

Министерство образования Республики Беларусь  
Учреждение образования  
«Белорусский государственный университет  
информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра программного обеспечения  
информационных технологий

**О. Г. Смолякова**

## **ОСНОВЫ РАЗРАБОТКИ ВЕБ-ПРИЛОЖЕНИЙ НА ЯЗЫКЕ ПРОГРАММИРОВАНИЯ JAVA**

*Рекомендовано УМО по образованию  
в области информатики и радиоэлектроники  
в качестве учебно-методического пособия  
для специальности 1-40 01 01  
«Программное обеспечение информационных технологий»*

УДК 004.774(076)

ББК 32.973.4я73

C51

Рецензенты:

кафедра веб-технологий и компьютерного моделирования  
Белорусского государственного университета  
(протокол №2 от 03.10.2018);

старший специалист отдела по подготовке персонала  
ИООО «ЭПАМ Системз» В. С. Шульга

**Смолякова, О. Г.**

C51      Основы разработки веб-приложений на языке программирования Java : учеб.-метод. пособие / О. Г. Смолякова. – Минск : БГУИР, 2019. – 131 с. : ил.

ISBN 978-985-543-495-6.

Содержит материалы к практическим занятиям, включающие вопросы и задания по учебной дисциплине «Веб-технологии».

УДК 004.774(076)

ББК 32.973.4я73

**ISBN 978-985-543-495-6**

© Смолякова О. Г., 2019

© УО «Белорусский государственный  
университет информатики  
и радиоэлектроники», 2019

## **СОДЕРЖАНИЕ**

1. ВВЕДЕНИЕ В JAVA PROGRAMMING LANGUAGE.....	4
2. КЛАССЫ И ОБЪЕКТЫ.....	43
3. НАСЛЕДОВАНИЕ И ИНТЕРФЕЙСЫ.....	67
4. LAYERED ARCHITECTURE – «РАССЛОЕНИЕ» СИСТЕМЫ.....	107
ЛИТЕРАТУРА.....	130

Библиотека БГУИР

# 1. ВВЕДЕНИЕ В JAVA PROGRAMMING LANGUAGE

Вопросы в начале темы:

- Как вы понимаете, в чем состоит отличие объектно-ориентированных языков программирования?
- Как вы понимаете, что такое «тип данных».
- Объясните, что такое Java-платформа?

## Основные термины и аббревиатуры

**Java** – это *объектно-ориентированный, платформенно-независимый язык программирования, используемый для разработки информационных систем*, работающих в сети *Internet*, а также это *вычислительная платформа*. **Java Platform** (Java-платформа) – набор спецификаций и программных продуктов, совместно представляющих собой систему для разработки программного обеспечения. Реализация Java Platform представлена в виде трех семейств (вариантов) технологий: Java SE (Oracle Java Platform, Standard Edition), Java EE (Java Platform, Enterprise Edition), Java ME (Java Platform, Micro Edition). **Java SE** – стандартная версия Java Platform, предназначенная для разработки и использования приложения индивидуального пользования или приложений, работающих в масштабах малых предприятий. **Java EE** – набор спецификаций и документации, описывающих архитектуру серверной платформы для задач средних и крупных предприятий; поставляется как бесплатный комплект разработки SDK (Software Development Kit), например Java EE 8 Platform SDK. Однако для запуска и работы Java EE-приложений необходим Java EE Application Server (GlassFish, JBoss, Tomcat и др.). **Java ME** – версия Java Platform для разработки приложений, работающих на устройствах с ограниченными ресурсами.

**JLS (Java Language Specification)** – техническое описание синтаксиса и семантики языка программирования Java. **Java API (Application Program Interface)**, или library (библиотека), содержит предопределенные классы и интерфейсы, используемые для разработки Java-программ. Есть три типа Java API: официальный основной Java API, предоставляемый с JDK или JRE в одном из вариантов Java Platform; дополнительные официальные API, загружаемые отдельно (их спецификации определяются в соответствии с JSR (Java Specification Request)); неофициальные API, разработанные третьими сторонами и не связанные в JSR.

**Javadoc (Java Documentation)** – документация Java API, созданная по определенным правилам (рис. 1).

The screenshot shows a web browser displaying the 'Overview (Java SE 9 & JDK 9)' page at <https://docs.oracle.com/javase/9/docs/api/overview-summary.html>. The page title is 'Java® Platform, Standard Edition & Java Development Kit Version 9 API Specification'. The navigation bar includes links for OVERVIEW, MODULE, PACKAGE, CLASS, USE, TREE, DEPRECATED, INDEX, and HELP. A search bar is located in the top right corner. The main content area contains sections for Java SE, JDK, and JavaFX, each with a brief description. Below this, there is a table titled 'Java SE' showing the modules and their descriptions:

Module	Description
<code>java.activation</code>	Defines the JavaBeans Activation Framework (JAF) API.
<code>java.base</code>	Defines the foundational APIs of the Java SE Platform.
<code>java.compiler</code>	Defines the Language Model, Annotation Processing, and Java Compiler APIs.

Рис. 1. Пример документации для версии Java 9

**JVM** (Java Virtual Machine) – виртуальная машина Java (часть программного обеспечения Java), основной задачей которой является интерпретация байт-кода, описываемого в классах-файлах. **JRE** (Java Runtime Environment) – среда выполнения Java, предназначенная только для запуска готовых Java-приложений, а потому содержащая лишь реализацию виртуальной машины и набор стандартных библиотек. **JDK** (Java Development Kit) – реализация Java SE, средство разработчика Java, включающее в себя набор утилит, стандартные библиотеки с их сходным кодом и набор демонстрационных примеров. Утилиты включают в себя:

- `java` – реализация JVM;
- `javac` – компилятор Java;
- `appletviewer` – средство для запуска аплетов;
- `jar` – архиватор формата JAR;
- `javadoc` – утилита для автоматической генерации документации и др.

## Основы использования памяти

Управление памятью непосредственно обеспечивается JVM.

В Java все объекты программы расположены в динамической памяти (heap) и доступны по объектным ссылкам. Объектные ссылки языка Java *содержат информацию о классе* объектов, на которые они ссылаются, так что объектные ссылки – это не чистые указатели, а дескрипторы объектов. Наличие дескрипторов позволяет JVM выполнять проверку совместимости типов на фазе интерпретации кода, генерируя исключение в случае ошибки.

Память для каждого объекта выделяется при помощи этой операции `new`. Стековая память выделяется и освобождается автоматически.

*Программа не освобождает выделенную под объекты память*, это делает JVM. Автоматическое освобождение динамической памяти, занимаемой уже ненужными (неиспользуемыми) объектами, выполняется в JVM программным механизмом, который называется **сборщиком мусора (garbage collector)**.

## Жизненный цикл программ на Java

Стандартный жизненный цикл Java-приложения приведен на рис. 2. Исходный код компилируется с помощью утилиты javac.exe и выполняется виртуальной машиной (java.exe).

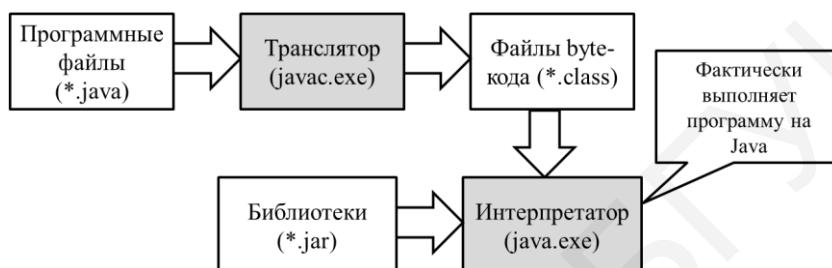


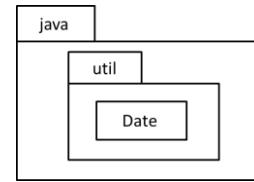
Рис. 2. Стандартный жизненный цикл Java-приложения

Для правильного понимания работы компилятора и JVM на начальном этапе изучения Java необходимо четко различать такие понятия, как ссылка на объект, собственно объект, тип ссылки на объект и тип объекта, а также стараться понимать, что такое линейный код и объектно-ориентированный код.

Пример линейного приложения	Пример объектно-ориентированного приложения
<pre>public class First {     public static void     main(String[] args){         System.out.print("Java ");         System.out.println("уже здесь!");     } }</pre> <p>Имя класса, перед которым стоит атрибут доступа public, должно совпадать с именем файла, в котором этот класс размещен</p>	<pre>public class AboutJava {     public void printReleaseData(){         System.out.println("Java уже здесь!");     } }  public class FirstOOPProgram {     public static void main(String[] args) {         AboutJava object = new AboutJava();         object.printReleaseData();     } }</pre>

## Пакеты

**Пакеты – это контейнеры классов**, которые используются для разделения пространства имен классов. Пакет в Java создается включением в текст программы первым оператором ключевого слова `package`.



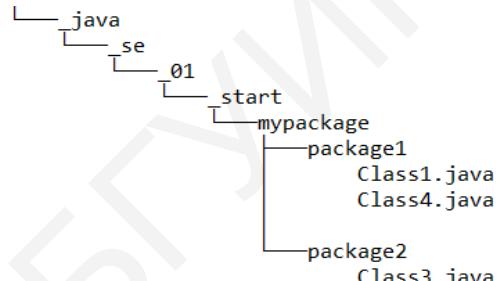
```
package имя_пакета;  
package имя_пакета.имя_подпакета.имя_подпакета;
```

Для хранения пакетов используются *каталоги файловой системы*.

```
package  
_java._se._01._start.mypackage.package1;  
package  
_java._se._01._start.mypackage.package2;
```

При компиляции поиск пакетов осуществляется:

- сначала в рабочем каталоге;
- затем используется параметр переменной среды `classpath`;
- после этого используется указанное местонахождение пакета в параметре компилятора `-classpath`.



Пакеты **регулируют права доступа** к классам и подклассам. Если ни один модификатор доступа не указан, то сущность (т. е. класс, метод или переменная) является доступной всем методам в том же самом *пакете*.

## Импортирование

Для подключения пакета используется ключевое слово `import`:

```
import имя_пакета.имя_подпакета.*;  
import имя_пакета.имя_подпакета.имя_подпакета.имя_класса;
```

Например,

```
package _java._se._01._start.mypackage.package1;  
public class Class1 { Class2 obj = new Class2(); int varInteger; }  
class Class2{ }
```

```
package _java._se._01._start.mypackage.package2;  
import _java._se._01._start.mypackage.package1.Class1;
```

```

public class Class3 {
    public static void main(String[] args) { Class1 cl1 = new Class1(); }
}

```

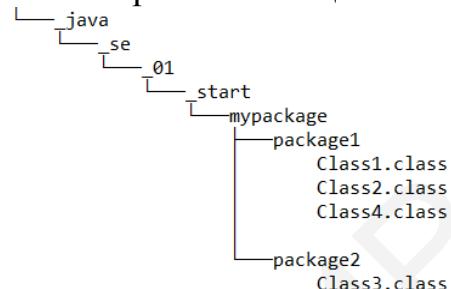
```

package
java._se._01._start.mypackage.package1;
public class Class4 {
    Class2 obj = new Class2();

    void methodClass4(Class1 cl1){
        cl1.varInteger = 4;
    }
}

```

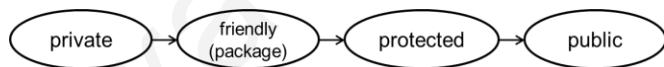
Структура и содержимое пакетов при компиляции:



В данном примере символ «\_» в начале имени пакета применен для улучшения чтения, собственные рабочие пакеты так называть не рекомендуется, как не рекомендуется давать пакетам имена java, se и прочие, которые могут при разборе кода ввести в заблуждение.

## Модификаторы доступа

Для описания прав доступа к классам и элементам классов Java работает с четырьмя атрибутами доступа:



- **private** – закрытый, доступен внутри класса, в котором описан;
- **friendly (package, private-package)** – закрытый внутри пакета, доступен внутри класса и внутри пакета, в котором описан;
- **protected** – защищенный, доступен внутри класса, внутри пакета и в производном классе в другом пакете;
- **public** – открытый, доступен внутри класса, внутри пакета, в производном классе в другом пакете и вообще везде.

## Компиляция и запуск приложения из командной строки

Для компиляции и запуска Java-приложения без средств IDE выполните следующие шаги, описанные далее.

Разместите код в каталоге <workdir>/src:

```

package start;
public class Console {
    public static void main(String[] args) {
        System.out.println("Hello!");
    }
}

```

Скомпилируйте программу командой  
**javac -d bin src\Console.java**

После успешной компиляции создастся файл **Console.class**. Если такой файл не создался, то, значит, код содержит ошибки, которые необходимо устраниить и еще раз скомпилировать программу.

Запуск программы из консоли осуществляется командой  
**java -cp ./bin start.Console**

## Работа с аргументами командной строки

Для передачи параметров командной строки и работы с переменной args метода main используйте следующий синтаксис:

```
package start;
public class CommandArg {
    public static void main(String[] args) {
        for(int i=0; i<args.length; i++){
            System.out.println("Аргумент " + i + " = " + args[i]);
        }
    }
}
```

Скомпилировать приложение и запустить его можно с помощью следующей командной строки:

**java start.CommandArg first second 23 56.2 23,9**

## Примитивные типы данных

Простые типы делятся на четыре группы:

- *целые: int, byte, short, long;*
- *числа с плавающей точкой: float, double;*
- *символы: char;*
- *логические: boolean.*

## Особенности примитивных типов

- Размер примитивных типов одинаков для всех платформ; за счет этого становится возможной переносимость кода.
- Размер boolean неопределен. Указано, что он может принимать значения true или false.

