное количество полей и методов. Их можно использовать, например, в коллекциях.

```
public class User{
    private UserRole userRole; // Перечисление как поле

    ....

    public UserRole getUserRole() {
        return userRole;
    }

    public void setUserRole(UserRole userRole) {
        this.userRole = userRole;
    }
    ......
}
```

```
if(user.getUserRole() == UserRole.Seller){
    ...
}
```

При использовании перечислений в switch имя перечисления можно опустить:

```
User user = new User();
switch (user.getUserRole()){
    case Admin: ... ); break;
    case Customer: ... ; break;
    case Support: ... ; break;
    case Seller: ... ; break;
}
```

Использование перечислений позволяет сделать исходные коды программ более читаемыми, так как позволяют заменить «магические числа», кодирующие определённые значения, на читаемые имена.

34.1. values() и valueOf()

У любого перечисления всегда есть два предопределённых метода: values() и valueOf(). Метод values() возвращает массив, содержащий список констант перечисления. А метод valueOf() возвращает константу перечисления, значение которой соответствует строке, переданной в качестве аргумента.

```
UserRole[] all = UserRole.values(); // Массив всех значений
for(UserRole role : all){
    // Любой массив можно использовать в foreach
    System.out.println(role);
}
```

```
Admin
Customer
Seller
Support
```

```
UserRole role = UserRole.valueOf("Admin");
System.out.println(role);
```

```
Admin
```

34.2. Поля, методы и конструкторы в перечислениях

Перечисления являются полноценными классами, поэтому в них можно добавлять поля и методы:

```
public enum UserRole {
    Admin(0), Customer(1), Seller(2), Support(3);

    // В базе будем хранить в виде целого числа
    private int dataBaseValue;


    UserRole(int dataBaseValue) {
        this.dataBaseValue = dataBaseValue;
    }

    UserRole getUserRoleByDatabaseNumber(int n){
        switch (n){
            case 0: return Admin;
            case 1: return Customer;
            case 2: return Seller;
            case 3: return Support;
            default:throw new IllegalArgumentException();
        }
    }
}
```

В данном примере перечисление включает в себя логику преобразования перечисления в идентификатор в базе данных, в которой эти значения хранятся в виде целых чисел. Для этого не только объявлено поле dataBaseValue, но и конструктор, который принимает один аргумент — dataBaseValue. Если объявлен конструктор с параметрами, то список перечислений выглядит иначе:

```
Admin(0), Customer(1), Seller(2), Support(3);
```

Конструктор вызывается один раз для каждого варианта перечисления. То есть каждый вариант перечисления — это отдельный объект, с собственным состоянием.

34.3. Ограничение

Несмотря на то, что перечисление является полноценной классом, существует одно ограничение: перечисление не может наследоваться от другого класса и не может быть суперклассом.

34.4. Суперкласс Enum

Перечисление не может наследоваться от другого класса и не может быть суперклассом, однако по умолчанию все перечисления являются наследниками класса Enum. Таким образом, у любого перечисления кроме методов *values()* и *valueOf()* есть ещё несколько методов:

`compareTo(E o)`	`Метод для сравнения перечислений. По умолчанию порядок определяется порядком объявления значений.`
`name()`	`Возвращает имя значения.`
`ordinal()`	`Возвращает порядковый номер значения в списке объявления. Обычно используется в таких структурах данных, как EnumSet и EnumMap, в других случаях почти не используется.`

35. Аннотации

Аннотации — специальная форма синтаксических метаданных, которая может быть добавлена в исходный код. Аннотации используются для анализа кода, компиляции или выполнения. Аннотируемы пакеты, классы, методы, переменные и параметры.

Например, любой переопределённый (или реализованный) метод можно аннотировать аннотацией **@everride**:

```
@Override
public String toString() {
    return super.toString();
}
```

Аннотация @Override не обязательна, но полезна: если метод аннотирован @Override, то проверяется, переопределён ли метод. Вызывает ошибку компиляции, если метод не найден в родительском классе или интерфейсе.

Аннотации выполняют следующие функции:

- дают необходимую информацию для компилятора;
- дают информацию различным инструментам для генерации другого кода, конфигураций и т. д.;
- могут использоваться во время выполнения для получения данных через отражение (reflection)[12];

35.1. Создание и использование аннотаций

Для создания аннотации используется ключевое слово interface вместе с «**@**»:

```
public @interface UnstableCode {}
```

Теперь можно аннотировать класс, метод или поле:

```
@UnstableCode
public class Main {
    ...
}
```

Аннотация может иметь параметры, которые объявляются как методы:

```
public @interface Version {
    String version();
}
```

Аннотация с параметрами:

```
@Version(version = "1.0")
private void doSome(){
    …
}
```

Можно использовать несколько аннотация для одного элемента:

12. См. пример «Получение данных из OpenStreetMap» на странице 82.

```
@Version(version = "1.1")
@UnstableCode
public class Main {
    ...
}
```

Существует несколько ограничений на создание аннотаций:

- Аннотации не могут участвовать в наследовании
- Все методы в аннотациях должны быть без параметров
- Только примитивные типы, String, Class, перечисления или массивы ранее перечисленных типов могут быть параметрами аннотации
- Аннотации не могут быть обобщёнными[13]

35.2. Значения по умолчанию

В параметрах аннотаций можно использовать значения по умолчанию:

```
public @interface Version {
    String version() default "1.0";
}
```

35.3. @Retention

Аннотация @Retention позволяет указать, в какой момент жизни программы будет доступна аннотация. @Retention применяется к другим аннотациям:

```
@Retention(RetentionPolicy.SOURCE)
public @interface Version {
    String version();
}
```

@Retention имеет один параметр — RetentionPolicy, который может принимать одно из трёх значений:

SOURCE — аннотация доступна только в исходном коде.
CLASS — аннотация хранится в .class файле, и доступна только на этапе компиляции.
RUNTIME — аннотация доступна во время выполнения программы.

35.4. @Target

Аннотация @Target применяется к аннотациям и ограничивает типы элементов, к которым можно применить аннотацию:

```
@Target(ElementType.FIELD)
public @interface Version {
    String version() default "1.0";
}
```

В данном примере, аннотацию Version можно применить только к полям.

Возможные значения ElementType:

TYPE	Классы, интерфейсы, перечисления
FIELD	Поля
METHOD	Методы
PARAMETER	Параметры
CONSTRUCTOR	Конструкторы
LOCAL_VARIABLE	Локальные переменные

13. См. раздел «37. Обобщения (Generics)» на странице 50.

ANNOTATION_TYPE	Другие аннотации
PACKAGE	Пакеты

При использовании @Target можно перечислить несколько вариантов:

```
@Target({ElementType.FIELD, ElementType.TYPE})
public @interface Version {
    String version() default "1.0";
}
```

36. Ввод-вывод

Стандартная библиотека Java содержит набор классов, который позволяет работать с потоками ввода/вывода, файловой системой и так далее. В этом разделе дан краткий обзор некоторых таких классов.

36.1. Класс File

Класс File позволяет оперировать с файлам и каталогами. Для чтения и записи используются потоки ввода/вывода, а класс File используется для чтения различных данных о файле.

Некоторые из методов класса File:

getAbsolutePath()	Возвращает полный путь к файлу
canRead() / canWrite()	Можно ли читать / писать
exists()	Существует ли файл
getName()	Имя файла
getPath()	Путь файла
lastModified()	Дата последнего изменения файла
list()	Возвращает список имён файлов в каталоге
listFiles	Возвращает список файлов в каталоге
isFile() / isDirectory()	Является ли объект файлом / каталогом
isAbsolute()	Имеет ли файл абсолютный путь
mkdir()	Создать каталог
renameTo(newFile)	Переименовать файл
delete()	Удалить файл

Также класс File содержит публичное поле File.separator, которое является разделителем в пути (в Windows это \, в Linux /).

Небольшой пример:

```
File file = new File("PATH_TO_FILE");

System.out.println(file.canExecute()); // true
System.out.println(file.canRead()); // true
System.out.println(file.canWrite()); // true

// /home/user/Downloads
System.out.println(file.getAbsolutePath());
System.out.println(file.getFreeSpace()); // 106993594368
System.out.println(file.getName()); // Downloads
System.out.println(file.getParent()); // /home/user
System.out.println(file.getParentFile().getName()); // user
```

```
System.out.println(file.getTotalSpace()); // 872322404352
System.out.println(file.isDirectory()); // true
System.out.println(file.isFile()); // false
System.out.println(file.length()); // 53248
System.out.println(file.exists()); // true
System.out.println(File.separator); // /

// File list
System.out.println(Arrays.toString(file.list()));
```

36.2. Потоки ввода/вывода

Потоки ввода/вывода в Java — это абстракция, являющаяся либо поставщиком, либо потребителем потока данных.

В Java потоки делятся два типа — байтовые потоки и символьные. Байтовые потоки, работают с байтами, а символьные, соответственно, с символами.

Потоки ввода/вывода образуют сложную иерархию классов, во главе которой находятся **InputStream** и **OutputStream**.

InputStream является абстрактным, и является потоком ввода. Большинство его методов могут генерировать исключение **IOException**.