Преобразования между типом boolean и другими типами не существуют.

## Сравнение примитивных типов

Примитивный тип	Размер (бит)	Минимальное значение	Максимальное значение	Класс-оболочка
boolean	-	-	-	Boolean
char	16	Unicode 0	U2 <sup>16</sup> -1	Character
byte	8	-128	127	Byte
short	16	-2 <sup>15</sup>	2 <sup>15</sup> -1	Short
int	32	-2 <sup>31</sup>	2 <sup>31</sup> -1	Integer
long	64	-2 <sup>63</sup>	2 <sup>63</sup> -1	Long
float	32	IEEE754	IEEE754	Float
double	64	IEEE754	IEEE754	Double
void	-	-	-	Void

## Значения по умолчанию

При создании объектов переменные экземпляра класса, а при загрузке класса переменные класса принимают значения по умолчанию.

Примитивный тип	Значение по умолчанию
boolean	false
char	'\u0000' (null)
byte	(byte)0
short	(short)0
int	0
long	0L
float	0.0f
double	0.0d

Основная форма объявления переменных:

При объявлении переменные могут быть проинициализированы

тип идентификатор [= значение];

В именах переменных не могут использоваться символы *арифметических* и *логических* операторов, а также символ «#». Применение символов «\$» и «\_» допустимо, в том числе и в первой позиции имени.

<pre>public class VariablesExample {     boolean statusOn;     double javaVar = 2.34;      public static void     main(String[] args) {         int itemsSold = 04;         double salary = 1.234e3;         float itemCost = 11.0f;     } }</pre>	<pre>int i = 0xFd45, k\$; double _interestRate; }  public void javaMethod() {     long simpleVar =     1_000_000_000_000L;     byte byteVar2 = 123; } }</pre>
--	---

## Ключевые и зарезервированные слова языка Java

<b>abstract</b>	<b>continue</b>	<b>for</b>	<b>new</b>	<b>switch</b>
<b>assert</b>	<b>default</b>	<b>goto</b>	<b>package</b>	<b>synchronized</b>
<b>boolean</b>	<b>do</b>	<b>if</b>	<b>private</b>	<b>this</b>
<b>break</b>	<b>double</b>	<b>implements</b>	<b>protected</b>	<b>throw</b>
<b>byte</b>	<b>else</b>	<b>import</b>	<b>public</b>	<b>throws</b>
<b>case</b>	<b>enum</b>	<b>instanceof</b>	<b>return</b>	<b>transient</b>
<b>catch</b>	<b>extends</b>	<b>int</b>	<b>short</b>	<b>try</b>
<b>char</b>	<b>final</b>	<b>interface</b>	<b>static</b>	<b>void</b>
<b>class</b>	<b>finally</b>	<b>long</b>	<b>strictfp</b>	<b>volatile</b>
<b>const</b>	<b>float</b>	<b>native</b>	<b>super</b>	<b>while</b>

Кроме ключевых слов, в Java существуют три литерала: **null**, **true**, **false**, не относящиеся к ключевым и зарезервированным словам, а также дополнительные зарезервированные слова: **const**, **goto**.

## Литералы

Литералы (константные значения в коде) примитивных типов в Java подразделяются на те же категории, что и примитивные типы: целочисленные, с плавающей точкой, булевы, символьные. Кроме них существуют еще и строковые литералы, являющиеся объектами (рис. 3).

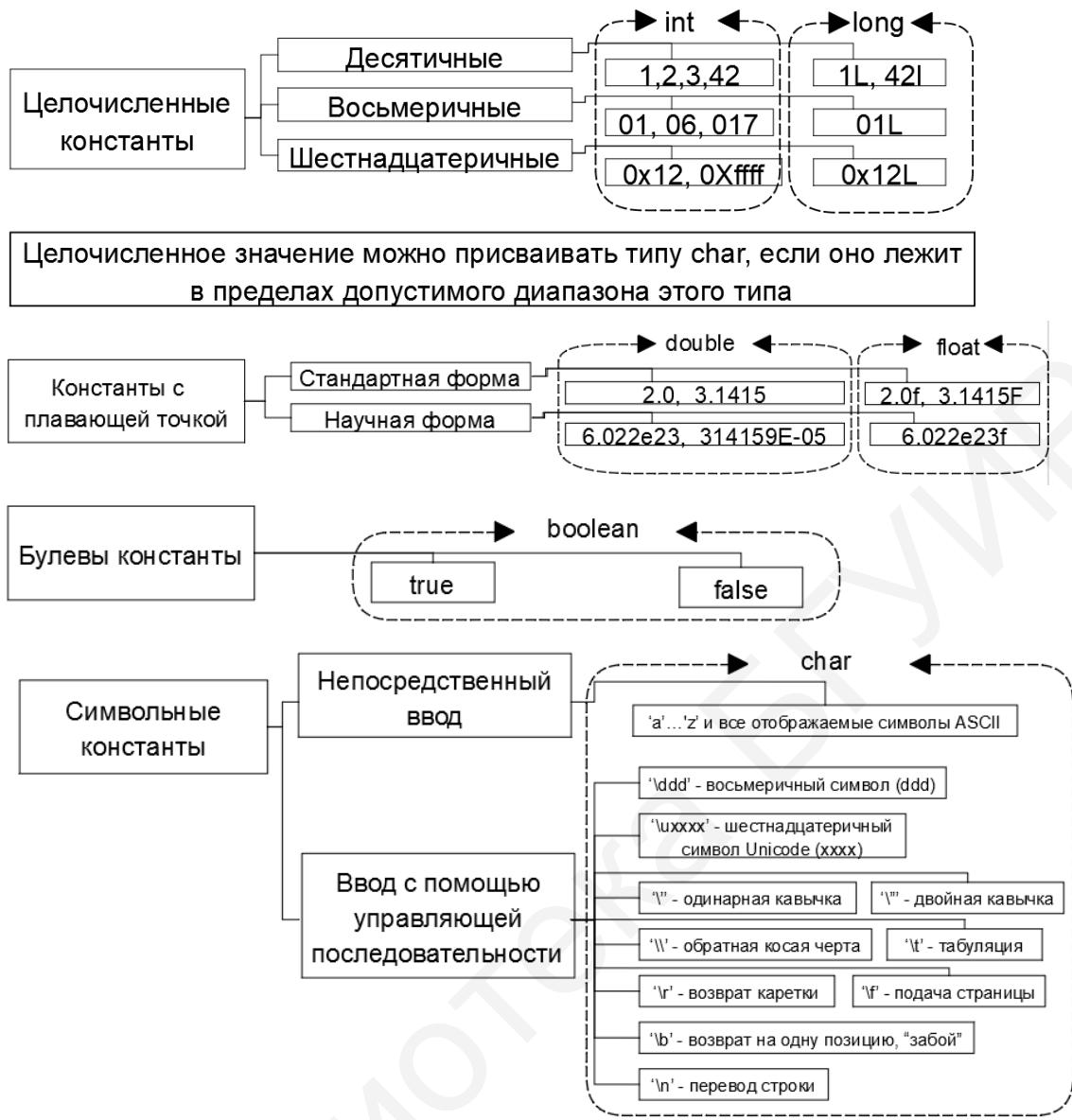


Рис. 3. Литералы

## Особенности работы с переменными

Java не позволяет присваивать переменной значение более длинного типа.

```
int intVar = 1_000_000_000L; //Error
```

Исключение составляют операторы инкремента, декремента и операторы `+=`, `-=`, `*=`, `/=`.

```
var @= expr == var = (typename)(var @ (expr))
```

```
int intVar = 100; long longVar = 1000000000000L;
intVar += longVar;
intVar = intVar + longVar; // Error
```

## Преобразование примитивных типов

Java запрещает смешивать в выражениях величины разных типов, однако при числовых операциях такое часто бывает необходимо.

Различают:

- **повышающее** (разрешенное, неявное) преобразование;
- **поникающее** (явное) приведение типа.

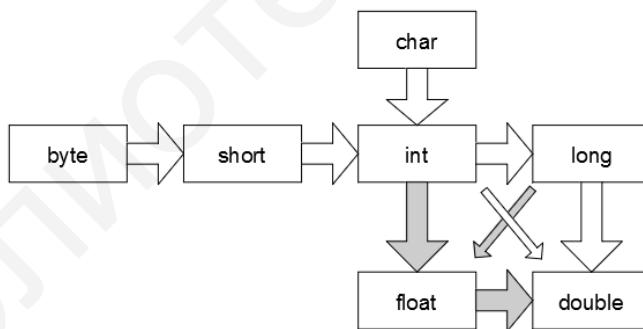
**Расширяющее (повышающее) преобразование.** Результирующий тип имеет больший диапазон значений, чем исходный тип:

```
int x = 200;  
long y = (long)x;  
long z = x;  
long value1 = (long)200; //необязательно, т. к. компилятор делает это автоматически
```

**Сужающее (поникающее) преобразование.** Результирующий тип имеет меньший диапазон значений, чем исходный тип.

```
long value2 = 1000L;  
int value3 = (int)value2; //обязательно
```

Повышающее преобразование осуществляется автоматически, даже в случае потери данных.



Серыми стрелками обозначены преобразования, при которых может произойти потеря точности.

```
public class LoseAccuracy01 {  
    public static void main(String[] args) {  
        int x1 = 123456789;      int x2 = 99999999;  
        float f1 = x1;           float f2 = x2;  
        System.out.println("f1 - "+f1);  
        System.out.println("f2 - "+f2);  
    } }
```

```

public class LoseAccuracy02 {
    public static void main(String[] args) {
        float f1 = 1.2345f;
        double d1 = f1;
        double d2 = 1.2345;

        System.out.println("f1 - " + f1);
        System.out.println("d1 - " + d1);
        System.out.println("-----");

        System.out.printf("f1 = %.16f\n", f1);
        System.out.printf("d2 = %.16f\n", d2);
    }
}

```

```

public class LoseAccuracy03 {
    public static void main(String[] args) {
        long l1 = 1234567891234L;
        float f1 = l1;

        System.out.println("l1 - " + l1);
        System.out.println("f1 - " + f1);
    }
}

```

## Приведение типов в выражении

При вычислении выражения (**a** @ **b**) аргументы **a** и **b** преобразуются в числа, имеющие одинаковый тип:

- если одно из чисел **double**, то в **double**;
- иначе, если одно из чисел **float**, то в **float**;
- иначе, если одно из чисел **long**, то в **long**;
- иначе оба числа преобразуются в **int**.

Арифметическое выражение над **byte**, **short** или **char** имеет тип **int**, поэтому для присвоения результата обратно в **byte**, **short** или **char** понадобится явное приведение типа.

```

public static void main(String[] args)
{
    long a = 10_000_000_000L;
    int x;

    x = (int) a;
    System.out.println("x - " + x);
    byte b5 = 50;
    // byte b4 = b5*2; // error
}

```

```

byte b4 = (byte) (b5 * 2);
byte b1 = 50, b2 = 20, b3 = 127;
int x2 = b1 * b2 * b3;
System.out.println("x2 - " + x2);

double d = 12.34;
int x3;
x3 = (int) d;
System.out.println("x3 - " + x3);
}

```

## Особенности приведения вещественных чисел

- Слишком большое дробное число при приведении к целому превращается в **Integer.MAX\_VALUE** или **Integer.MIN\_VALUE**
- Слишком большой *double* при приведении к *float* превращается в **Float.POSITIVE\_INFINITY** или **Float.NEGATIVE\_INFINITY**

```
public static void main(String[] args) {  
    double d = 1000000e100;  
    int x = (int) d; int y = (int) (-d);  
    System.out.println("x = " + x);  
    System.out.println("Integer.MAX_VALUE = " + Integer.MAX_VALUE);  
    System.out.println("y = " + y);  
    System.out.println("Integer.MIN_VALUE = " + Integer.MIN_VALUE);  
  
    float z = (float) d;  
    float k = (float) (-d);  
  
    System.out.println("z = " + z); System.out.println("k = " + k);  
}
```

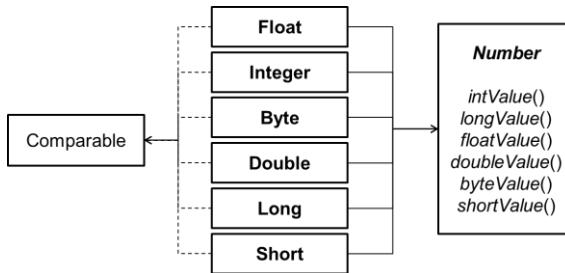
## Классы-оболочки

Кроме базовых типов данных широко используются соответствующие классы (wrapper-классы) **Boolean**, **Character**, **Integer**, **Byte**, **Short**, **Long**, **Float**, **Double**:

- Объекты этих классов могут хранить те же значения, что и соответствующие им базовые типы.
- Объекты этих классов представляют ссылки на участки динамической памяти, в которой хранятся их значения, и являются классами-оболочками для значений базовых типов.
- Объекты этих классов являются константными.

Классы-оболочки (кроме **Boolean** и **Character**) являются наследниками абстрактного класса **Number** и реализуют интерфейс **Comparable**, представляющий собой интерфейс для работы со всеми скалярными типами.