Некоторые метода из класса InputStream:

`int available()`	Количество байт, доступных для чтения
`void close()`	Закрывает поток
`int read()`	Возвращение целочисленное значение следующего байта. По достижении конца файла возвращает -1.
`int read(byte b[])`	Читаем массив байт, записывая его в массив

Класс OutputStream описывает поток вывода.

Некоторые метода из класса OutputStream:

`void close()`	Закрывает поток
`int flush()`	Очищает все буферы, делая конечным состояние вывода.
`void write(byte b[])`	Записывает массив в поток

36.3. FileInputStream

Класс FileInputStream реализует InputStream и предназначен для чтения байт из файла. Пример использования FileInputStream:

```
InputStream inputStream = null;

try {
    // Создаём inputStream в try
    inputStream = new FileInputStream("PATH_TO_FILE");

    int nRead;
    byte[] data = new byte[128];  // Буфер для чтения

    // nRead - количество прочитанного
    while ((nRead = inputStream.read(data, 0, data.length)) != -1) {
         System.out.println(Arrays.toString(data)); // просто выводим на консоль
    }
} catch (Exception e) {
    e.printStackTrace(); // В случае ошибки выводим стектрейс
} finally { // в блоке finally закрываем поток
    if (inputStream != null)
        try {
            inputStream.close();   // close() тоже может "бросить" исключение
        } catch (IOException e1) {
            e1.printStackTrace();
        }
}
```

Этот код может показаться немного сложным из-за необходимости в нескольких блоках try-catch. Код можно упростить, если использовать конструкцию try-with-resources[14]:

```
// у inputStream будет автоматически вызван метод close()
try (InputStream inputStream = new
        FileInputStream("PATH_TO_FILE")) {

    int nRead;
    byte[] data = new byte[128];  // Буфер для чтения

    // nRead - количество прочитанного
    while ((nRead = inputStream.read(data, 0, data.length))
            != = -1) {
        System.out.println(Arrays.toString(data));
    }
} catch (Exception e) {
    e.printStackTrace(); // В случае ошибки выводим стектрейс
}
```

36.4. FileOutputStream

Класс FileOutputStream реализует OutputStream и предназначен для байтового вывода в файл:

```
public static void writeBytes(byte[] bytes)  {
    try (OutputStream outputStream = new FileOutputStream("PATH_TO_FILE")){
        outputStream.write(bytes);
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

36.5. Буферизованные потоки ввода-вывода

Буферизованные потоки ввода-вывода — вспомогательные классы, которые оптимизируют работу с потоками ввода-вывода с помощью буфера в памяти. Такие потоки «оборачивают» InputStream и OutputStream, при этом расширяя их:

```
public static void readBytes() throws IOException {
    InputStream inputStream = new BufferedInputStream(new FileInputStream(new
                                                                File("PATH_TO_FILE")));

    int data = inputStream.read();
    char content;

    while (data != -1) {
        content = (char) data;
        System.out.print(content);
        data = inputStream.read();
    }
}
```

```
public static void writeBytes(byte[] bytes)  {
    try (OutputStream outputStream = new BufferedOutputStream(new
                                                        FileOutputStream("PATH_TO_FILE"))){
        outputStream.write(bytes);
        outputStream.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

В этих двух примерах FileOutputStream и FileInputStream «оборачиваются» в буферизованные потоки ввода-вывода.

14. См. раздел «30.10. Try-with-resources» на странице 39.

36.6. Reader и Writer

Абстрактные классы **Reader** и **Writer** осуществляют потоковый ввод и вывод символов. Такие классы позволяют читать символы или текст из файла или из других источников. В частности, такие классы как BufferedReader, BufferedWriter, FileReader, FileWriter. По аналогии с байтовыми потоками BufferedReader и BufferedWriter могут оборачивать другие Reader и Writer.

Классы InputStreamReader и OutputStreamWriter позволяют организовать символьные потоки на основе любых потоков ввода/вывода (если известно, что их данные — символы)[15].

Например, чтение файла построчно можно написать так:

```
try (BufferedReader br = new BufferedReader(new FileReader(file))) {
    String line;
    while ((line = br.readLine()) != null) {
        // process the line.
    }
}
```

И запись:

```
try (BufferedWriter br = new BufferedWriter(new FileWriter(file))) {
    for(String s : strings){
        br.write(s);
    }
}
```

36.7. Дополнительная информация

В Java существует ещё много специфических потоков ввода/вывода, перечисление которых займёт здесь слишком много места. Есть много специфических задач, для которых есть специфичные классы и решения.

Также есть множество сторонних библиотек, например Apache Commons IO. Commons IO содержит большое количество вспомогательных инструментов для работы с потоками.

37. Обобщения (Generics)

Зная, что с помощью Object можно ссылаться на любые объекты, можно писать классы, работающие с любыми объектами, например динамические массивы, хранящие Object. Или писать методы, которые на входе принимают массив любых объектов. Но есть один недостаток такого подхода — это небезопасно. Небезопасно в том смысле, что приведение таких объектов к другим типам может закончится ошибкой. Для решения этой проблемы и существуют параметризованные типы (обобщения, generic).

37.1. Простой пример

Предположим, что есть некий объект, в который можно «положить» или «достать» из него другие объекты:

```
public class Box {
    private Object object;

    public Object get() {
        return object;
    }

    public void set(Object object) {
        this.object = object;
    }
}
```

Легко получить ошибку:

```
Box box = new Box();
box.set("value");

...

// Где-то в другом месте кода:
// Ошибка во время выполнения! Там String, а не Integer!
Integer integer = (Integer) box.get();
```

15. См. пример «Получение данных из OpenStreetMap» на странице 82.

Все меняется, когда класс параметризован:

```
public class Box<T> {
    private T object;

    public T get() {
        return object;
    }

    public void set(T object) {
        this.object = object;
    }
}
```

В объявлении класса указано, что класс параметризован (<T>). T — это имя параметра в классе, оно может быть любым (на него действуют правила именования как у переменных), но принято использовать заглавные большие буквы. Теперь параметр T можно использовать как тип в методах, полях, и так далее.

При создании экземпляра такого класса он параметризуется конкретным типом:

```
Box<String> box = new Box<String>();
box.set("value");
box.set(new File("name")); // ОШИБКА, Можно только String

...

// Где-то в другом месте кода:
// Ошибка во время компиляции!
Integer integer = (Integer) box.get();
```

В данном примере, когда создаётся объект (new Box<String>()) параметризованный типом String, компилятор автоматически проверит типы аргументов, возвращаемых значений и переменных.

37.2. Только ссылочные типы

Нельзя параметризовать объект примитивным типом. Например, нельзя при создании Box параметризовать его типом int. Но если это все-таки требуется, то можно использовать классы-обёртки[16], то есть использовать Integer.

37.3. Обобщённые типы и приведение типов

Классы, обобщённые разными типами, неприводимы друг к другу:

```
Box<String> boxString = new Box<String>();
Box<Integer> boxInteger = new Box<Integer>();

boxString = boxInteger; // Ошибка компиляции
```

37.4. Параметризация по умолчанию

Если по каким-то причинам параметризация не требуется, то можно опустить параметр. Тогда, объект будет параметризован типом Object. Такие два объявления эквивалентны:

```
Box box1 = new Box();
Box<Object> box2 = new Box<Object>();
```

37.5. Автоопределение параметра

При создании объекта параметризованного типа параметр можно не указывать, если компилятор может определить тип автоматически:

```
Box<String> box = new Box<>();
```

16. См. раздел «31. Классы-обёртки» на странице 39.

Но не стоит путать new Box() и new Box<>(). В первом случае объект не параметризован, во втором — тип определён автоматически.

37.6. Два и более параметра

При объявлении классов можно использовать два и более параметра:

```
class Box<T, V> {
    private T first;
    private V second;

    public T getFirst() {
        return first;
    }

    public V getSecond() {
        return second;
    }
}
```

```
Box<String, Integer> box = new Box<String, Integer>();
box.setFirst("String");
box.setSecond(100); // autoboxing to Integer
```

37.7. Ограниченные типы

Иногда требуется ограничить возможные параметры класса определёнными типами. Для этого используется ключевое слово **extends**.

В примере ниже, тип должен являться наследником класса **Number (**наследниками которого являются, в частности, классы Byte, Double, Float, Integer, Long, Short):

```
class Box<T extends Number> {
    private T number;

    public T getNumber() {
        return number;
    }

    public void setNumber(T number) {
        this.number = number;
    }

    @Override
    public String toString() {
        // Так можно делать, ведь number точно какой-то из подклассов Number
        // А метод doubleValue() объявлен в классе Number
        return "value = " + number.doubleValue();
    }
}
```

Экземпляры такой реализации класса Box могут быть параметризованы или классом Number, или его наследниками. Такое ограничение позволяет вызывать методы класса Number внутри параметризованного класса Box (так как известно, что number точно является наследником Number).

Можно усложнять такие условия, требуя расширения какого-то класса и реализации нескольких интерфейсов:

```
class Box<T extends Number & Comparable & Cloneable & Serializable> {
    private T number;

    public T getNumber() {
        return number;
    }

    public void setNumber(T number) {
        this.number = number;
    }

    @Override
    public String toString() {
        return "value = " + number.doubleValue();
    }
}
```

Если в таком списке ограничений есть класс (в данном случае Number), то он должен быть объявлен первым и быть единственным. Интерфейсов может быть несколько (ведь можно реализовывать несколько интерфейсов). Параметр такого класса должен реализовывать все из них.