Класс **Character** не наследуется от **Number**, ему нет необходимости поддерживать интерфейс классов, предназначенных для хранения результатов арифметических операций. Класс **Character** имеет целый ряд специфических методов для обработки символьной информации. У этого класса, в отличие от других классов оболочек, не существует конструктора с параметром типа **String**.



```

public class IntegerType {
    public static void main(String[] args) {
        Integer i = new Integer(10);      System.out.println("i1=" + i);
        changeInteger(i);   System.out.println("i2=" + i);
    }
    public static void changeInteger(Integer x) {
        System.out.println("x1=" + x);  x = new Integer(20);
        System.out.println("x2=" + x);
    }
}

public class CharacterType {
    public static void main(String[] args) {
        char c = '9';          Character ch = new Character(c);
        System.out.println("charValue() - " + ch.charValue());
        System.out.println("isJavaIdentifierStart? - "
            + Character.isJavaIdentifierStart(c));
        System.out.println("isLetter? - " + Character.isLetter(c));
        System.out.println("digit for 12 - " + Character.forDigit(14, 16));
    }
}

```

## Big-классы

Java включает два класса для работы с высокоточной арифметикой: **BigInteger** и **BigDecimal**, которые поддерживают целые числа и числа с фиксированной точкой произвольной точности.

```

import java.math.BigInteger;
...
BigInteger numI1, numI2, bigNumI;
numI1 = BigInteger.valueOf(1_000_000_000_000L);
numI2 = numI1.multiply(numI1); System.out.println(numI2);
numI2 = numI1.multiply(numI1).multiply(numI1); System.out.println(numI2);
numI2 = numI1.multiply(numI1).multiply(numI1).multiply(numI1);
System.out.println(numI2);
numI2 = numI1.multiply(numI1).multiply(numI1)
        .multiply(numI1).multiply(numI1);
System.out.println(numI2);

```

```
numI2 = numI1.multiply(numI1).multiply(numI1).multiply(numI1)
        .multiply(numI1).multiply(numI1);
System.out.println(numI2);
```

## Автоупаковка и автораспаковка

В версии 5.0 введен процесс автоматической инкапсуляции данных базовых типов в соответствующие объекты оболочки и обратно (**автоупаковка**). При этом нет необходимости в создании соответствующего объекта с использованием оператора **new**.

```
Integer iob = 71;
```

**Автораспаковка** – процесс извлечения из объекта-оболочки значения базового типа. Вызовы таких методов, как **intValue()**, **doubleValue()** становятся излишними. Допускается участие объектов в арифметических операциях, однако не следует использовать это очень часто, поскольку упаковка/распаковка является ресурсоемким процессом.

```
Integer j = 71; // создание объекта+упаковка
Integer k = ++j; // распаковка+операция+упаковка
int i = 2;
k = i + j + k;
System.out.println(k);
```

В классах **Long**, **Integer**, **Short** и **Byte** присутствует внутренний кеш ссылок на значения от -128 до 127.

```
Integer i1 = 10; Integer i2 = 10; System.out.println(i1 == i2);
i1 = 128; i2 = 128; System.out.println(i1 == i2);
```

При инициализации объекта класса-оболочки значением базового типа преобразование типов необходимо указывать явно. Возможно создавать объекты и массивы, сохраняющие различные базовые типы без взаимных преобразований, с помощью ссылки на класс **Number**. При автоупаковке значения базового типа возможны ситуации с появлением некорректных значений и непроверяемых ошибок.

```
Number n1 = 1; Number n2 = 7.1; Number array[] = {71, 7.1, 7L};
Integer i1 = (Integer)n1;
Integer i2 = (Integer)n2; // runtime error
Integer[] i3 = (Integer[])array; // runtime error
```

# Операторы

## Арифметические операторы

+	Сложение	/	Деление
+=	Сложение (с присваиванием)	/=	Деление (с присваиванием)
-	Бинарное вычитание и унарное изменение знака	%	Деление по модулю
-=	Вычитание (с присваиванием)	%=	Деление по модулю (с присваиванием)
*	Умножение	++	Инкремент
*=	Умножение (с присваиванием)	--	Декремент

## Операторы отношения

Применяются для сравнения символов, целых и вещественных чисел, а также для сравнения ссылок при работе с объектами.

<	Меньше	>	Больше
<=	Меньше либо равно	>=	Больше либо равно
==	Равно	!=	Не равно

## Битовые операторы

=	ИЛИ (с присваиванием)	>>=	Сдвиг вправо (с присваиванием)
&	И	>>>	Сдвиг вправо с появлением нулей
&=	И (с присваиванием)	>>>=	Сдвиг вправо с появлением нулей и присваиванием
^	Исключающее ИЛИ	<<	Сдвиг влево
^=	Исключающее ИЛИ (с присваиванием)	<<=	Сдвиг влево с присваиванием
~	Унарное отрицание		

>> – арифметический сдвиг

>>> – логический сдвиг

## Логические операторы

,  ,  =	ИЛИ	&&, &, &=	И
!	Унарное отрицание	^, ^=	Исключающее ИЛИ

&& и || – вычисление по сокращенной схеме

& и | – вычисление по полной схеме

## Дополнительные операторы Java

К операторам относится также оператор определения принадлежности типу **instanceof**, оператор [ ] и тернарный оператор ?: (if-then-else). Логические операции выполняются над значениями типа **boolean** (**true** или **false**). Оператор **instanceof** возвращает значение **true**, если объект является экземпляром данного класса.

### Приоритет операций

```
int i=3;  
i = -i++ + i++ + -i;  
System.out.println(i);
```

Приоритет	Операция	Порядок выполнения операций в выражении при одном приоритете
1	[ ] . () (вызов метода)	Слева направо
2	! ~ ++ -- +(унарный) -(унарный) () (приведение) new	Справа налево
3	* / %	Слева направо
4	+ - , +(конкатенация строк)	Слева направо
5	<< >> >>>	Слева направо
6	< <= > >= instanceof	Слева направо
7	== !=	Слева направо
8	&(битовое), &(логическое)	Слева направо
9	^(битовое), ^(логическое)	Слева направо
10	(битовое),   (логическое)	Слева направо
11	&&	Слева направо
12		Слева направо
13	?:	Слева направо
14	= += -= *= /= %=  = ^= <<= >>= >>>=	Справа налево

### Операции над целыми числами

Операции над целыми числами: +, -, \*, %, /, ++,-- и битовые опера-

ции **&**, **|**, **^**, **~** аналогичны операциям большинства языков программирования. Деление на нуль целочисленного типа вызывает исключительную ситуацию, переполнение не контролируется (исключение не выбрасывается).

## IEEE 754

Операции над числами с плавающей точкой практически те же, что и в других языках. Все вычисления, которые проводятся над числами с плавающей точкой, следуют стандарту **IEEE 754**. По стандарту **IEEE 754** введены понятие бесконечности **+Infinity** и **-Infinity** и значение **NaN** (Not a Number).

- Результат деления положительного числа на нуль равен положительной бесконечности, отрицательного – отрицательной бесконечности.
- Вычисление квадратного корня из отрицательного числа или деление 0/0 – не число.
- Переполнение дает **+Infinity** или **-Infinity** в зависимости от направления.
- Любая арифметическая операция с **NaN** дает **NaN**.
- **NaN**  $\neq$  **NaN**.

Java вводит следующие значения в классах **Double** и **Integer**:

<b>Double.MAX_VALUE;</b> <b>Double.MIN_VALUE;</b> <b>Double.POSITIVE_INFINITY;</b> <b>Double.NEGATIVE_INFINITY;</b> <b>Double.NaN;</b> <b>Double.isNaN()</b> .	<b>Float.MAX_VALUE;</b> <b>Float.MIN_VALUE;</b> <b>Float.POSITIVE_INFINITY;</b> <b>Float.NEGATIVE_INFINITY;</b> <b>Float.NaN;</b> <b>Float.isNaN()</b>
---	---

```
double i = 7.0;  double k;  
System.out.println(i / 0);      System.out.println(-i / 0);  
System.out.println(k=Math.sqrt(-i));  System.out.println(Double.isNaN(k));
```

## strictfp

- Java использует FPU (*Floating-Point Unit*, модуль операций с плавающей запятой) для вычислений с плавающей точкой.
- Регистры FPU могут быть шире 64 бит.
- Результаты вычислений могут отличаться.
- Модификатор **strictfp** включает режим строгой совместимости, результаты будут идентичны на любом процессоре.

```
public strictfp class Main {  
    public strictfp void method() { }  
}
```

## Math&StrictMath

Для организации математических вычислений в Java существует класс **Math** и **StrictMath**.

- `java.lang.StrictMath` – класс-утилита, содержащий основные математические функции. Гарантирует точную повторяемость числовых результатов вплоть до бита на разных аппаратных платформах.
- `java.lang.Math` – класс-утилита, работающий быстрее чем StrictMath (на старых версиях машины), но не гарантирующий точное воспроизведение числовых результатов.
- В версии 1.6 `java.lang.Math` делегирует вызовы `StrictMath`.

## Статический импорт

Ключевое слово **import** с последующим ключевым словом **static** используется для импорта статических полей и методов классов, в результате чего отпадает необходимость в использовании имен классов перед ними.

```
import static java.lang.Math.pow;
import static java.lang.Math.PI;
public class StaticImport {
    private int i = 20;
    public void staticImport() {
        double x;
        x = pow(i, 2)*PI;
        System.out.println("x=" + x);
    }
}
```

## Блоки кода

Блоки кода обрамляются в фигурные скобки “{“ “}” и охватывают: определение класса; определения методов; логически связанные разделы кода.

```
public class CodeBlock {
    public Date getToday() {           //...
    }
    public static void main(String[] args) {
        CodeBlock object = CodeBlock.getInstance();
        if(object != null){
            //..
        }
    }
}
```

## Оператор if

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

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

## Циклы

Циклы выполняются, пока булево выражение *boolexp* равно **true**. Оператор прерывания цикла **break** и оператор прерывания итерации цикла **continue** можно использовать с меткой для обеспечения выхода из вложенных циклов.

- 1) **while** (boolexp) { /\*операторы\*/ }
- 2) **do** { /\*операторы\*/ }  
    **while** (boolexp);
- 3) **for**(exp1; boolexp; exp3){ /\*операторы\*/ }
- 4) **for**((Тип exp1 : exp2){ /\*операторы\*/ }

## Операторы безусловного перехода

**break** – применяется для выхода из цикла оператора **switch**

**continue** – применяется для перехода к следующей итерации цикла

В языке Java расширились возможности оператора прерывания цикла **break** и оператора прерывания итерации цикла **continue**, которые можно использовать с меткой.

Outer:

```
for(int i=0; i < args.length ; i++) {  
    // ...          break Outer;      // ...  
}
```

## Использование циклов

Проверка условия для всех циклов выполняется только один раз за одну итерацию: для циклов **for** и **while** – перед итерацией, для цикла **do/while** – по окончании итерации.

Цикл **for** следует использовать при необходимости выполнения алгоритма строго определенное количество раз. Цикл **while** используется в случае, когда неизвестно число итераций для достижения необходимого результата, например, поиск необходимого значения в массиве или кол-

лекции. Этот цикл применяется для организации бесконечных циклов в виде **while(true)**.

Для цикла **for** не рекомендуется изменять индекс цикла. Условие завершения цикла должно быть очевидным, чтобы цикл не «сорвался» в бесконечный цикл. Для индексов следует применять осмысленные имена. Циклы не должны быть слишком длинными. Такой цикл претендует на выделение в отдельный метод. Вложенность циклов не должна превышать трех.

## Оператор switch

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

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

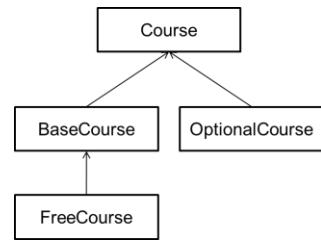
```
public class SwitchWithBreak {  
    public static void main(String[] args) {  
        String s = new String("one");  
        switch (s) {  
            case "two":  
                System.out.println("two");  
                break;  
            case "three":  
                System.out.println("three");  
                break;  
            case "four":  
                System.out.println("four");  
                break;  
            case "one":  
                System.out.println("one");  
                break;  
            default:  
                System.out.println("default");  
        }  
    }  
}
```

```
public class SwitchWithoutBreak {  
    public static void main(String[] args) {  
        int x = 10;  
        switch (x) {  
            case 20:  
                System.out.println("20");  
            case 30:  
                System.out.println("30");  
            default:  
                System.out.println("default");  
            case 10:  
                System.out.println("10");  
            case 40:  
                System.out.println("40");  
        }  
    }  
}
```

## Оператор instanceof

Оператор **instanceof** возвращает значение true, если объект является экземпляром данного типа. Например, для иерархии наследования:

```
class Course extends Object {}  
class BaseCourse extends Course {}  
class FreeCourse extends BaseCourse {}  
class OptionalCourse extends Course {}
```



Объект подкласса может быть использован всюду, где используется объект суперкласса.

Результатом действия оператора **instanceof** будет истина, если объект является объектом типа, с которым идет проверка или одного из его подклассов, но не наоборот.

```
public class InstanceofOp {  
    public static void main(String[] args) {  
        doLogic(new BaseCourse());  
        doLogic(new OptionalCourse());  
        doLogic(new FreeCourse());  
    }  
  
    public static void doLogic(Course c) {  
        if (c instanceof BaseCourse) {  
            System.out.println("BaseCourse");  
        } else if (c instanceof OptionalCourse) {  
            System.out.println("OptionalCourse");  
        } else {  
            System.out.println("Что-то другое.");  
        }  
    }  
}
```

## Java Beans (основы)

**Java Beans** – гибкая, мощная и удобная технология разработки многократно используемых программных компонент, называемых *beans*.

С точки зрения объектно-ориентированного программирования (ООП), компонента *Java Bean* – это классический самодостаточный *объект*, который, будучи написан один раз, может быть многократно использован при построении новых аплетов, сервлетов, полноценных приложений, а также других компонент *Java Bean*.

Отличие от других технологий заключается в том, что компонента Java Bean строится по определенным правилам, с использованием в некоторых ситуациях строго регламентированных интерфейсов и базовых классов.

Java Bean – многократно используемая компонента, состоящая из **свойств (properties), методов (methods) и событий (events)**

**Свойства компонент Bean** – это дискретные, именованные атрибуты соответствующего объекта, которые могут оказывать влияние на режим его функционирования.

В отличие от атрибутов обычного класса свойства компоненты Bean должны задаваться вполне определенным образом: *нежелательно объявлять* какой-либо атрибут компоненты Bean как *public*. Наоборот, его следует декларировать как *private*, а сам класс дополнить двумя методами **set** и **get**.

```
import java.awt.Color;
public class BeanExample {
    private Color color;

    public void setColor(Color newColor) {      color = newColor; }
    public Color getColor() {                  return color; }
}
```

Согласно спецификации Bean методы *set* и *get* необходимо использовать не только для атрибутов простого типа, таких как int или String, но и в *более сложных ситуациях*, например для *внутренних массивов* String[].

```
public class BeanArrayExample {
    private double data[];

    public double getData(int index) {          return data[index]; }
    public void setData(int index, double value) { data[index] = value; }
    public double[] getData() {                  return data; }

    public void setData(double[] values) {
        data = new double[values.length];
        System.arraycopy(values, 0, data, 0, values.length);
    }
}
```

Атрибуту типа **boolean** в классе Bean должны соответствовать методы **is** и **set**.

```
public class BeanBoolExample {
```

```
private boolean ready;

public void setReady(boolean newStatus) {      ready = newStatus; }
public boolean isReady() {                      return ready;    }
}
```

Формально к свойствам компоненты Bean следует отнести также инициируемые ею события. Каждому из этих событий в компоненте Bean также должно соответствовать два метода – add и remove.

Создаваемый компонент Bean зачастую функционирует в программной среде *со многими параллельными потоками (threads)*, т. е. в условиях, когда сразу от нескольких потоков могут поступить запросы на доступ к тем или иным методам или атрибутам объекта. Доступ к таким объектам следует синхронизировать.

## Массивы

Для хранения нескольких однотипных значений используется ссылочный тип – массив. Массивы элементов базовых типов состоят из значений, проиндексированных начиная с нуля. Все массивы в языке Java являются динамическими, поэтому для создания массива требуется выделение памяти с помощью оператора **new** или инициализации:

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

Значения элементов неинициализированных массивов, для которых выделена память, устанавливаются в нуль. Многомерных массивов в Java не существует, но можно объявлять массивы массивов. Для задания начальных значений массивов существует специальная форма инициализатора:

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

Массивы объектов в действительности представляют собой массивы ссылок, проинициализированных по умолчанию значением **null**. Все массивы хранятся в куче (**heap**) – одной из подобластей памяти, выделенной системой для работы виртуальной машины. Определить общий объем памяти и объем свободной памяти можно с помощью методов **totalMemory()** и **freeMemory()** класса **Runtime**.

## Одномерные массивы

Имена массивов являются ссылками. Для объявления ссылки на массив можно записать пустые квадратные скобки после имени типа, например: `int a[]`. Аналогичный результат получится при записи `int[] a`.

```

int myArray[];
int mySecond[] = new int[100];
int a[] = { 5, 10, 0, -5, 16, -2 };
myArray = a;

public class CreateArray {
    public static void main(String[] args) {
        int[] price = new int[10];
        int[] rooms = new int[] { 1, 2, 3 };
        Item[] items = new Item[10];
        Item[] undefinedItems = new Item[] { new Item(1), new Item(2), new Item(3) };
    }
}

class Item {
    public Item(int i) {}
}

```

## Работа с массивами

Обращение к ячейке массива в Java аналогично большинству других языков.

```

public class FindMax {
    public static void main(String[] args) {
        int a[] = { 5, 10, 0, -5, 16, -2 };
        int max = a[0];
        for (int i = 0; i < a.length; i++){if (max < a[i]) max = a[i];      }
        System.out.println("Max="+max);
        for(int x : a){      System.out.print(x+" ");}
        System.out.println();
    }
}

```

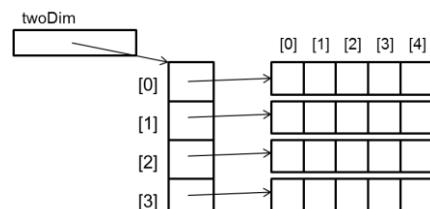
## Массив массивов

Двумерных массивов в Java нет. Есть только массивы массивов:

```
int twoDim [][] = new int[4][5];
```

Каждый из массивов может иметь отличную от других длину:

```
int twoDim [][] = new int[4][];
```



```
twoDim[0] = new int [10];
twoDim[1] = new int [20];
twoDim[2] = new int [30];
twoDim[3] = new int [100];
```

Первый индекс указывает на порядковый номер массива, например **arr[2][0]** указывает на первый элемент третьего массива, а именно на значение **4**.

```
int arr[][] = {
    { 1 },
    { 2, 3 },
    { 4, 5, 6 },
    { 7, 8, 9, 0 }
};
```

## Работа с массивами массивов

Члены объектов-массивов:

- **public final int length** – это поле содержит длину массива;
- **public Object clone()** – создает копию массива;
- + все методы класса Object.

```
public class CloneArray {
    public static void main(String[] args) {
        int ia[][] = { { 1, 2 }, null };
        int ja[][] = (int[][]) ia.clone();
        System.out.print((ia == ja) + " ");
        System.out.println(ia[0] == ja[0] && ia[1] == ja[1]);
    }
}
```

## Приведение типов в массивах

Любой массив можно привести к классу Object или к массиву совместимого типа:

```
public class ConvertArray {
    public static void main(String[] args) {
        ColoredPoint[] cpa = new ColoredPoint[10];
        Point[] pa = cpa;
        System.out.println(pa[1] == null);
        try { pa[0] = new Point();
        } catch (ArrayStoreException e) { System.out.println(e);
    }
}
```

```
        }
    }
}

class Point {int x, y;}
class ColoredPoint extends Point { int color;}
```

## Ошибки времени выполнения при работе с массивами

Обращение к несуществующему индексу массива отслеживается виртуальной машиной во время исполнения кода:

```
public class ArrayIndexError {
    public static void main(String[] args) {
        int array[] = new int[] { 1, 2, 3 };
        System.out.println(array[3]);
    }
}
```

Попытка поместить в массив неподходящий элемент пресекается виртуальной машиной:

```
public class ArrayTypegetError {
    public static void main(String[] args) {
        Object x[] = new String[3];
        // попытка поместить в массив содержимое
        // несоответствующего типа
        x[0] = new Integer(0);
    }
}
```

## Code conventions for Java Programming

80 % стоимости программного обеспечения уходит на поддержку. Программное обеспечение не может весь свой жизненный цикл поддерживаться автором. Code conventions улучшает удобочитаемость программного кода, позволяя понять новый код более быстро и полностью.

<http://www.oracle.com/technetwork/java/codeconvtoc-136057.html>

Рекомендации по написанию кода.

*Объявляйте локальные переменные сразу перед использованием*

- Определяется их область видимости.
  - Уменьшается вероятность ошибок и неудобочитаемости.
- Поля объявляйте как private*
- Декларирование полей как public в большинстве случаев некорректно, оно не защищает пользователя класса от изменений в реализации класса.

- Объявляйте поля как private. Если пользователю необходимо получить доступ к этим полям, следует предусмотреть set- и get- методы.

*При объявлении разделяйте public и private-члены класса*

- Это общераспространенная практика разделения членов класса согласно их области видимости (public, private, protected). Данные с каким атрибутом доступа будут располагаться первыми, зависит от программиста.

*Используйте Javadoc*

- Javadoc – это мощный инструмент, который необходимо использовать.

*С осторожностью используйте System.Exit(0) с многопоточными приложениями.*

- Нормальный способ завершения программы должен завершать работу всех используемых потоков.

*Проверяйте аргументы методов*

- Первые строки методов обычно проверяют корректность переданных параметров. Идея состоит в том, чтобы как можно быстрее сгенерировать сообщение об ошибке в случае неудачи. Это особенно важно для конструкторов.

*Применяйте дополнительные пробелы в списке аргументов*

- Дополнительные пробелы в списке аргументов повышают читабельность кода – как (this ) вместо (that).

*Применяйте Testing Framework*

- Используйте Testing Framework, чтобы убедиться, что класс выполняет контракт.

*Используйте массивы нулевой длины вместо null*

- Когда метод возвращает массив, который может быть пустым, не следует возвращать null. Это позволяет не проверять возвращаемое значение на null.

*Избегайте пустых блоков catch*

- В этом случае, когда происходит исключение, ничего не происходит, и программа завершает свою работу по непонятной причине.

*Применяйте оператор throws*

- Не следует использовать базовый класс исключения вместо нескольких его производных, в этом случае теряется важная информация об исключении.

*Правильно выбирайте используемые коллекции*

- Документация Sun определяет ArrayList, HashMap и HashSet как предпочтительные для применения. Их производительность выше.

*Работайте с коллекциями без использование индексов*

- Применяете for-each или итераторы. Индексы всегда остаются одной из главных причин ошибок.

*Структура source-файла*

- public-класс или интерфейс всегда должен быть объявлен первым в файле.
- если есть ассоциированные с public-классом private-классы или интерфейсы, их можно разместить в одном файле.

### *Declarations. Длина строк кода*

- Не используйте строки длиной более 80 символов.

### *Объявление переменных*

- Не присваивайте одинаковые значения нескольким переменным одним оператором.

fooBar.fChar = barFoo.lchar = 'c'; // AVOID!!!

*При декларировании переменных объявляйте по одной переменной в строке кода*

- Такое объявление позволяет писать понятные комментарии.

*Statements. Каждая строка кода должна содержать только один оператор:*

- Example:

argv++; // Correct

argc--; // Correct

argv++; argc--; // AVOID!

Имена файлов:

- Customer.java

- Person.class

Имена пакетов:

- java.util

- javax.swing

Имена классов:

- Customer

- Person

Имена свойств класса:

- firstName

- id

Имена методов:

- getName

- isAlive

Имена констант:

- SQUARE\_SIZE

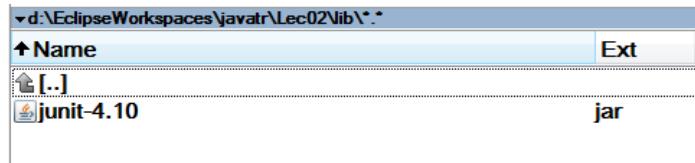
Также могут использоваться цифры от 1 до 9 и символы «\_», «\$».

## **Основы тестирования с помощью Junit**

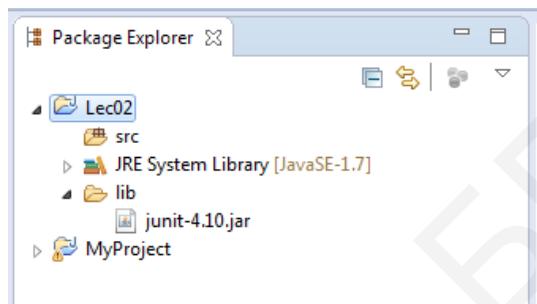
Модульное тестирование (unit testing) – процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы путем запуска тестов в искусственной среде. Модуль – наименьший компонент, который можно скомпилировать, либо наименьший компонент, функциональность которого можно проверить. Оценивая каждый модуль изолированно и подтверждая корректность его работы, точно установить проблему значительно проще, чем если бы элемент был частью системы. С помощью модульного тестирования обычно тестируют низкоуровневые элементы кода, такие как методы и интерфейсы.

Для ознакомления с процедурой тестирования изучите последовательность действий, приведенную далее:

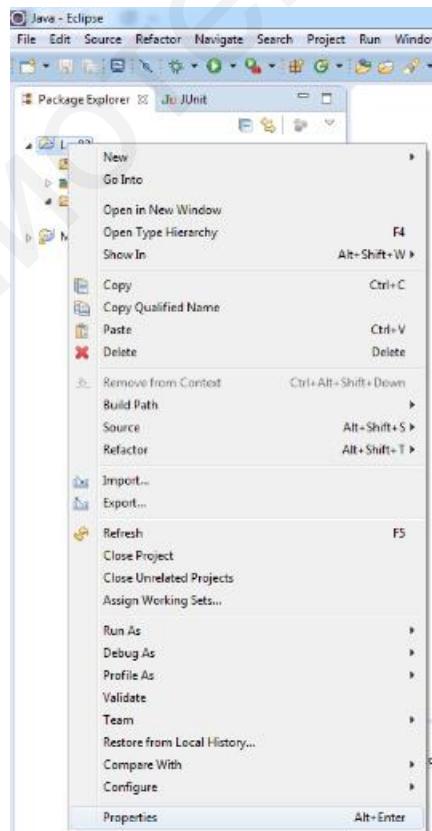
1. Скачайте junit.jar, создайте в своем проекте папку lib и скопируйте туда jar-файл.