37.8. Метасимвольные аргументы

Если параметризованный класс выступает в качестве типа аргумента какого-то метода, то его параметры тоже можно ограничивать. Например, рассмотрим такой метод:

```
public void doSome(Box<? extends Double> box){
    ...
}
```

В такой метод можно передать только те экземпляры Box, которые параметризованным типом Double (или наследниками). Если не объявить никаких условий, то ограничений, соответственно, не будет (за исключением ограничений в самом Box). В таких аргументах можно использовать и слово **super**:

```
public void doSome(Box<? super Double> box){…}
```

То есть передать в метод *doSome()* можно только те экземпляры Box, которые параметризованным типом Double или его родителями.

Более понятным примером использования такой конструкции служит метод *sort()* класса Arrays:

```
public static <T> void sort(T[] a, Comparator<? super T> c) {
    if (LegacyMergeSort.userRequested)
        legacyMergeSort(a, c);
    else
        TimSort.sort(a, c);
}
```

Такой метод принимает массив параметризованного типа T и требует, чтобы Comparator был параметризован таким же типом, или типом суперкласса (может не прямого). Это связано с тем, что внутри метода *sort()* будет вызываться метод *compare()* у объекта реализующего Comparator, и так как этот метод можно вызвать, передав любой объект выше чем T по иерархии наследования, то это будет работать. В качестве аргумента метода *sort()* можно использовать классы, реализующие интерфейс Comparator для суперклассов T:

```
// Comparator для всех подклассов Number
Comparator<Number> numberComparator  = new Comparator<Number>() {
    public int compare(Number o1, Number o2) {
        return Double.compare(o1.doubleValue(), o2.doubleValue());
    }
};

// Массив целых чисел. Integer - подкласс Number
Integer[] array = new Integer[]{...}

Arrays.sort(array, numberComparator); // Работает
```

37.9. Параметризованные методы

Если взглянуть на метод *sort()* класса Arrays, то можно заметить, что перед типом возвращаемого значения тоже указан параметр. Это и есть пример обобщённого (параметризованного) метода. Тип метода никак не связан с типом класса, в котором он объявлен. Можно параметризовать как обычные методы, так и статические, можно делать это в параметризованном классе, а можно в обычном. Например:

```
public static <T> T getRandoFromArray(T[] arr){
    return ...
}
```

В примере выше метод параметризован, возвращает объект типа параметра, и принимает массив такого же типа. Вызов метода:

```
Integer[] integers = new Integer[]{...};
String[] strings = new String[]{...};

Integer randomInteger = getRandoFromArray(integers);
String randomString = getRandoFromArray(strings);
```

В данном случае необязательно указывать тип явно, но можно и указать:

```
Integer randomInteger = Main.<Integer>getRandoFromArray(integers);
```

Как и у классов, тип можно ограничить:

```
public static <T extends Number> T getRandoFromArray(T[] arr){
    return ...
}
```

37.10. Параметризованные интерфейсы

По аналогии с классами можно объявлять параметризованные интерфейсы. Примером может быть интерфейс Comparable[17] из пакета java.util:

```
public interface Comparable<T> {
    /**
     A lot of comments...
     */
    public int compareTo(T o);
}
```

37.11. Иерархия параметризованных классов

Обобщённые классы могут быть подклассами других классов (параметризованных или нет), и быть суперклассами других классов (параметризованных или нет). При расширении суперкласса аргументы типа, требующиеся суперклассу, должны быть переданы вверх по иерархии.

Подкласс может быть параметризован как суперкласс (или интерфейс):

```
// Класс Box параметризован, и реализует интерфейс Comparable
// В Comparable передаётся тот же Box<T>, чтобы можно было
// сравнивать Любые два объекта Box<T>
class Box<T extends Number> implements Comparable<Box<T>>{
    private T number;

    public T getNumber() {
        return number;
    }

    public void setNumber(T number) {
        this.number = number;
    }
```

17. См. пример «Сортировка строк по длине» на странице 75.

```
    public int compareTo(Box<T> o) {
        return Double.compare(number.doubleValue(),
                o.number.doubleValue());
    }
}
```

Подкласс может быть не параметризован в отличие от суперкласса (или интерфейса), но тогда параметр суперкласса следует указать:

```
// Годится только для String
class StringLegthComparator implements Comparator<String>{
    public int compare(String o1, String o2) {
        return Integer.compare(o1.length(),o2.length());
    }
}
```

37.12. Приведение типов

Параметризованные типа приводимы, только если их типы одинаковы:

```
class A<T>{} // Параметризованный класс
class B<T> extends A<T>{}
…
A<String> a = new B<String>();
B<String> b = (B<String>) a;
```

37.13. Возникновение неоднозначности

При использовании обобщённых типов могут появиться неоднозначности, которые компилятор не может разрешить. В примере ниже, если объект будет параметризован так, что оба параметры будут одинаковы (<String, String>), то оба метода *set()* будут иметь одинаковую сигнатуру. Поэтому в принципе такой код скомпилирован не будет:

```
class Box<T, V>{
    private T t;
    private V v;

    public void set(T t) {
        this.t = t;
    }

    public void set(V v) {
        this.v = v;
    }
}
```

37.14. Некоторые ограничения

1. Создавать экземпляры по параметру типа нельзя:

```
class Box<T>{
    public Box() {
        t = new T(); // Ошибка!!!
    }
}
```

2. Статические методы не могут использовать параметр класса.
3. Создавать массивы по параметру типа нельзя):

```
class Box<T>{
    private T[] t = new T[10]; // Нельзя
}
```

4. Нельзя создавать параметризованные исключения

38. Многопоточное программирование

Многопоточность — свойство, состоящее в том, что процесс, порождённый в операционной системе, может состоять из нескольких потоков, выполняющихся «параллельно», то есть без предписанного порядка во времени. При выполнении некоторых задач такое разделение может достичь более эффективного использования ресурсов вычислительной машины.

Такие потоки называют также потоками выполнения (от англ. thread of execution); иногда называют «нитями» (буквальный перевод англ. thread) или неформально «тредами».

Не стоит путать потоки выполнения (threads) и потоки ввода/вывода (streams).

В рамках курса рассматриваются основы многопоточного программирования — создание потоков и основы синхронизации. Для углублённого изучения рекомендуется книга Vijay Garg — Concurrent and Distributed Computing in Java.

38.1. Создание потока

Создать поток можно двумя путями:

1. Реализовав интерфейс **Runnable**
2. Расширив класс **Thread**

38.2. Реализация интерфейса Runnable

В интерфейсе **Runnable** определён один метод — r*un()*. Этот метод и будет выполнен в отдельном потоке. Для запуска потока используется класс **Thread**. В конструктор класса Thread можно передать экземпляр Runnable:

```
Runnable runnable = new Runnable() { // Анонимный класс
    @Override
    public void run() {
        System.out.println("Other thread");
    }
};

Thread thread = new Thread(runnable);
thread.start(); // Запуск потока
```

Стоит обратить внимание, что поток стартует только после явного вызова метода *start()* у объекта thread.

38.3. Расширение класса Thread

Второй вариант запуска потока — расширение класса Thread. Расширить нужно тоже метод *run():*

```
Thread thread = new Thread(){
    @Override
    public void run() {
        System.out.println("Other thread");
    }
};

thread.start();
```

И снова вызывается метод *start().*

38.4. Создание нескольких потоков

Создать несколько потоков можно просто создав несколько экземпляров **Thread**:

```
class MyThread extends Thread{
    @Override
    public void run() {
      ...
    }
}
...
for (int i = 0; i < 10; i++) {
    new MyThread().start();
}
```

Важно отметить, что один поток может быть исполнен только один раз. То есть, нельзя вызвать метод *start()* у одного и того же объекта дважды. Именно поэтому в примере выше на каждой итерации цикла создаётся новый экземпляр класса Thread.

38.5. Главный поток

При старте программы запускается главный поток — в нем и вызывается метод *main()*. Несмотря на то, что этот поток запускается виртуальной машиной, можно получить ссылку на объект Thread:

```
Thread mainThread = Thread.currentThread();
```

Метод Thread.currentThread() возвращает ссылку на поток, из которого вызван этот метод.

38.6. sleep()

У класса Thread есть статический метод *sleep():*

```
Thread.sleep(3000);
```

Поток, вызвавший этот метод «уснёт» на время, указанное в параметре (в миллисекундах).

38.7. Метод join()

С помощью метода *join()* один поток может ждать выполнения другого. Поток, вызвавший этот метод, будет ждать завершения другого:

```
class MyThread extends Thread {

    Thread thread; // Другой поток, завершения которого нужно ждать
    public MyThread(Thread thread) {
        this.thread = thread;
    }

    public void run() {
        try {
            this.thread.join(); // Ждем, пока другой поток завершится
            System.out.println("I was waiting main thread");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```
for (int i = 0; i < 10; i++) {
    new MyThread(Thread.currentThread()).start();
}

Thread.sleep(3000);
```

В данном примере все десять потоков MyThread будут ждать, пока завершится главный поток (3 секунды), после чего будет выведена строка "I was waiting main thread" (каждый поток по одному разу).