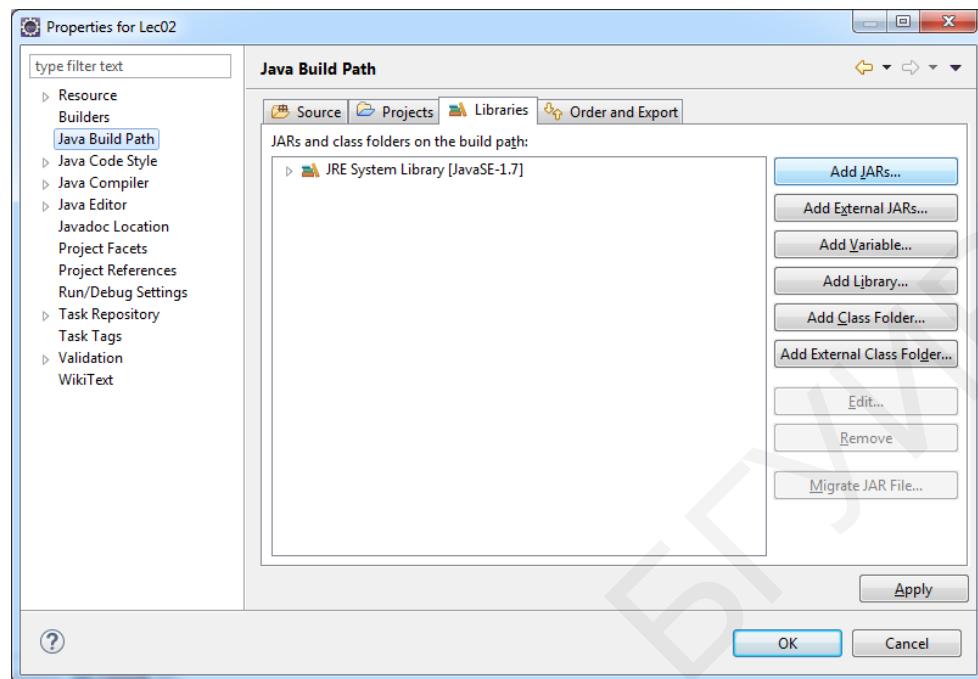
2. Обновите проект в Eclipse, нажав кнопку F5.



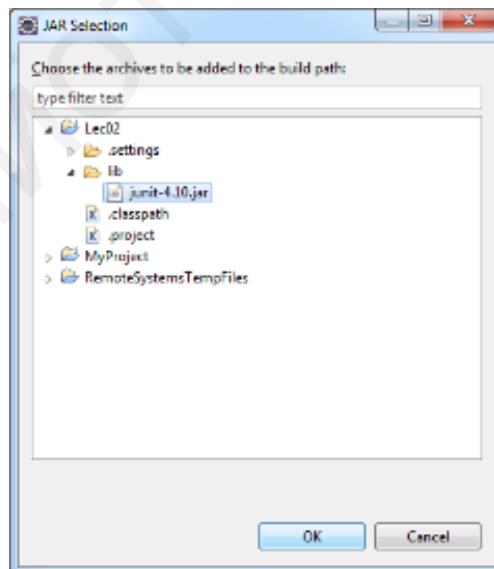
3. Далее необходимо подключить jar-файл к проекту. Выделите имя проекта, щелкните правой кнопкой мыши и в контекстном меню выберите пункт Properties.



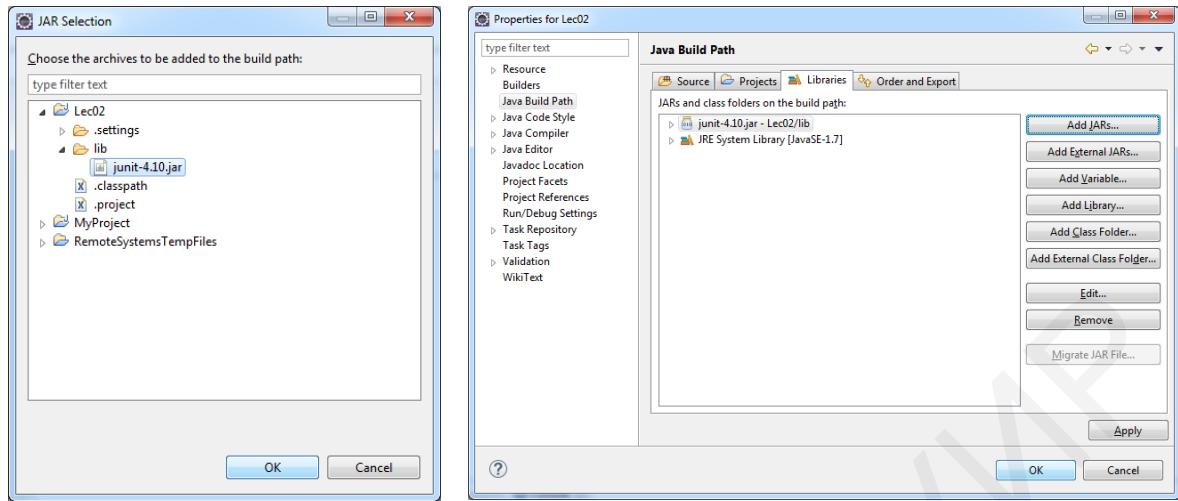
4. В открывшемся окне properties среди списка пунктов настройки выберите Java Build Path, а затем в панели справа с аналогичным названием выберите вкладку Libraries и нажмите кнопку Add JARs...



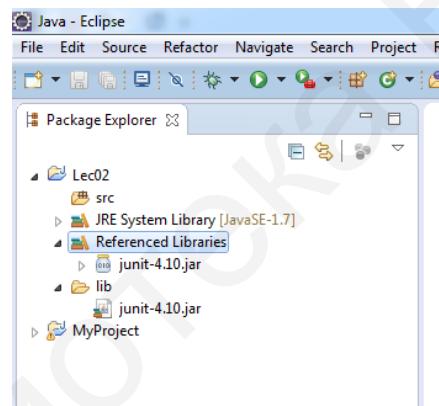
5. В появившемся окне раскройте свой проект и в папке lib выберите jar-файл.



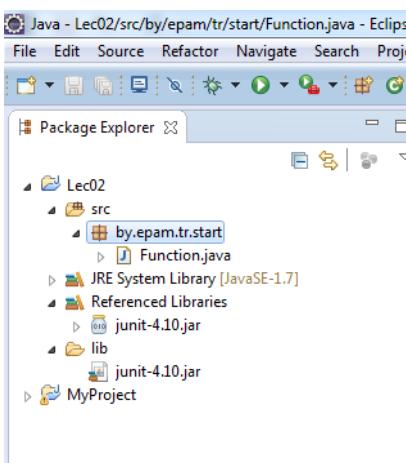
6. Дважды нажмите кнопку OK.



В проекте появилась вкладка на подключенные библиотеки.



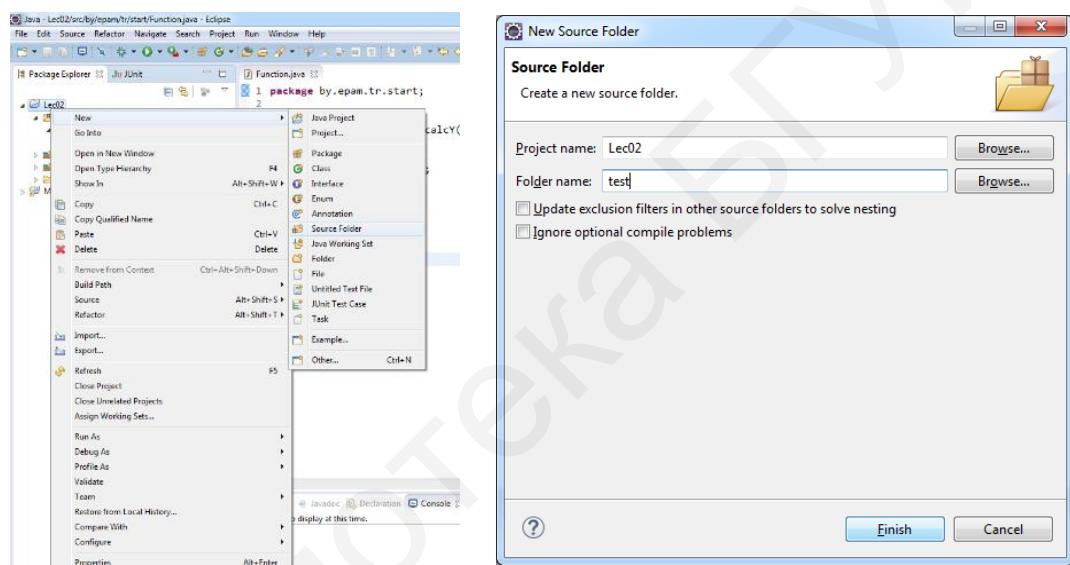
7. Создайте класс Function с пакете by.epam.tr.start.



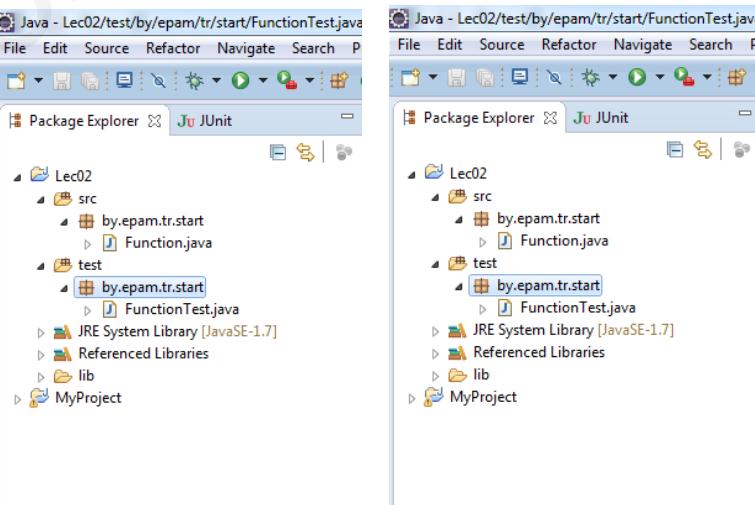
8. Добавьте в класс следующий код:

```
package by.epam.tr.start;
public class Function {
    public static double calcY(double x){
        double y = 0;
        if(x > 3){ y = 1/(x*x+1);
        }else{ y = 9;
        return y;
    }
}
```

9. Создайте новый ресурсный каталог с именем test.



10. Создайте в нем класс FunctionTest в пакете by.epam.tr.start.



11. Добавьте в файл следующий код:

```
package by.epam.tr.start;

import org.junit.Assert;
import org.junit.Test;

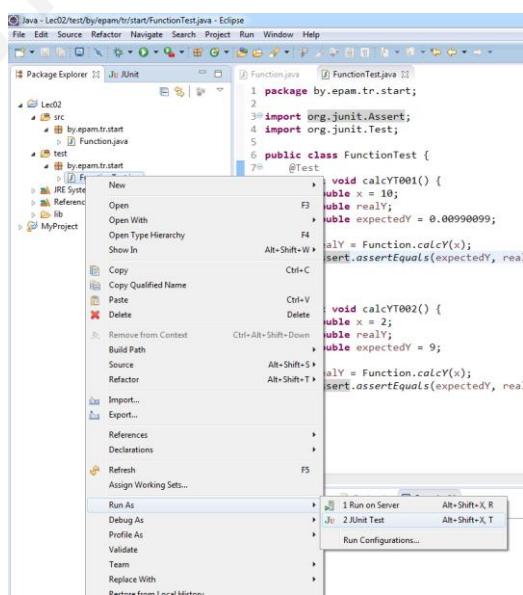
public class FunctionTest {
    @Test
    public void calcYT001() {
        double x = 10;
        double realY;
        double expectedY = 0.00990099;

        realY = Function.calcY(x);
        Assert.assertEquals(expectedY, realY, 0.00000001);
    }

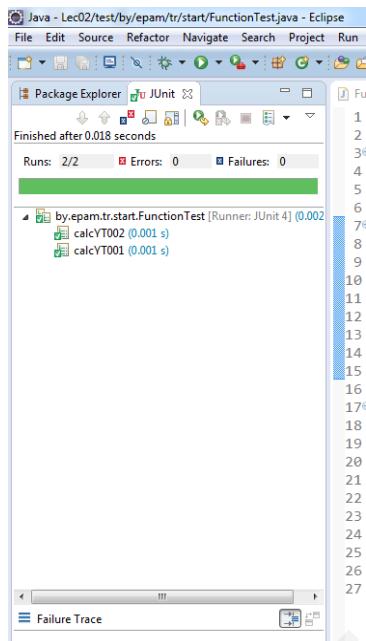
    @Test
    public void calcYT002() {
        double x = 2;
        double realY;
        double expectedY = 9;

        realY = Function.calcY(x);
        Assert.assertEquals(expectedY, realY, 0.00000001);
    }
}
```

12. Запустите приложение как Junit-тест.



13. В окне JUnit вы увидите результаты выполнения тестов.



Запускать Junit-тесты можно не только из-под IDE, но и в консоли.

```
Administrator: C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\Olga_Smoljyakova@epam.com>d:
D:\>cd EclipseWorkspaces\javatr\Lec02

D:\EclipseWorkspaces\javatr\Lec02>java -cp ./bin;EclipseWorkspaces\javatr\Lec02/lib/junit-4.10.jar org.junit.runner.JUnitCore by.epam.tr.start.FunctionTest
JUnit version 4.10
..
Time: 0.01

OK (2 tests)

D:\EclipseWorkspaces\javatr\Lec02>
```

## Создание TestSuite

1. В классах FunctionTest1 и FunctionTest2 определите следующее содержимое:

```
package by.epam.tr.start;
import org.junit.Assert;
import org.junit.Test;
public class FunctionTest1 {
    @Test
    public void calcYT001() {
        double x = 10;
        double realY;
        double expectedY = 0.00990099;
        realY = Function.calcY(x);
```

```

        Assert.assertEquals(expectedY, realY, 0.00000001);
    }
}

package by.epam.tr.start;
import org.junit.Assert;
import org.junit.Test;
public class FunctionTest2 {
    @Test
    public void calcYT0021() {
        double x = 10;
        double realY;
        double expectedY = 0.00990099;
        realY = Function.calcY(x);
        Assert.assertEquals(expectedY, realY, 0.00000001);
    }
}

```

2. Создайте класс Lec02Suite со следующим содержимым:

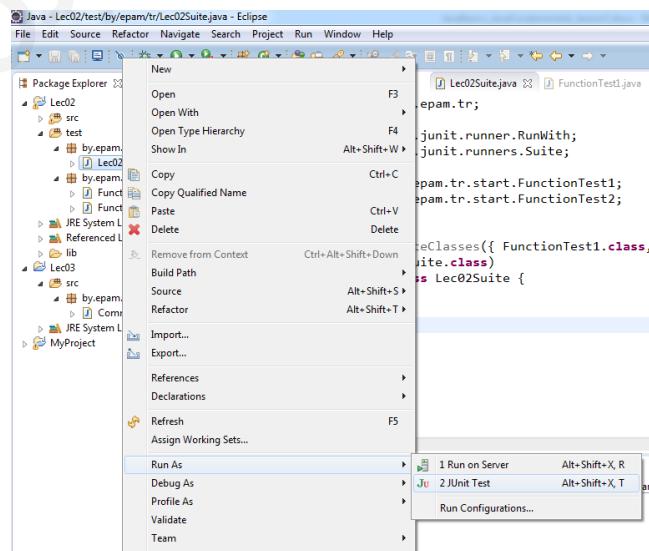
```

package by.epam.tr;
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import by.epam.tr.start.FunctionTest1;
import by.epam.tr.start.FunctionTest2;

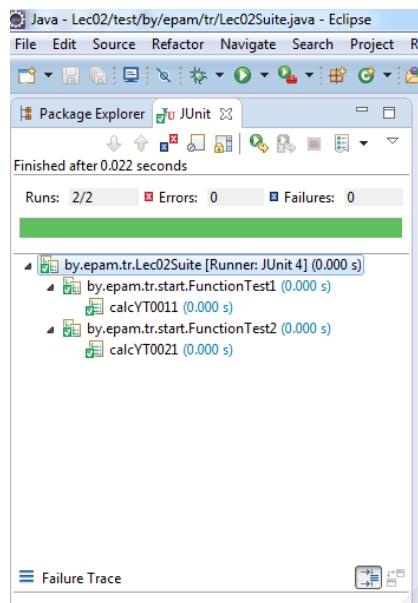
@Suite.SuiteClasses({ FunctionTest1.class, FunctionTest2.class })
@RunWith(Suite.class)
public class Lec02Suite {
}

```

3. Запустите приложение под JUnit.



#### 4. Изучите результаты выполнения Test Suite.



#### Вопросы для закрепления знаний

- Дайте определение понятиям Java – язык программирования и Java-платформа.
- Поясните, как связаны имя java-файла и классы, которые в этом файле объявляются.
- Расшифруйте аббревиатуры JVM, JDK и JRE; покажите, где «они находятся» и что собой представляют.
- Объясните, как скомпилировать и запустить приложение из командной строки, а также зачем в переменных среды окружения прописывать пути к установленному jdk.
- Перечислите атрибуты доступа, объясните их действие.
- Что такое пакеты в java-программе, что представляют собой пакеты на диске? Каково соглашение по именованию пакетов? Как создать пакет?
- Объясните, какие классы, интерфейсы, перечисления необходимо импортировать в вашу программу, как это сделать. Влияет ли импорт пакета на импорт классов, лежащих в подпакетах? Какой пакет в Java импортируется по умолчанию?
- Объясните различия между терминами «объект» и «ссылка на объект».
- Какие примитивные типы Java вы знаете, как создать переменные примитивных типов? Объясните процедуру, по которой переменные примитивных типов передаются в методы как параметры.
- Каков размер примитивных типов, как размер примитивных типов зависит от разрядности платформы, что такое преобразование (приведение) типов и зачем оно необходимо? Какие примитивные типы не приводятся ни к какому другому типу?

- Объясните, что такое явное и неявное приведение типов, приведите примеры, когда такое преобразование имеет место.
- Что такое литералы в Java-программе, какую классификацию литералов вы знаете, как записываются литералы различных видов и типов в Java-программе?
- Как осуществляется работа с типами при вычислении арифметических выражений в Java?
- Что такое классы-оболочки, для чего они предназначены? Объясните, что значит: объект класса оболочки – константный объект.
- Объясните разницу между примитивными и ссылочными типами данных. Поясните существующие различия при передаче параметров примитивных и ссылочных типов в методы. Объясните, как константные объекты ведут себя при передаче в метод.
- Поясните, что такое автоупаковка и автораспаковка.
- Перечислите известные вам арифметические, логические и битовые операторы, определите случаи их употребления. Что такое приоритет оператора, как определить, в какой последовательности будут выполняться операции в выражении, если несколько из них имеют одинаковый приоритет?
- Укажите правила выполнения операций с плавающей точкой в Java (согласно стандарту IEEE754). Как определить, что результатом вычисления стала бесконечность или нечисло?
- Что такое статический импорт, какие элементы можно импортировать при статическом импорте?
- Объясните работу операторов if, switch, while, do-while, for, for-each. Напишите корректные примеры работы этих операторов.
- Объясните работу оператора instanceof. Что будет результатом работы оператора, если слева от него будет стоять ссылка, равная null?
- Дайте определение массиву. Как осуществляется индексация элементов массива? Как необходимо обращаться к  $i$ -му элементу массива?
- Приведите способы объявления и инициализации одномерных и двумерных массивов примитивных и ссылочных типов. Укажите разницу между массивами примитивных и ссылочных типов.
- Объясните, что представляет собой двумерный массив в Java и что такое «рваный массив». Как узнать количество строк и количество элементов в каждой строке для «рваного» массива?
- Объясните ситуации, когда в java-коде могут возникнуть следующие исключительные ситуации java.lang.ArrayIndexOutOfBoundsException и java.lang.ArrayStoreException.

## Лабораторная работа

Общие требования к коду задач, входящих в лабораторную работу:

- При написании приложений обязательно используйте Java Code Convention.
- Не размещайте код всего приложения в одном методе (даже если задача вам кажется маленькой и «там же нечего писать»).
- Обязательно используйте пакеты.
- Не смешивайте в одном классе код, работающий с данными, и логику (даже если вам кажется, что так быстрее). Создавайте разные типы классов: классы, объекты которых хранят данные, и классы, методы которых обрабатывают данные. Размещайте такие классы в разных пакетах.
- Требование к наличию JUnit-тестов является обязательным.
- Именуйте переменные, методы, класс и прочее так, чтобы можно было понять назначение элемента. Не используйте сокращения, только если они не общепринятые.

**Задача 1.** Решить задачу.

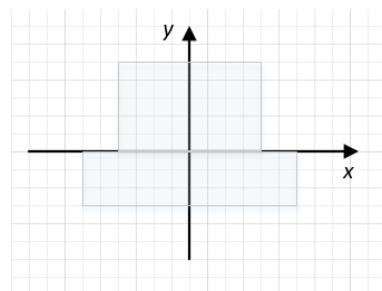
Вычислить значение выражения по формуле (все переменные принимают действительные значения)

$$\frac{1 + \sin^2(x + y)}{2 + \left| x - \frac{2x}{1 + x^2 y^2} \right|} + x.$$

Для модульного тестирования приложения создать JUnit-тесты.

**Задача 2.** Решить задачу.

Для данной области составить программу, которая печатает true, если точка с координатами  $(x, y)$  принадлежит закрашенной области, и false – в противном случае. Для модульного тестирования приложения создать JUnit-тесты.



**Задача 3.** Решить задачу.

Составить программу для вычисления значений функции  $F(x) = \operatorname{tg}(x)$  на отрезке  $[a, b]$  с шагом  $h$ . Результат представить в виде таблицы, первый столбец которой – значения аргумента, второй – соответствующие значения функции. Для модульного тестирования приложения создать JUnit-тесты.

**Задача 4.** Решить задачу.

Задан целочисленный массив размерностью  $N$ . Есть ли среди элементов массива простые числа? Если да, то вывести номера этих элементов. Для модульного тестирования приложения создать JUnit-тесты.

**Задача 5.** Решить задачу.

Дана целочисленная таблица  $A[n]$ . Найти наименьшее число  $K$  элементов, которые можно выкинуть из данной последовательности, так чтобы осталась возрастающая подпоследовательность. Для модульного тестирования приложения создать JUnit-тесты.

**Задача 6.** Решить задачу.

Даны действительные числа  $a_1, a_2, \dots, a_n$ . Получить следующую квадратную матрицу порядка  $n$ .

$$\begin{pmatrix} a_1 & a_2 & a_3 & \cdots & a_{n-2} & a_{n-1} & a_n \\ a_2 & a_3 & a_4 & \cdots & a_{n-1} & a_n & a_1 \\ a_3 & a_4 & a_5 & \cdots & a_n & a_1 & a_2 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ a_{n-1} & a_n & a_1 & \cdots & a_{n-4} & a_{n-3} & a_{n-2} \\ a_n & a_1 & a_2 & \cdots & a_{n-3} & a_{n-2} & a_{n-1} \end{pmatrix}$$

Для модульного тестирования приложения создать JUnit-тесты.

**Задача 7.** Решить задачу.

Сортировка Шелла. Дан массив  $n$  действительных чисел. Требуется упорядочить его по возрастанию. Делается это следующим образом: сравниваются два соседних элемента  $a_i$  и  $a_{i+1}$ . Если  $a_i \leq a_{i+1}$ , то числа продвигаются на один элемент вперед. Если  $a_i > a_{i+1}$ , то производится перестановка и числа сдвигаются на один элемент назад. Составить алгоритм этой сортировки.

**Задача 8.** Решить задачу.

Пусть даны две неубывающие последовательности действительных чисел  $a_1 \leq a_2 \leq \dots \leq a_n$  и  $b_1 \leq b_2 \leq \dots \leq b_m$ . Требуется указать те места, в которые нужно вставлять элементы последовательности  $b_1 \leq b_2 \leq \dots \leq b_m$  в первую последовательность так, чтобы новая последовательность оставалась возрастающей.

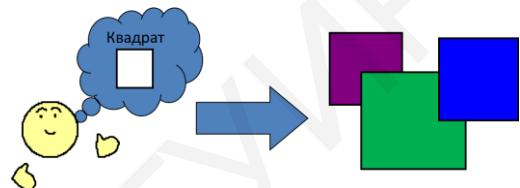
## 2. КЛАССЫ И ОБЪЕКТЫ

Вопросы в начале темы:

- Попытайтесь ответить на вопрос: почему класс – это тип данных?
- Как вы думаете, всегда ли при программировании удобно пользоваться классами?
- Классы и интерфейсы – это одно и то же, или все-таки есть различия, и не только в способе объявления?

### Определения

Объект – некоторая КОНКРЕТНАЯ сущность моделируемой предметной области. Класс – шаблон или АБСТРАКЦИЯ сущности предметной области.



### Класс

**Классом** называется описание совокупности объектов с общими атрибутами, методами, отношениями и семантикой. Классы определяют структуру и поведение некоторого набора элементов предметной области, для которой разрабатывается программная модель.

*Объявление класса имеет вид:*

```
[спецификаторы] class имя_класса
    [extends суперкласс] [implements список_интерфейсов]{
        /*определение класса*/
    }
```

### Спецификаторы класса

Спецификатор класса может быть:

- **public** (класс доступен объектам данного пакета и вне пакета);
- **final** (класс не может иметь подклассов);
- **abstract** (класс содержит абстрактные методы, объекты такого класса могут создавать только подклассы).

По умолчанию спецификатор доступа устанавливается в **friendly(package)** (класс доступен в данном пакете). Данное слово при объявлении вообще не используется и не является ключевым словом языка, мы его используем для обозначения.

## **Свойства и методы класса**

**Определение класса включает:**

- модификатор доступа;
- ключевое слово **class**;
- свойства класса;
- конструкторы;
- методы;
- статические свойства;
- статические методы.

**Объект состоит из следующих трех частей:**

- имя объекта;
- состояние (переменные состояния);
- методы (операции).

**Свойства классов:**

- уникальные характеристики, которые необходимы при моделировании предметной области;
- ОБЪЕКТЫ различаются значениями свойств;
- свойства отражают состояние объекта.

**Методы классов:**

- метод отражает ПОВЕДЕНИЕ объектов;
- выполнение методов, как правило, меняет значение свойств;
- поведение объекта может меняться в зависимости от состояния.