38.8. Синхронизация

Если два и более потока используют общий ресурс (объекты, файлы и так далее), необходима гарантия, чтобы не произошло никаких ошибок, связанных с конкурентным доступом. Для этого существуют механизмы синхронизации потоков[18].

В основе этих механизмов лежит понятие монитора.

Монитор — механизм взаимодействия и синхронизации процессов, обеспечивающий доступ к неразделяемым ресурсам. Одновременно монитором может владеть только один поток. Если в момент попытки захвата монитор занят, поток, пытающийся его захватить, будет ждать до тех пор, пока он не освободится.

Синхронизировать код в Java можно с помощью ключевого слова **synchronized**.

18. Классическим примером синхронизации является система производителя-потребителя (стр. 79).

38.9. synchronized-блоки

Код, помещённый внутрь блока **synchronized**, является синхронизированным. В примере ниже, в методе *doSomeWork()* объявлен блок synchronized. Когда какой-то поток входит в этот блок, монитор (монитором может быть любой объект) захватывается. Все другие потоки будут ждать, пока монитор освободится (то есть первый поток выйдет из блока synchronized). В примере кода, описанном ниже, в двух методах происходит синхронизация по одному монитору. Пока один поток «находится внутри» одного из таких блоков, другие потоки не могут войти ни в один из них.

```
class Worker {
    private Object lock = new Object();

    public void doSomeWork() {
        synchronized (lock) {
            // todo
        }
    }

    public void doSomeOtherWork() {
        synchronized (lock) {
            // todo
        }
    }
}
```

Стоит отметить, что блок synchronized не связан с методами или с полями класса напрямую. В качестве монитора можно использовать любой объект, а внутри метода может быть несколько synchronized-блоков:

```
class Worker {
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    public void doSomeWork() {
        synchronized (lock1) {
            synchronized (lock2) {
            }
        }
    }
}
```

38.10. synchronized-методы

Можно использовать слово synchronized при объявлении методов. Для нестатических методов две синтаксических конструкции, описанных ниже эквивалентны:

```
public void doSomeWork() {
    synchronized (this) {
        // todo
    }
}
```

```
public synchronized void doSomeWork() {
    // todo
}
```

Для статических:

```
public static void doSomeWork() {
    synchronized (Worker.class) {
        // todo
    }
}
```

```
public static synchronized void doSomeWork() {
    // todo
}
```

38.11. wait(), notify(), notifyAll()

Иногда требуется не просто синхронизировать блоки кода во избежание ошибок, но и ждать, пока один поток завершит какую-то работу, от которой зависят дальнейшие действия. Для решения таких задач существует механизм взаимодействия потоков, реализованный в методах *wait(), notify(), notifyAll()*. Классе Object содержит три метода для управления потоками:

wait()	Поток, в котором произошёл вызов такого метода будет ждать, пока какой-то другой поток не вызовет метод notify()
notify()	Возобновляет работу потока, который вызвал wait()
notifyAll()	Возобновляет работу всех потоков, которые вызвали wait()

Речь идёт о вызове этих методов у **одного объекта**.
Все вышеперечисленные три метода, должны вызываться только внутри блока **synchronized**:

```
public  void doSomeWork() {
    synchronized (lock){
        try {
            lock.wait();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

38.12. Получение состояния

Можно получить состояние любого потока с помощью метода *getState():*

```
Thread.State state = thread.getState();
```

Thread.State является перечислением. В нем объявлены следующие значения:

NEW	Поток создан, но ещё не запущен
RUNNABLE	Поток выполняется
BLOCKED	Поток приостановлен
WAITING	Поток ждёт вызова notify() или завершения другого потока
TIMED_WAITING	Ожидает определённое время (например, после вызова sleep()).
TERMINATED	Поток завершён

39. Коллекции

Коллекции в Java — это набор классов и интерфейсов, описывающие и реализующие различные структуры данных, такие как связные списки, динамические массивы, множества, очереди и так далее, плюс вспомогательные классы и интерфейсы.

В данном разделе приведены общие сведения об использовании таких классов. Для более углублённого изучения возможностей коллекций и их устройства рекомендуется обратиться к специализированной литературе.

39.1. Интерфейсы коллекций

Иерархия основных интерфейсов коллекция представлена на схеме 3:

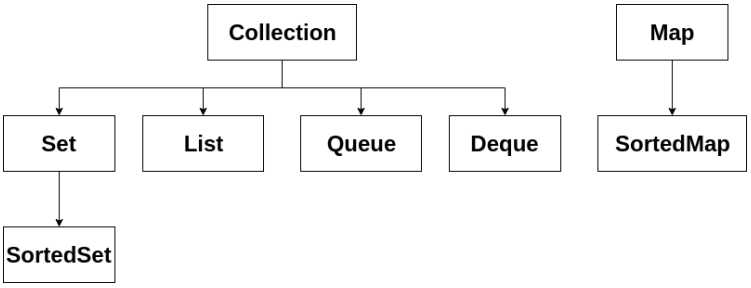

Схема 4: Иерархия основных интерфейсов коллекций.

Collection	Интерфейс, описывающий работу с группами объектов
Set	Неупорядоченные множества
List	Списки
Queue	Очереди
Deque	Двусторонние очереди
SortedSet	Упорядоченные множества
Map	Для хранения пар «ключ-значение»
SortedMap	**Аналог Map, только с отсортированными ключами**

Это только интерфейсы, описывающие структуру коллекций. У каждого интерфейса есть по несколько реализаций (некоторые реализуют сразу несколько интерфейсов).

Также есть вспомогательные интерфейсы, в том числе **Comparable**, **Comparator**, **Iterable**, **Iterator**.

Стоит отметить, что все они являются обобщёнными.

39.2. Интерфейс Collection

Некоторые методы интерфейса Collection:

boolean add(E e)	Добавляет объект в коллекцию
boolean addAll(Collection<? extends E> c)	Добавляет объекты в коллекцию
void clear()	Удаляет все элементы
boolean contains(Object o)	Проверяет, содержится ли объект в коллекции
boolean containsAll(Collection<?> c)	Проверяет, содержатся ли объекты в коллекции
boolean isEmpty()	Проверяет, пустая ли коллекция
boolean remove(Object o)	Удаляет элемент из коллекции
int size()	Возвращает количество элементов
Iterator<E> iterator()	Возвращает итератор для обхода коллекции

39.3. List

List — коллекция, которая может содержать упорядоченный набор элементов.
В дополнении к методам из интерфейса Collection интерфейс List добавляет ещё несколько, в том числе:

void add(int index, E e)	Добавляет на определённую позицию
E get(int index)	Возвращает объект по индексу
int indexOf(Object e)	Возвращает индекс объекта в коллекции
List<E> subList(int start, int end)	Возвращает часть списка в виде нового списка

Реализациями интерфейса являются, например, ArrayList, LinkedList, Stack.
Рассмотрим несколько примеров использования ArrayList.
Создание и заполнение:

```
List<String> stringList = new ArrayList<>();
stringList.add("First");
stringList.add("Second");
```

Пример использования:

```
// По индексу
for(int i = 0; i < stringList.size(); i++){
    System.out.println(stringList.get(i));
}

// Или с помощью for each
for(String s : stringList){
    System.out.println(s);
}
…
if(stringList.contains("NewString")){
    System.out.println("Содержит!");
}
```

39.4. Set

Интерфейс Set расширяет интерфейс Collection, но не добавляет никаких новых методов. Set определяет структуру, которая хранит недублирующиеся элементы. При попытке добавить уже добавленный ранее элемент, метод add() возвращает false.
Рассмотрим пример использования Set на примере HashSet:

```
HashSet<String> set = new HashSet<>();
set.add("First");
set.add("Second");
set.add("Second"); // Второй раз добавлена не будет

...