## **Методы**

Все функции определяются внутри классов и называются **методами**.

*Объявление метода имеет вид*

[спецификаторы] [static|abstract] возвращаемый\_тип  
имя\_метода([аргументы]) {  
    /\*тело метода\*/  
}|;

*Невозможно создать метод, не являющийся методом класса или объявить метод вне класса.*

**Спецификаторы доступа методов:**

static	public	friendly	synchronized
final	private	native	
protected	abstract	strictfp	

## Поля

Данные – члены класса, которые называются полями или переменными класса, объявляются в классе следующим образом:

**спецификатор тип имя;**

**Спецификаторы доступа полей класса:**

static	public	final	private
protected	friendly	transient	volatile

## Конструкторы

**Конструктор** – это метод, который автоматически вызывается при создании объекта класса и выполняет действия *только* по инициализации объекта. Конструктору характерно:

- то же имя, что и класс;
- вызов не по имени, а только вместе с ключевым словом **new** при создании экземпляра класса;
- не возвращает значение, но может иметь параметры и быть перегружаемым.

*Создание объекта имеет вид*

**имя\_класса имя\_объекта = new конструктор\_класса([аргументы]);**

```
public class BookUse {  
  
    public static void main(String[] args) {  
        Book book = new Book("Java");  
        System.out.println(book.getTitle());  
    }  
}  
public class Book {  
    private String title;  
  
    public Book() {  
        setTitle("without a title");  
    }  
    public Book(String title) {  
        setTitle(title);  
    }  
  
    public void setTitle(String title) {
```

```

    if (null == title) {
        this.title = "no title";
    } else {
        this.title = title;
    }
}

public String getTitle() {
    return title;
}
}

```

## Передача параметров в методы

Ссылки в методы передаются по значению.

Выделяется память под параметры метода, и те переменные, которые являются ссылочными аргументами, инициализируются значением своих фактических параметров. Таким образом, минимум две ссылки начинают указывать на один объект.

```

import java.util.Date;
public class TransferParameter {

    public static void main(String[] args) {
        Date myDate = new Date();
        System.out.println("Before:" + myDate.getDate());
        changeDate(myDate);
        System.out.println("After:" + myDate.getDate());
    }

    public static void changeDate(Date date) {
        System.out.println("      - before change: " + date.getDate());
        date.setDate(12);
        System.out.println("      - after change: " + date.getDate());
    }
}

```

## Передача константных объектов в методы

При передаче в метод аргумента-ссылки можно изменить состояние объекта и оно сохранится после возвращения из метода, т. к. в этом случае новый объект не создается, а создается лишь новая ссылка, указывающая на старый объект. Из этого правила существует одно исключение – когда передается ссылка, указывающая на константный объект.

Константный объект – это такой объект, изменить состояние которого нельзя. При попытке его изменить создается новый модифицированный объект. Примером таких объектов являются объекты класса String.

Если необходимо вернуть в вызывающий метод ссылку на новый **константный** объект, созданный в этом методе, следует указать ее тип как тип возвращаемого методом значения и использовать **return**.

```
public class StringForChange {  
    public static void main(String[] args) {  
        String str = "I like ";  
        System.out.println("Before: " + str);  
        changeString(str);  
        System.out.println("After: " + str);  
    }  
  
    public static void changeString(String s) {  
        System.out.println("      - before change: " + s);  
        s = s + " Java.>";  
        System.out.println("      - after change: " + s);  
    }  
}
```

## Вызов методов

Для вызова методов в Java используется оператор . (**dot operator**)

```
public class BookMethodInspector {  
  
    public static void main(String[] args) {  
        Book book = new Book();  
        book.setPrice(45_000);  
    }  
}
```

## Основы работы со строками

Создание строкового объекта: `String s1 = new String("World");`  
Можно использовать упрощенный синтаксис:

```
String s; //создание ссылки  
s = "Hello"; //присвоение значения
```

Знак «+» применяется для объединения двух строк. Если в строковом выражении применяется нестроковый аргумент, то он преобразуется к строке автоматически.

Чтобы сравнить на равенство две строки, необходимо воспользоваться методом **equals()**:

```
if(str1.equals(str2)){ }
```

Длина строки определяется с помощью метода **length()**:

```
int len = str.length();
```

**Пул литералов** – это коллекция ссылок на строковые объекты.

- Пул литералов представляет все литералы, созданные в программе.
- Каждый раз, когда создаются строковые литералы, аналогичный литерал ищется в пуле.
- Если создаваемый литерал уже существует в пуле, то новый экземпляр для него не создается, а возвращается адрес уже имеющегося.

```
public class ComparingStrings {  
    public static void main(String[] args) {  
        String s1, s2;  
        s1 = "Java";  
        s2 = "Java";  
        System.out.println("сравнение ссылок " + (s1 == s2));  
        s1 += '2';  
        // s1="a"; //ошибка, вычитать строки нельзя  
        s2 = new String(s1);  
        System.out.println("сравнение ссылок " + (s1 == s2));  
        System.out.println("сравнение значений " + s1.equals(s2));  
    }  
}
```

## Объектно-ориентированное программирование на Java. Принципы ООП

Объектно-ориентированное программирование основано на трех принципах (**инкапсуляции, наследовании, полиморфизме**) и одном механизме (**позднем (динамическом) связывании**).

**Инкапсуляция** (encapsulation) – это механизм, который объединяет данные и код, манипулирующий этими данными, а также защищает и то, и другое от внешнего вмешательства или неправильного использования.

**Наследование** (inheritance) – это процесс, посредством которого один объект может наследовать свойства другого объекта и добавлять к ним черты, характерные только для него. Наследование бывает двух видов:

- *одиночное* – когда каждый класс имеет одного и только одного предка;

- **множественное** – когда каждый класс может иметь любое количество предков.

**Полиморфизм** (polymorphism) – это свойство, которое позволяет одно и то же имя использовать для решения двух или более схожих, но технически разных задач.

**Динамическое связывание** (dynamic binding) – связывание, при котором ассоциация между ссылкой (именем) и классом не устанавливается, пока объект с заданным именем не будет создан на стадии выполнения программы.

## Перегрузка методов

**Overload (перегрузка метода)** – определение методов с одинаковым наименованием, но различной сигнатурой. Фактически такие методы – это совершенно разные методы с совпадающим наименованием. Сигнатура метода определяется наименованием метода, а также числом и типом параметров метода.

```
import java.util.Date;

public class DatePrinter {

    public int printDate(String s) {
        System.out.printf("String s=", s);
        return 1;
    }

    public void printDate(int day, int month, int year) {
        System.out.println("int day=" + day);
    }

    public static void printDate(Date d) {
        System.out.printf("Date d=", d);
    }
}

import java.util.Date;
public class DatePrinterInspector {

    public static void main(String[] args) {
        DatePrinter dp = new DatePrinter();
        int x = dp.printDate("01.01.2015");
        dp.printDate(new Date());
    }
}
```

```
        dp.printDate(1, 1, 2015);
    }
}
```

Перегрузка реализует концепцию раннего связывания. Статические методы могут перегружаться нестатическими и наоборот без ограничений. При непосредственной передаче объекта в метод выбор производится в зависимости от типа ссылки на этапе компиляции.

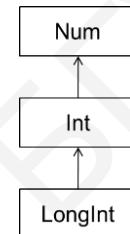
## Перегрузка конструкторов. Сложные варианты перегрузки

Конструкторы перегружаются аналогично другим методам.

```
public class Mathematica {
    public Mathematica(Num obj){ }
    public Mathematica(Int obj){ }
    public Mathematica(Num obj1, Int obj2){ }
    public Mathematica(Int obj1, Int obj2) { }

    public static void main(String[] args){
        Num o1 = new Num();
        Int o2 = new Int();
        LongInt o3 = new LongInt();
        Num o4 = new Int();

        Mathematica m1 = new Mathematica(o1);
        Mathematica m2 = new Mathematica(o2);
        Mathematica m3 = new Mathematica(o3);
        Mathematica m4 = new Mathematica(o4);
        Mathematica m5 = new Mathematica(o1, o2);
        Mathematica m6 = new Mathematica(o3, o2);
        Mathematica m7 = new Mathematica(o1, o4);//error
        Mathematica m8 = new Mathematica(o3, o4);//error
    }
}
```



}

При перегрузке всегда следует придерживаться следующих правил:

- не использовать сложных вариантов перегрузки;
- не использовать перегрузку с одинаковым числом параметров;
- заменять при возможности перегруженные методы на несколько разных методов.

## Применение this в конструкторе

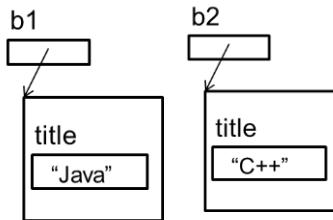
Для вызова тела одного конструктора из другого первым оператором вызывающего конструктора должен быть оператор **this([аргументы])**.

```
public class Point2D {  
  
    private int x;  
    private int y;  
  
    public Point2D(int x, int y) {  
        this.x = x;           this.y = y;  
    }  
  
    public Point2D(int size) {           this(size, size); }  
}
```

## Явные и неявные параметры метода

Явные параметры метода определяются списком параметров. Неявный параметр – это **this** – ссылка на вызвавший метод объект.

```
public class Book {  
    private String title;  
  
    public Book(String title) {  
        this.title = title;  
    }  
  
    public String getTitle() {  
        return this.title;  
    }  
}
```



```
Book b1 = new Book("Java");  
Book b2 = new Book("C++");  
String s1 = b1.getTitle();  
String s2 = b2.getTitle();
```

<pre>public String getTitle(Book this) {     return this.title; }</pre>	b1.getTitle() == getTitle(b1)
---	-------------------------------

## Статические методы:

- являются методами класса;
- не привязаны ни к какому объекту;
- не содержат указателя **this** на конкретный объект, вызвавший метод;
- реализуют парадигму раннего связывания, жестко определяющую версию метода на этапе компиляции.

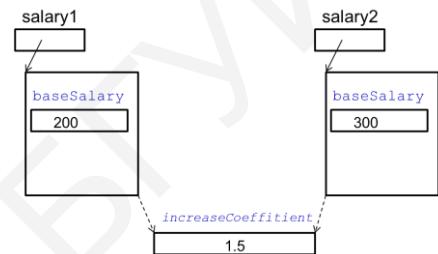
Объявление статического метода имеет вид

```
[спецификаторы] static возвращаемый_тип  
    имя_метода([аргументы]) {  
        /*тело метода*/}
```

## Статические поля

```
Salary salary1 = new Salary(200);  
Salary salary2 = new Salary(300);
```

Поля данных, объявленные в классе как **static**, являются общими для всех объектов класса и называются **переменными класса**. Если один объект изменит значение такого поля, то это изменение увидят все объекты.



```
public class Salary {  
    private double baseSalary;  
    public static double increaseCoefficient = 1.5;  
  
    public Salary(double baseSalary) {  
        if (baseSalary <= 0) {  
            this.baseSalary = 100;  
        } else {  
            this.baseSalary = baseSalary;  
        }  
    }  
  
    public double calcSalary() {  
        return baseSalary * increaseCoefficient;  
    }  
}
```

Статические поля и методы не могут обращаться к нестатическим полям и методам напрямую (по причине недоступности ссылки **this**), т. к. для обращения к статическим полям и методам достаточно имени класса, в котором они определены.

```

public class SalaryWithWrongStatic {
    private double baseSalary;
    public static double increaseCoeffitient = 1.5;

    public SalaryWithWrongStatic(double baseSalary) {
        if (baseSalary <= 0) {
            this.baseSalary = 100;
        } else {
            this.baseSalary = baseSalary;
        }
    }

    public double calcSalary() {
        return baseSalary * increaseCoeffitient;
    }

    public static void setIncreaseCoeffitient(double newIncreaseCoeffitient)
    {
        if (newIncreaseCoeffitient <= 0) {
            throw new IllegalArgumentException(
                "Wrong parameter: newIncreaseCoeffitient = "
                + newIncreaseCoeffitient);
        }
        increaseCoeffitient = newIncreaseCoeffitient;
        // calcSalary(); // ERROR
    }
}

```

## Применение статических методов

Статические методы не работают с объектами, поэтому их использовать следует в двух случаях:

- когда методу *не нужен доступ к состоянию объекта*, а все необходимые параметры задаются явно (например, метод Math.pow(...));
- когда методу нужен *доступ только к статическим полям* класса (статический метод не может получить доступ к нестатическим полям класса, т. к. они принадлежат объектам, а не классам).