// У множеств нет индекса,
// так что обход только с помощью for each
for(String s : set){
    System.out.println(s);
}
...
set.remove("NewString"); // Удаление
...
System.out.println(set.size()); // Количество элементов
```

39.5. Queue

Интерфейс Queue описывает поведение очереди. Очередь — абстрактный тип данных с дисциплиной доступа к элементам «первый пришёл — первый вышел» (FIFO, First In — First Out). Добавление элемента (принято обозначать словом enqueue — поставить в

очередь) возможно лишь в конец очереди, выборка — только из начала очереди (что принято называть словом dequeue — убрать из очереди), при этом выбранный элемент из очереди удаляется.

К методам интерфейса Collection интерфейс Queue добавляет несколько методов, в том числе:

boolean offer(E e)	Добавляет элемент в очередь
E peek()	Возвращает элемент из головы очереди. Элемент при этом из очереди не удаляется.
E poll()	Возвращает элемент из головы очереди и удаляет его.

Один из классов, реализующий Queue — LinkedList, который также реализует интерфейс List. Пример использования:

```
LinkedList<String> linkedList = new LinkedList<>();
linkedList.offer("First");
linkedList.offer("Second");

…

System.out.println(linkedList.poll()); // Вернет "First"
System.out.println(linkedList.poll()); // Вернет "Second"
System.out.println(linkedList.poll()); // Вернет null
```

39.6. Map

Map — абстрактный тип данных, позволяющий хранить пары вида «(ключ, значение)» и поддерживающий операции добавления пары, а также поиска и удаления пары по ключу.

Предполагается, что ассоциативный массив не может хранить две пары с одинаковыми ключами.

Важно заметить, что Map не реализует интерфейс Collection, а является самостоятельным.

Методы интерфейса Map:

void clear()	Очищает Map
boolean containsKey(Object key)	Проверяет, содержится ли ключ
boolean containsValue(Object value)	Проверяет, содержится ли Значение
Set<Map.Entry<K,V>> entrySet()	Возвращает множества всех записей. Одна запись — элемент Map.Entry<K,V>, который содержит пару ключ-значение
boolean equals(Object o)	Сравнивает два экземпляра Map
V get(Object key)	Возвращает значение по ключу (null, если такого значения нет).
int hashCode()	Возвращает хешкод
boolean isEmpty()	Возвращает true, если map пуст
Set<K> keySet()	Возвращает множество всех ключей
V put(K key, V value)	Добавляет элемент
void putAll(Map<? extends K,? extends V> m)	Добавить данные из другого объекта Map
V remove(Object key)	Удаляет значение по ключу
int size()	Возвращает количество значений
Collection<V> values()	Возвращает множество всех значений

Некоторые классы, реализующие интерфейс Map — **HashMap**, **Hashtable**, **TreeMap**.

Пример использования Map на примере HashMap[19]:

```
Map<String, Integer> map = new HashMap<String, Integer>();
map.put("Вася", 28); // Добавление элементов
map.put("Петя", 20);
map.put("Маша", 24);
map.put("Паша", 27);

Integer age = map.get("Паша"); // Получение значения по ключу
map.containsKey("Паша"); // Содержится ли ключ
map.containsValue(20); // Содержится ли значение
```

Есть три способа перебрать все значения в Map:

```
// Получить множество всех ключей
for(String key : map.keySet()){
    System.out.println(map.get(key)); // Получить значение по ключу
}

// Перебирать сразу множество значений
for(Integer value : map.values()){
    System.out.println(value);
}

// Получить множество всех записей Map.Entry<String, Integer>
for(Map.Entry<String, Integer> pair : map.entrySet()){
    // Каждая запись содержит ключ и значение:
    System.out.println(pair.getKey() + " " + pair.getValue());
}
```

39.7. Коллекции и вычислительная сложность

Выбор реализации во многом зависит от контекста: какие будут данные, как будет использоваться коллекция и так далее. Вычислительная сложность операций для некоторых коллекций приведена в таблице ниже:

List:

	Добавление	Удаление	Получение
ArrayList	O(1)	O(n)	O(1)
LinkedList	O(1)	O(1)	O(n)

Set:

	Добавление	Удаление	Получение
HashSet	O(1)	O(1)	O(h/n)
LinkedHashSet	O(1)	O(1)	O(1)
EnumSet	O(1)	O(1)	O(1)
TreeSet	O(log n)	O(log n)	O(log n)

Map:

	Добавление	Получение
HashMap	O(1)	O(1)
LinkedHashMap	O(1)	O(1)
TreeMap	O(log n)	O(log n)

19. См. пример про построение частотного словаря русских символов на странице 76.

39.8. Iterator и Iterable

Интерфейсы Iterator и Iterable описывают поведение объекта, содержащего какое-то множество значений, и которые можно перебрать.

В интерфейсе Iterable объявлен один обязательный к реализации метод:

```
public interface Iterable<T> {

    Iterator<T> iterator();
    …
}
```

То есть, один метод, возвращающий экземпляр интерфейса Iterator.

В интерфейсе Iterator объявлено два обязательных к реализации метода:

```
public interface Iterator<E> {

    boolean hasNext(); // Есть ли следующий
    E next(); // Вернуть следующий
}
```

Пример использования итератора[20]:

```
List<String> list = new ArrayList<String>();

....

// Collection наследует интерфейс Iterable
Iterator<String> iterator= list.iterator();

while (iterator.hasNext()){ // Цикл, пока есть следующий
    System.out.println(iterator.next()); // Получить и вывести
}
```

39.9. Iterable и foreach

Экземпляр класса, реализующего интерфейс Iterable можно использовать в циклах foreach. Так как Collection наследует Iterable, то в таком цикле можно использовать любую коллекцию:

```
for(String s : list){
    ...
}
```

Таким образом, в циклах foreach можно использовать:

- Массивы
- Реализации интерфейса Iterable

39.10. Comparator и Comparable

Интерфейсы Comparator и Comparable используются для сравнения объектов. Эти интерфейсы используются, например, при сортировке объектов.

Интерфейс Comparable имеет один метод:

```
public interface Comparable<T> {

    public int compareTo(T o);

}
```

Метод *compareTo(T o)* возвращает отрицательное число, ноль или положительное число, если объект меньше, равен или больше чем объект, переданный как параметр.

К примеру, если необходимо каким-то образом сравнивать по порядку экземпляры класса User, то можно написать так:

20. Более сложный пример см. на странице 78.

```
class User implements Comparable<User>{
    private String name;

    @Override
    public int compareTo(User user) {
        return name.compareTo(user.name);
    }
}
```

Метод compareTo описывает, по какому признаку будут сравниваться экземпляры класса User. В данном примере, вызывается тот же метод compareTo, только у поля name. Теперь можно отсортировать массив таких объектов:

```
Arrays.sort(users);
```

Если по каким-то причинам невозможно определить метод *compareTo()*, то можно использовать дополнительный класс, реализующий интерфейс Comparator<T>. Например, невозможно переопределить метод *compareTo()* в классе String, поэтому для нестандартной сортировки строк можно использовать отдельный класс, наследующий Comparator<T>.

Пример сортировки строк по их длине с помощью отдельной реализации Comparator<T> можно посмотреть на странице 75.

39.11. Класс Collections

В пакете java.util можно найти класс Collections с набором статических методов. В них реализованы алгоритмы, связанные с коллекциями, такие как сортировка, бинарный поиск и другие.

Некоторые методы из класса Collections:

static <T> int binarySearch(List<? extends Comparable<? super T>> list, T key)	Реализация бинарного поиска. Возвращает индекс найденного элемента
static <T> int binarySearch(List<? extends T> list, T key, Comparator<? super T> c)	Реализация бинарного поиска. Возвращает индекс найденного элемента
static <T> void copy(List<? super T> dest, List<? extends T> src)	Копирует все элемент из одного листа в другой
static boolean disjoint(Collection<?> c1, Collection<?> c2)	Возвращает true, если коллекции не содержат общего элемента
static <T> List<T> emptyList() static <T> Iterator<T> emptyIterator() static <K,V> Map<K,V> emptyMap() static <T> Set<T> emptySet()	Возвращает «пустые» реализации интерфейсов
static int frequency(Collection<?> c, Object o)	Возвращает количество вхождений объекта в коллекцию
static int indexOfSubList(List<?> source, List<?> target)	Индекс, с которого в списке начинается другой список.
static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> coll)	Поиск максимума
static <T> T max(Collection<? extends T> coll, Comparator<? super T> comp)	Поиск максимума
static <T extends Object & Comparable<? super T>> T min(Collection<? extends T> coll)	Поиск минимума
static <T> T min(Collection<? extends T> coll, Comparator<? super T> comp)	Поиск минимума
static void reverse(List<?> list)	Меняет порядок следования элементов в List на обратный
static void shuffle(List<?> list)	Перемешивает все элементы в List
static <T extends Comparable<? super T>> void sort(List<T> list)	Сортировка
static <T> void sort(List<T> list, Comparator<? super T> c)	Сортировка

40. Лямбда-выражения

Лямбда-выражение — специальный синтаксис для определения функциональных объектов. Применяется для объявления анонимных функций по месту их использования. Используя лямбда-выражения, можно объявлять функции в любом месте кода.

Рассмотрим такой код:

```
Runnable runnable = new Runnable() { // Анонимный класс
    @Override
    public void run() {
        System.out.println("Other thread");
    }
};

Thread thread = new Thread(runnable);
thread.start(); // Запуск потока
```

Здесь создаётся экземпляр анонимного класса, реализующего Runnable, чтобы создать поток. Можно переписать этот код с использованием лямбда-выражения:

```
Thread thread = new Thread(() -> System.out.println("Other thread"));
thread.start(); // Запуск потока
```

Очевидно, что второй пример кода короче и лучше читаем.

40.1. Лямбда-оператор

Лямбда-выражение состоит из двух частей — списка параметров и тела выражения. Эти две части разделяются лямбда-оператором (->). В примере выше список параметров пустой (), а тело состоит из одного вызова метода.

(список параметров) -> {тело выражения}

40.2. Функциональный интерфейс

В Java 1.8 вводится понятие функционального интерфейса. Функциональный интерфейс — интерфейс ровно с одним абстрактным методом.

Можно заменить реализацию такого интерфейса лямбда-выражением всякий раз, когда ожидается объект интерфейса с одним абстрактным методом.

Например, интерфейс Runnable:

```
public interface Runnable {
    public abstract void run();
}
```

В нем определён один только метод, именно поэтому, код из предыдущего примера работает.

Или, вместо реализации метода ***compare*** интерфейса **Comparator**, можно использовать лямбда-выражение:

```
Arrays.sort(strings, new Comparator<String>() {
    @Override
    public int compare(String o1, String o2) {
        return o1.length() - o2.length();
    }
});

// Или, эквивалентно:
Arrays.sort(strings, (o1, o2) -> o1.length() - o2.length());
```

Так как лямбда-выражения связаны с функциональными интерфейсами, то можно присваивать лямбда-выражение переменной имеющий тип такого интерфейса:

```
Runnable runnable = () -> System.out.println("Other thread");
```

В каком-то смысле, лямбда-выражения это альтернатива созданию анонимного класса.

40.3. @FunctionalInterface

В Java 1.8 появляется аннотация @FunctionalInterface. Ею можно аннотировать интерфейсы. В таком случае компилятор проверит, что такой интерфейс действительно является функциональным (то есть имеет один абстрактный метод). При этом интерфейс, содержащий только один абстрактный метод, считается функциональным независимо от наличия аннотации.

40.4. Лямбда-выражения и параметры

У метода *compare()* есть два параметра, именно поэтому в лямбда-выражении в списке параметров их тоже два: (o1, o2). Если компилятор может определить типы параметров лямбда-выражения, можно опустить их, как сделано в примере выше.

Если в методе интерфейса нет параметров, все равно нужно указать круглые скобки.

Если в методе один параметр, то скобки можно опустить:

```
doSome(value -> System.out.println("Printed value"));
```

Если тело лямбда-выражения является однострочным и результат выражения соответствует возвращаемому типу, то return можно опустить (как в примере с Comparator). Однако, можно написать так, если это необходимо:

```
Arrays.sort(strings, (o1, o2) -> {
    int result = o1.length() - o2.length();
    return result;
});
```

40.5. Обобщённые функциональные интерфейсы

Если интерфейс является обобщённым, то тип возвращаемого значения лямбда выражения определяются исходя их других параметров или явного указания типа параметра при использовании ссылки на лямбда-выражение.

40.6. Ссылки на методы

Иногда уже существует метод, который делает именно то, что нужно, но чтобы его переиспользовать, его надо было вызвать явно. Допустим, что определён некий класс с методом void без параметров:

```
class MyClass {

    public void doSome() {
        ...
    }
}
```

Чтобы вызвать этот метод в другом потоке, необходимо написать следующее:

```
Thread thread = new Thread(new Runnable() {
    @Override
    public void run() {
        myClass.doSome();
    }
});
…
// Или с лямбда-выражением:
Thread thread = new Thread(() -> myClass.doSome());
```

Для упрощения таких конструкций существуют ссылки на методы:

```
Thread thread = new Thread(myClass::doSome);
```

С помощью оператора «::» можно получить ссылку на метод, если у него такая же сигнатура, какая требуется.

Для получения ссылок на методы также можно использовать ключевые слова this и super:

```
Thread thread = new Thread(this::doSome);
```

41. Небольшой пример работы с сетью[21]

В пакете java.net содержатся классы для работы с сетью, в том числе для работы с сокетами, с TCP/IP (в том числе HTTP, HTTPS) и так далее. Набор инструментов в Java слишком обширен, чтобы уместить его в это пособие. Более того, Герберт Шилдт в своей книге о Java 8 (Java: The Complete Reference) говорит о том, что в его книге (1376 страниц) для этого тоже не хватило места. Однако, после изучение примеров из этого пособия рекомендуется углубиться в пакет java.net с помощью упомянутой книги, после чего уже можно изучать статьи о специфических инструментах по необходимости.

41.1. Класс URL

Для работы с ресурсами, заданными адресами URL, имеется класс URL.
URL — Единый указатель ресурса (англ. Uniform Resource Locator) — единообразный локатор (определитель местонахождения) ресурса.
Общепринятые схемы (протоколы) URL включают ftp, http, mailto, file и так далее.

С помощью класса URL можно посмотреть свойства URL или получить экземпляр класса URLConnection:

```
URL url = new URL("http://en.wikipedia.org/wiki/Java");
System.out.println(url.getHost());
System.out.println(url.getPath());
System.out.println(url.getProtocol());
```

В результате на консоль будет выведено:

```
en.wikipedia.org
/wiki/Java
http
```

С помощью метода *openConnection()* можно получить экземпляр класса URLConnection, который предназначен для удалённого подключения:

```
URLConnection urlConnection = url.openConnection();
```

Если для протокола (такого как HTTP) существует публичный, специальный подкласс входящий в пакет java.lang, java.io, java.util или java.net, то будет возвращён его экземпляр (например HttpURLConnection).

41.2. HttpURLConnection

Если используется протокол HTTP (HTTPS), то метод *openConnection()* вернёт экземпляр HttpURLConnection (HttpsURLConnection). В нем определено несколько методов, в том числе:

int getContentLength()	Возвращает длину в байтах
String getContentType()	Возвращает ContentType
long getDate()	Возвращает время и дату ответа
String getHeaderField(int n)	Возвращает поле заголовка по индексу
Map<String,List<String>> getHeaderFields()	Возвращает все поля заголовка
InputStream getInputStream()	Возвращает поток данных

Результатом такого кода:

```
URL url = new URL("https://yandex.ru");
URLConnection urlConnection = url.openConnection();

System.out.println(urlConnection.getDate());
System.out.println(urlConnection.getContentType());
System.out.println(urlConnection.getContentLength());

Map<String,List<String>> fields = urlConnection.getHeaderFields();
```

21. См. пример «Получение данных из OpenStreetMap» на странице 82.

```
for(Map.Entry<String,List<String>> field : fields.entrySet()){
    System.out.println(field.getKey() + ": " + field.getValue());
}
```

Будет вывод:

```
1511790183000
text/html; charset=UTF-8
90999
null: [HTTP/1.1 200 Ok]
X-Content-Type-Options: [nosniff]
Last-Modified: [Mon, 27 Nov 2017 13:43:03 GMT]
P3P: [policyref="/w3c/p3p.xml", CP="NON DSP ADM DEV PSD IVDo OUR IND STP PHY PRE NAV UNI"]
Date: [Mon, 27 Nov 2017 13:43:03 GMT]
X-Frame-Options: [DENY]
Cache-Control: [no-cache,no-store,max-age=0,must-revalidate]
...
Expires: [Mon, 27 Nov 2017 13:43:03 GMT]
Content-Length: [90999]
X-XSS-Protection: [1; mode=block]
Content-Type: [text/html; charset=UTF-8]
```

Пожалуй, самым интересным методом является метод openConnection(). Метод возвращает экземпляр InputStream для чтения. Получив InputStream, с ним можно работать как с обычным потоком ввода/вывода. Зная, что содержимым является text/html, можно прочесть содержимое как текст:

```
URL url = new URL("https://www.yandex.ru");
URLConnection urlConnection = url.openConnection();
String line;
InputStream inputStream = urlConnection.getInputStream();
InputStreamReader reader = new InputStreamReader();
BufferedReader bufferedReader = new BufferedReader(reader);

while ((line = bufferedReader.readLine()) != null) {
    System.out.println(line);
}
```

В результате в консоли можно увидеть содержимое страницы https://www.yandex.ru.

42. Более сложные примеры

42.1. Синус

Вычисление примерного значения sin(x) используя ряд Маклорена:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!}, x \in \mathbb{C}$$

```
double x = 180;
// Градусы в радианы
x = x % (2 * Math.PI);

double term = 1.0;
double sum = 0.0;       // Сумма предыдущих членов ряда

for (int i = 1; term != 0.0; i++) {
    term *= (x / i);
    if (i % 4 == 1) sum += term;
    if (i % 4 == 3) sum -= term;
}
System.out.println(sum);
```

42.2. Счастливый билет

Метод, который на входе получает шестизначное число и проверяет, является ли данный билет счастливым:

```
public boolean isHappy(int n) {
    // Проверяем, что число шестизначное
    if (n < 100000 || n > 999999)
        return false;

    // Сумма разрядов первой и второй половин
    int half1 = (n/100000)%10 + (n/10000)%10 + (n/1000)%10;
    int half2 = (n/100)%10 + (n/10)%10 + n%10;
    return half1 == half2; // проверка равенства
}
```

42.3. Убрать элементы из массива

Дан массив и значение. Метод удаляет все вхождения этого элемента из массива, и возвращает новую длину массива:

```
public int removeElement(int[] nums, int val) {
    // Будем смещать элементы "назад", на освободившиеся места
    int offset = 0;

    // Перебираем все элемент массива
    for (int i = 0; i < nums.length; i++) {

        if (nums[i] == val) {  // Если нашлось вхождения
            offset++; // Стало на один элемент меньше, offset увеличивается
        } else {
            nums[i - offset] = nums[i]; // смещаем элемент на размер offset позиций
        }
    }
    return nums.length - offset; // Длина исходного массива - количество удаленных объектов
}
```

42.4. Найти наибольший элемент в массиве

```
public int max(int array[]) {
    int max = array[0]; // Будущий максимум

    // Перебор с помощью foreach
    for(int value : array){
        if(max > value){ //Если нашелся элемент больше max
            max = value;
        }
    }

    return max;
}
```

42.5. Найти наибольший элемент в двумерном массиве

```
public int findMax(int[][] array) {
    int max = 0;

    // Проходим по строкам
    for (int i = 0; i < array.length; i++) {
        // Проходим по всем элементам строки
        for (int j = 0; j < array[i].length; j++) {
            if (array[i][j] > max) // если нашли новый max
                max = array[i][j];
        }
    }
    return max;
}
```

42.6. Максимальная сумма

В двумерном массиве найти строку, сумма элементов которой является максимальной среди всех строк матрицы:

```
public int[] maxRow(int[][] array) {
    int maxSum = 0; // Последняя наибольшая сумма
    int index = 0;

    // Перебираем строки
    for (int i = 0; i < array.length; i++) {
        int newSum = 0;
        // Ищем сумму элементов строки
        for (int j = 0; j < array[i].length; j++)
            newSum += array[i][j];

        // Если новая сумма больше запоминаем сумму и индекс
        if (newSum > maxSum) {
            maxSum = newSum;
            index = i;
        }
    }
    return array[index]; // Вернуть строку по индексу
}
```

42.7. Класс вектор

Класс, описывающий вектор в трёхмерном пространстве, с методами для расчёта угла между векторами, для умножения на другой вектор и вычисления длины вектора:

```
public class Vector {
    // Три приватных поля
    private double x;
    private double y;
    private double z;

    public Vector() { // Конструктор без параметров
        this(0, 0, 0); // Вызывается другой конструктор
    }

    // С трема параметрами
    public Vector(double x, double y, double z) {
        this.x = x;
        this.y = y;
        this.z = z;
    }

    public Vector(Vector vector) { // Конструктор
        this(vector.x, vector.y, vector.z); // Вызывается другой конструктор
    }

    public double getX() {
        return x;
    }

    public double getY() {
        return y;
    }

    public double getZ() {
        return z;
    }



    // Длина вектора. Корень из суммы квадратов
    public double length() {
        return Math.sqrt(x * x + y * y + z * z);
    }

    // Умножить на другой вектор
    public double multiply(Vector vector) {
        return x * vector.x + y * vector.y + z * vector.z;
    }

    // Косинус между двумя векторами
    public double cos(Vector vector) {
        // Для вычисления длины и произведения используются
        //методы multiply и length
        return multiply(vector) / (length() * vector.length());
    }
}
```

42.8. Склеить все строки из массива через запятую

```
public String concatAll(String[] strings){
    String result = ""; // изначально пустая строка

    for(int i = 0; i < strings.length; i++){
        // на каждой итерации цикла добавляем новую строку
        result = result + strings[i];

        // Проверяем, чтобы элемент был не последний
        if(i < strings.length - 1){
            // Если не последний, то еще прибавим запятую
            result = result + ", ";
        }
    }

    return result;
}
```

Более правильное решение с помощью StringBuilder:

```
public String concatAll(String[] strings){
    // изначально пустая строка
    StringBuilder result = new StringBuilder();

    for(int i = 0; i < strings.length; i++){
        // на каждой итерации цикла добавляем новую строку
        result.append(strings[i]);
        // Проверяем, чтобы элемент был не последний
        if(i < strings.length - 1){
            // Если не последний, то еще прибавим запятую
            result.append(", ");
        }
    }
    return result.toString();
}
```

42.9. Проверить, является ли слово палиндромом

Решение с помощью StringBuilder:

```
private boolean IsPolyndrome(String inputString) {
    boolean result = false;
    StringBuilder sb = new StringBuilder(inputString);
    if (inputString.equals(sb.reverse().toString())) {
        result = true;
    }
    return result;
}
```

Решение с помощью сравнения симметричных символов попарно:

```
private boolean IsPolyndrome(String inputString) {
    for (int i = 0; i < inputString.length(); i++) {
        if (inputString.charAt(i) != inputString.charAt(inputString.length() - 1 - i))
            return false;
    }
    return true;
}
```

42.10. Поиск длиннейшей строки:

```
public String getStringWithMaxLength(String[] strings){

    if(strings.length == 0){
        return null;
    }

    String maxLenght = strings[0];

    for(int i = 0; i < strings.length; i++){
        if(maxLenght.length() < strings[i].length()){
            maxLenght = strings[i];
        }
    }

    return maxLenght;
}
```

42.11. Количество подстрок

Метод, возвращающий количество вхождений одной строки в другую. Решение:

```
public int countSubStrings(String text, String subString) {
    int index = 0; // Последний индекс начала подстроки
    int count = 0; // Количество

    while (index != -1) {
        // Индекс первого вхождения подстроки
        index = text.indexOf(subString, index);

        // Если нашлась подстроки
        if (index != -1) {
            count++; // Количечество увеличивается на 1
            index += subString.length(); // Индекс для следующего поиска
        }
    }
    return count;
}
```

42.12. Сравниваемый класс User

Класс User, объекты которого будут сравнимы по имени:

```
class User implements Comparable<User>{
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public User(String name) {
        this.name = name;
    }

    /* Метод возвращает отрицательно, ноль и или положительное
     * число, если объект меньше, равен или больше аргумента */
    public int compareTo(User user) {
        return name.compareTo(user.getName()); // Вызываем аналогичный метод только у String
    }
}
```

42.13. У каких пользователей день рождения

Метод, который находит количество пользователей родившихся в определённом месяце.

```
class User {
    private String name;
    private Date birthDay;

    public Date getBirthDay() {
        return birthDay;
    }
}
```

```
private int usersWithBirthDay(User[] users, int month){
    int count = 0;
    // Объект Calendar будет переиспользоваться
    Calendar userCalendar = Calendar.getInstance();

    for(User user : users){ // Перебор
        // Обновляем userCalendar
        userCalendar.setTime(user.getBirthDay());
        if(userCalendar.get(Calendar.MONTH) == month){
            count++; // Увеличиваем счётчик, если месяца совпали
        }
    }
    return count;
}
```

42.14. Сортировка строк по длине

```
// В метод Arrays.sort передаётся два параметра:
// 1. Массив
// 2. Экземпляр анонимного класса
Arrays.sort(strings, new Comparator<String>() {
    @Override
    public int compare(String s1, String s2) {
        // Используем метод Integer.compare()
        // для сравнения двух длин
        return Integer.compare(s1.length(), s2.length());
    }
});
```

42.15. Фильтр

Метод filter(), который фильтрует входной список объектов по признаку, описанному во втором параметре — реализации интерфейса Filter.

```
interface Filter<T>{
    // Вернуть true если объект должен остаться в списке
    boolean apply(T t);
}
```

```
public <T> List<T> filter(Collection<T> collection,
                          Filter<? super T> filter){
    // Создается новый список
    List<T> listFiltered = new ArrayList<>();
    for(T t : collection){ // Перебирается

        // Если apply(t) == true -> добавить в новый список
        if(filter.apply(t)){
            listFiltered.add(t);
        }
    }

    return listFiltered;
}
```

```
// Пример использования:
ArrayList<String> list = new ArrayList<>();
...
// Передаётся list и экземпляр Filter
filter(list, new Filter<String>() {
    @Override
    public boolean apply(String s) {
        return !s.isEmpty(); // Пустые строки не нужны
    }
});
```

42.16. Построение частотного словаря русских символов

```
/*
 Метод получает на вход путь к каталогу, который содержит
 текстовые файлы - документы, книги и так далее.
*/
public Map<Character, Integer> buildFrequencies(String directory) {

    // Будем заполнять этот Map
    Map<Character, Integer> characterIntegerMap =
            new HashMap<>();

    // Список файлов каталога
    File[] files = new File(directory).listFiles();

    for (File file : files) { // Проходим по всем файлам

        // Каждый файл читаем по строкам
        try (BufferedReader br = new BufferedReader(new
                FileReader(file))) {
            String line;
            // Читаем, пока  br.readLine() не вернет null
            while ((line = br.readLine()) != null) {
                for(char c : line.toLowerCase().toCharArray(){
                    // Получаем массив символов
                    // Нужны только русские буквы
                    if(c >= 'а' && c <= 'я'){
                        // Получаем предыдущее количество или
                        // считаем, что оно равно 0
                        Integer count =
                            characterIntegerMap.containsKey(c) ? characterIntegerMap.get(c) : 0;

                        // Добавляем пару (символ, количество)
                        characterIntegerMap.put(c, count + 1);
                    }
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    return characterIntegerMap;
}
```

42.17. Реализация односвязного списка

```
/**
 * Реализация односвязного списка
 */
public class List<T> {

    // Вспомогательный класс для узла
    private static class Node<T> {
        private Node next; // Ссылка на следующий узел
        private T value; // Значение

        public Node<T> getNext() {
            return next;
        }

        public void setNext(Node next) {
            this.next = next;
        }

        public T getValue() {
            return value;
        }

        // Конструктор
        public Node(T value) {
            this.value = value;
        }
    }

    // Ссылка на первый узел
    private Node<T> first;

    // Ссылка на последний узел
    private Node<T> last;

    // Размер
    private int size;

    // Добавить элементыы
    public void add(T... s) {
        T[] arr = s;
        for (T var : s) {
            add(var);
        }
    }

    // Получить элемент по индексу
    public T get(int index) {
        return getNode(index).getValue();
    }

    private void add(T s) {

        // Если узел езе пуст создать первый узел
        if (first == null) {
            first = new Node<T>(s);
            last = first;
        } else {
            // Если лист не пуст, добавить новый узел
            Node<T> newNode = new Node<>(s);

            // Старый последний ссылается на новый последний
            last.setNext(newNode);
            last = newNode; // Обновили последний узел
        }
        size++; // Увеличили размер
    }
```

```
    // Получить узел по индексу
    private Node<T> getNode(int index) {
        Node<T> last = first;

        // Проходим к следующему узлу index раз
        for (int i = 0; i < index; i++) {
            last = last.getNext();
        }
        return last;
    }

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

42.18. Пример использования:

```
// Пример использования:
List<String> list = new List<>();
list.add("1");
list.add("2");
list.add("One more");
for(int i = 0; i < list.size(); i++){
    System.out.println(list.get(i));
}
```

42.19. Итератор по двумерному массиву

Класс, содержащий двумерный массив и реализующий интерфейс Iterable:

```
public class ArrayWrapper<T> implements Iterable<T> {

    private T[][] array; // Массив

    // Конструктор
    public ArrayWrapper(T[][] array) {
        this.array = array;
    }

    @Override
    public Iterator<T> iterator() {
        // Вернуть новый экземпляр анонимного класса
        return new Iterator<T>() {

            // Последняя пара индексов
            private int n = 0;
            private int m = 0;

            @Override
            public boolean hasNext() {
                // Надо посмотреть, есть ли элименты после
                // последнего возвращенного
                for (int i = m; i < array.length; i++) {
                    for (int j = n; j < array[i].length; j++) {
                        return true; // есть
                    }
                }
                return false; // нет
            }
```

```
            @Override
            public T next() {

                // В документации описано, что необходимо
                //"бросить" исключение если больше нет элементов
                if (!hasNext()) {
                    throw new NoSuchElementException();
                }

                T next = array[m][n]; // Следующее значение

                // Увеличим индекс
                if (n < array[m].length - 1) {
                    n++;
                } else { // Новая строка
                    n = 0;
                    m++;
                }
                return next;
            }
        };
    }
}
```

Пример использования:

```
String array[][] = new String[][]{{"1","2","3"}, {"4","5","6"}, {"7","8","9"}};
ArrayWrapper<String> arrayWrapper = new ArrayWrapper<>(array);

for (String s : arrayWrapper) {
    System.out.println(s);
}
```

42.20. Производитель-потребитель

Одна из классических задач по многопоточности. Дано два потока — производитель и потребитель. Производитель генерирует некоторые данные (в примере — числа). Производитель забирает их.

Два потока разделяют общий буфер данных, размер которого ограничен. Если буфер пуст, потребитель должен ждать, пока там появятся данные. Если буфер заполнен полностью, производитель должен ждать, пока потребитель заберёт данные и место освободится.

Это классическая задача синхронизации двух потоков.

Производитель:

```
// implements Runnable чтобы запускать в отдельном потоке
public class Producer implements Runnable {

    // Общая очередь
    private final Queue<Double> sharedQueue;

    // Максимальный размер
    private final int SIZE;

    // Конструктор
    public Producer(Queue<Double> sharedQueue, int size) {
        this.sharedQueue = sharedQueue;
        this.SIZE = size;
    }
```

```
    @Override
    public void run() {
        // Цикл бесконечен
        while (true) {
            try {
                // В цикле вызывается метод produce
                // со случайным аргументом
                produce(Math.random());
                System.out.println("Produced");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    private void produce(double i) throws InterruptedException {
         /*
             Методы wait() и notifyAll() можно вызвать только внутри synchronized-блока
             В качестве монитора будет использовать общий объект - очередь
          */
        synchronized (sharedQueue) { // обязательно synchronized
            if (sharedQueue.size() == SIZE) {
                System.out.println("Queue is full");
                // Если очередь наполнена, то ждём
                sharedQueue.wait();
            }
        }

        synchronized (sharedQueue) {
            // Добавили элемент в очередь.
            sharedQueue.add(i);

            // Уведомили другой поток на случай, если он ждет
            sharedQueue.notifyAll();
        }
    }
}
```

Потребитель:

```
// implements Runnable чтобы запускать в отдельном потоке
public class Consumer implements Runnable {

    // Общая очередь
    private final Queue<Double> sharedQueue;

    // Конструктор
    public Consumer(Queue<Double> sharedQueue) {
        this.sharedQueue = sharedQueue;
    }




    @Override
    public void run() {
        while (true) {
            try {
                // Вывести потребленное
                System.out.println("Consumed: " + consume());
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
        }
    }
```

```
    // Метод, извлекающий элементы из общей очереди
    private Double consume() throws InterruptedException {
        if (sharedQueue.isEmpty()) { // Если пуста, надо ждать
            synchronized (sharedQueue) {
                System.out.println("Queue is empty");
                sharedQueue.wait();
            }
        }

        // Извлекается элемент
        synchronized (sharedQueue) {
            sharedQueue.notifyAll();

            // Уведомили другой поток на случай, если он ждёт
            return sharedQueue.poll();
        }
    }
}
```

Создание и запуск:

```
LinkedList<Double> sharedQueue = new LinkedList<>();
int size = 4;
Thread prodThread = new Thread(new Producer(sharedQueue, size), "Producer");
Thread consThread = new Thread(new Consumer(sharedQueue), "Consumer");
prodThread.start();
consThread.start();
```

42.21. Взаимная блокировка

Ниже продемонстрирована взаимная блокировка двух потоков.

Первый поток сначала заходит в synchronized-блок по resource1, а потом пытается зайти в synchronized-блок по resource2. При этом он не может попасть во второй блок, потому что второй поток захватил resource2 и не освободит его пока не попадёт в synchronized-блок по resource1 (который занят первым потоком). В итоге, эта программа «зависнет».

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

        final Object resource1 = "resource1";
        final Object resource2 = "resource2";

        Thread t1 = new Thread() {
            public void run() {


                synchronized (resource1) {
                    synchronized (resource2) {
                      ...
                    }
                }
            }
        };

        Thread t2 = new Thread() {
            public void run() {

                synchronized (resource2) {
                    synchronized (resource1) {
                      ...
                    }
                }
            }
        };

        t1.start();
        t2.start();
    }
}
```

42.22. Получение данных из OpenStreetMap

Программа, которая делает запрос к сервису Nominatim[22], который позволяет производит поиск по картам OpenStreetMap[23].
Ответ от сервера приходит в формате JSON[24].
Программа использует библиотеку GSON[25] от Google, которая сериализует и десериализует объекты Java в GSON и наоборот.
Программа в цикле считывает адрес, делает запрос к серверу и выводит на консоль список объектов, которые находятся по этому адресу.

```
public static void main(String[] strings) throws IOException{

    class Place { // Можно использовать локальный класс
        @SerializedName("display_name")
        private String displayName;

        public String getDisplayName() {
            return displayName;
        }
    }

    // общий вид ссылки, согласно протоколу. street=%s
    // с помощью String.format %s заменяется на адрес
    final String link = "http://nominatim.openstreetmap.org/search?" +
            "street=%s&format=json&city=СПб";

    Scanner scanner = new Scanner(System.in); // для консоли

    // для того, чтобы парсить json. См. библиотеку gson
    Gson gson = new Gson();

    Type listType = new TypeToken<ArrayList<Place>>() {}.getType(); // Тип для Gson

    while (true) {
            System.out.println("Enter address here:");
            String street = scanner.nextLine();  // просим ввести адрес и получаем его

          /*
              убираем все пробелы и прочие знаки. То есть из
                "биржевая линия" получаем
              %D0%B1%D0%B8%D1%80%D0%B6%D0%B5%D0%B2%D0%B0%D1%8F
                +%D0%BB%D0%B8%D0%BD%D0%B8%D1%8F.
           */
            street = URLEncoder.encode(street, "UTF-8");

            // создаём наш объект URL используя link.
            // Заменяем там параметр на наш адрес.
            URL url = new URL(String.format(link, street));

            URLConnection urlConnection = url.openConnection();// подключились

            // Получили InputStream из urlConnection
            InputStream inputStreamFromServer = urlConnection.getInputStream();

            // распарсили овтет с помощью библиотеки,
            // все поля из json были перенесены в объекты Place
            InputStreamReader reader = new InputStreamReader(inputStreamFromServer);
            List<Place> places = gson.fromJson(reader, listType);

            for (Place pl : places) {
                // Просто вывод результата
                System.out.println(pl.getDisplayName());
            }
    }
}
```

22. http://wiki.openstreetmap.org/wiki/Nominatim
23. http://ru.wikipedia.org/wiki/OpenStreetMap
24. http://ru.wikipedia.org/wiki/JSON
25. http://github.com/google/gson

43. Список рекомендуемой дополнительной литературы

- Герберт Шилдт — Java 8. Полное руководство.
- Joshua Bloch — Effective Java
- Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Влиссидес — Приёмы объектно-ориентированного проектирования. Паттерны проектирования.
- Vijay K. Garg — Concurrent and Distributed Computing in Java.