**Статические методы можно вызывать, даже если ни один объект этого класса не создан.** Кроме того, статические методы часто используют в качестве порождающих.

```
public class SimpleSingleton {  
    private static SimpleSingleton instance;  
  
    private SimpleSingleton(){ }  
  
    public static SimpleSingleton getInstance()  
    {  
        if (null == instance){  
            instance = new SimpleSingleton();  
        }  
        return instance;  
    }  
}
```

Статические поля используются довольно редко, а поля **static final** – наоборот часто. Статические константы нет смысла делать закрытыми, а обращаются к ним через имя класса:

**имя\_класса.имя\_статической\_константы**

```
public class System  
{  
    ...  
    public static final PrintStream out = ...  
    ...  
}
```

### Логический блок инициализации

При описании класса могут быть использованы логические блоки. **Логическим блоком называется код**, заключенный в фигурные скобки и не принадлежащий ни одному методу текущего класса:

**{ /\* код \*/ }**

При создании объекта блоки инициализации класса вызываются последовательно, в порядке размещения, вместе с инициализацией полей как простая последовательность операторов, и только после выполнения последнего блока будет вызван конструктор класса.

```
public class LogicBlock {  
    private int x = 89;  
  
    {      x = 20;      }  
  
    public LogicBlock() { }  
}
```

```

public LogicBlock(int x) {           this.x = x; }
public int getX() {
    return x;
}

public static void main(String[] args) {
    LogicBlock logic1 = new LogicBlock();
    LogicBlock logic2 = new LogicBlock(200);
    System.out.println("logic1.x = " + logic1.getX());
    System.out.println("logic2.x = " + logic2.getX());
}
}

```

Логические блоки чаще всего используются в качестве инициализаторов полей, но могут содержать вызовы методов как текущего класса, так и не принадлежащих классов.

```

import java.util.Date;
public class LogicBlock2 {

{
    System.out.println("logic (1) id=" + this.id);
}
private int id = 7;
{
    System.out.println("logic (2) id=" + id);
    Date d = new Date();
    calc(d);
}
public LogicBlock2(int d) {
    id = d;  System.out.println("конструктор id=" + id);
}

{
    id = 10;
    System.out.println("logic (3) id=" + id);
}

private void calc(Date d){      System.out.println(d.getTime());      }

public static void main(String[] args) {
    LogicBlock2 logic = new LogicBlock2(3);
}
}

```

## Статические блоки инициализации

Для инициализации статических переменных существуют **статические блоки инициализации**. В этом случае фигурные скобки предваряются ключевым словом **static**. Такой блок вызывается только один раз в жизненном цикле приложения: либо при **создании объекта**, либо при **обращении к статическому методу (полю)** данного класса.

```
public class StaticBlock {  
    private double baseSalary;  
    public static double increaseCoeffitient = 2.5;  
  
    static{  
        increaseCoeffitient = 1.5;  
        //baseSalary = 100; // error  
    }  
  
    public StaticBlock(double baseSalary) { this.baseSalary = baseSalary; }  
  
    public double calcSalary() {return baseSalary * increaseCoeffitient; }  
  
    public static void setIncreaseCoeffitient(double newIncreaseCoeffitient)  
    {  
        increaseCoeffitient = newIncreaseCoeffitient;  
    }  
}
```

## Инициализация полей класса

Общий порядок инициализации следующий:

1. При создании объекта или при первом обращении к статическому методу (полю) статические поля инициализируются значениями по умолчанию.
2. Инициализаторы всех статических полей и статические блоки инициализации выполняются в порядке их перечисления в объявлении класса.
3. При вызове конструктора класса все поля данных инициализируются своими значениями, предусмотренными по умолчанию.
4. Инициализаторы всех полей и блоки инициализации выполняются в порядке их перечисления в объявлении класса.
5. Если в первой строке конструктора вызывается тело другого конструктора, то выполняется вызванный конструктор.
6. Выполняется тело конструктора.

## Модификатор final

Модификатор **final** используется для определения констант в качестве члена класса, локальной переменной или параметра метода. **Константа может быть объявлена как поле экземпляра класса, но не проинициализирована.** В этом случае она должна быть проинициализирована в логическом блоке класса или конструкторе, но только в одном из указанных мест. Константные статические поля могут быть проинициализированы или при объявлении, или в статическом блоке инициализации. Значение по умолчанию константа получить не может в отличие от переменных класса.

```
import java.util.Date;

public class FinalVar {
    private final int finalVar;
    public static final int staticFinalVar;
    private final Date date;

    static {
        staticFinalVar = 2;
    }

    {
        finalVar = 1;
    }

    public void method(final int var) {
        final int temp = 12;
        // var++; // error
        date.setYear(2999 - 1900);
        // date = new Date(); //error
    }

    public FinalVar() {
        // finalVar = 3;//error
        // staticFinalVar = 4;//error
        date = new Date();
    }
}
```

## Модификатор native

Модификатор **native** указывает на то, что метод написан не на языке Java. Методы, помеченные **native**, можно переопределять обычными методами в подклассах. Тело нативного метода должно заканчиваться на **(;)**, как в абстрактных методах, идентифицируя то, что реализация опущена.

```
public native int loadCripto(int num);
```

```
package java.lang;
public class Object {
...
protected native Object clone() throws CloneNotSupportedException;
...
}
```

## Спецификатор synchronized

При использовании нескольких потоков управления в одном приложении необходимо синхронизировать методы, обращающиеся к общим данным. Когда интерпретатор обнаруживает **synchronized**, он включает код, блокирующий доступ к данным при запуске потока и снимающий блок при его завершении.

```
public class SinchronizedSingleton {
    private static SinchronizedSingleton instance;

    private SinchronizedSingleton(){}
    public static synchronized SinchronizedSingleton getInstance() {
        if (null == instance){
            instance = new SinchronizedSingleton();
        }
        return instance;
    }
}
```

## Класс Object

**Класс java.lang.Object** – родительский для всех классов.

Содержит следующие методы:

- **protected Object clone()** – создает и возвращает копию вызывающего объекта;

- **boolean equals(Object ob)** – предназначен для переопределения в подклассах с выполнением общих соглашений о сравнении содержимого двух объектов;
- **Class<? extends Object> getClass()** – возвращает объект типа **Class**;
- **protected void finalize()** – вызывается перед уничтожением объекта автоматическим сборщиком мусора (garbage collection);
- **int hashCode()** – возвращает хэш-код объекта;
- **String toString()** – возвращает представление объекта в виде строки.

## Переопределение метода equals()

Метод **equals()** при сравнении двух объектов возвращает истину, если содержимое объектов эквивалентно, и ложь – в противном случае.

При переопределении должны выполняться соглашения:

- **рефлексивность** – объект равен самому себе;
- **симметричность** – если **x.equals(y)** возвращает значение **true**, то и **y.equals(x)** всегда возвращает значение **true**;
- **транзитивность** – если метод **equals()** возвращает значение **true** при сравнении объектов **x** и **y**, а также **y** и **z**, то и при сравнении **x** и **z** будет возвращено значение **true**;
- **непротиворечивость** – при многократном вызове метода для двух не подвергшихся изменению за это время объектов возвращаемое значение всегда должно быть одинаковым;
- **ненулевая ссылка** при сравнении с литералом **null** всегда возвращает значение **false**.

## Переопределение метода hashCode()

Метод **int hashCode()** возвращает хэш-код объекта, вычисление которого управляется следующими соглашениями:

- во время работы приложения значение хэш-кода объекта не изменяется, если объект не был изменен;
- все одинаковые по содержанию объекты одного типа должны иметь одинаковые хэш-коды;
- различные по содержанию объекты одного типа могут иметь различные хэш-коды.

*Метод hashCode() следует переопределять всегда, когда переопределен метод equals().*

## Переопределение метода toString()

Метод **toString()** следует переопределять таким образом, чтобы кроме стандартной информации о пакете (опционально), в котором находится

класс, и самого имени класса (опционально), он возвращал значения полей объекта, вызвавшего этот метод (т. е. всю полезную информацию объекта), вместо хэш-кода, как это делается в классе **Object**. В классе **Object** возвращает строку с описанием объекта в виде

```
getClass().getName() + '@' + Integer.toHexString(hashCode())
```

```
class Pen {  
    private int price;  
    private String producerName;  
  
    public Pen(int price, String producerName) {  
        this.price = price;          this.producerName = producerName;  
    }  
  
    public boolean equals(Object obj) {  
        if (this == obj) {  
            return true;  
        }  
        if (null == obj) {  
            return false;  
        }  
        if (getClass() != obj.getClass()) {  
            return false;  
        }  
  
        Pen pen = (Pen) obj;  
        if (price != pen.price) {  
            return false;  
        }  
        if (null == producerName) {  
            return (producerName == pen.producerName);  
        } else {  
            if (!producerName.equals(pen.producerName)) {  
                return false;  
            }  
        }  
        return true;  
    }  
    public int hashCode() {  
        return (int) (31 * price + ((null == producerName) ? 0 :  
producerName.hashCode()));  
    }  
}
```

```
public String toString() {
    return getClass().getName() + "@" + "price: " + price
           + ", producerName: " + producerName;
}
}
```

## Метод finalize

Иногда при уничтожении объект должен выполнять какое-либо действие. Используя метод **finalize()**, можно определить конкретные действия, которые будут выполняться непосредственно перед удалением объекта сборщиком мусора.

```
protected void finalize() throws Throwable{
    try{
        // освобождение ресурсов
    }finally{
        super.finalize();
    }
}
```

Метод **finalize()** вызывается, когда сборщик мусора решит уничтожить объект. «Сборка мусора» происходит нерегулярно во время выполнения программы. Можно ее выполнить вызовом метода **System.gc()** или **Runtime.getRuntime().gc()**. Вызов метода **System.runFinalization()** приведет к запуску метода **finalize()** для объектов, утративших все ссылки. По возможности следует избегать использования метода **finalize** (из-за невозможности предсказать последствия его работы). Лучше освобождать ресурсы программно.

## Методы с переменным числом параметров

Возможность передачи в метод нефиксированного числа параметров позволяет отказаться от предварительного создания массива объектов для его последующей передачи в метод.

```
void methodName(Тип ... args){ }
```

Чтобы передать несколько массивов в метод по ссылке, следует использовать следующее объявление:

```
void methodName(Тип[]... args){ }
```

В списке аргументов аргумент с переменным числом параметров должен быть самым последним:

```
void methodName(char s, int ... args){ }
void methodName(int ... x, char s){ } //error
```

Методы с переменным числом аргументов могут быть перегружены:

```
void methodName(Integer...args) {}
void methodName(int x1, int x2) {}
void methodName(String...args) {}
```

```
public class VarArgs {
    public static int getArgCount(Integer... args) {
        if (args.length == 0) {
            System.out.print("No arg");
        }
        for (int i : args) {
            System.out.print("arg:" + i + " ");
        }
        return args.length;
    }

    public static void getArgCount(Integer[]... args) {
        if (args.length == 0) {
            System.out.print("No arg2");
        }
        for (Integer[] mas : args) {
            for (int x : mas) {
                System.out.print("arg2:" + x + " ");
            }
        }
    }

    public static void main(String args[]) {
        System.out.println("N=" + getArgCount(7, 71, 555));
        Integer[] i = { 1, 2, 3, 4, 5, 6, 7 };
        System.out.println("N=" + getArgCount(i));
        getArgCount(i, i);
        // getArgCount(); //error
    }
}
```

```

public class DemoOverload {
    public static void printArgCount(Object... args) { // 1
        System.out.println("Object args: " + args.length);
    }
    public static void printArgCount(Integer[]... args) { // 2
        System.out.println("Integer[] args: " + args.length);
    }
    public static void printArgCount(int... args) { // 3
        System.out.print("int args: " + args.length);
    }
    public static void main(String[] args) {
        Integer[] i = { 1, 2, 3, 4, 5 };
        printArgCount(7, "No", true, null);
        printArgCount(i, i, i);
        printArgCount(i, 4, 71);
        printArgCount(i); // будет вызван метод 1
        printArgCount(5, 7); //неопределенность!
    }
}

```

## **Вопросы для закрепления знаний**

- Дайте развернутое объяснение трем концепциям ООП.
- Дайте определение таким понятиям, как «класс» и «объект». Приведите примеры объявления класса и создания объекта класса. Какие спецификаторы можно использовать при объявлении класса?
- Как вы решаете, какие поля и методы необходимо определить в классе? Приведите пример. Какие спецификаторы можно использовать с полями, а какие – с методами (и что они значат)?
- Что такое конструктор? Как вы отличите конструктор от любого другого метода? Сколько конструкторов может быть в классе? Что такое конструктор по умолчанию, может ли в классе совсем не быть конструкторов? Объясните, что делает оператор this() в конструкторе?
- Приведите правила, которым должен следовать компонент java-bean.
- Опишите процедуру инициализации полей класса и полей экземпляра класса. Когда инициализируются поля класса, а когда – поля экземпляров класса. Какие значения присваиваются полям по умолчанию? Где еще в классе полям могут быть присвоены начальные значения?
- Дайте определение перегрузке методов. Как вы думаете, чем удобна перегрузка методов? Укажите, какие методы могут перегружаться и какими методами они могут быть перегружены? Можно ли перегрузить методы в базовом и производном классах? Можно ли private-метод базового класса перегрузить public-методом производного? Можно ли

- перегрузить конструкторы, и можно ли при перегрузке конструкторов менять атрибуты доступа у конструкторов?
- Объясните, что такое раннее и позднее связывание? Перегрузка – это раннее или позднее связывание? Объясните правила, которым следует компилятор при разрешении перегрузки; в том числе, если методы перегружаются примитивными типами, между которыми возможно неявное приведение, или ссылочными типами, состоящими в иерархической связи.
  - Объясните, как вы понимаете, что такое неявная ссылка this? В каких методах эта ссылка присутствует, а в каких – нет, и почему?
  - Что такое финальные поля, какие поля можно объявить со спецификатором final? Где можно инициализировать финальные поля?
  - Что такое статические поля, статические финальные поля и статические методы? К чему имеют доступ статические методы? Можно ли перегрузить и переопределить статические методы? Наследуются ли статические методы?
  - Что такое логические и статические блоки инициализации? Сколько их может быть в классе, в каком порядке они могут быть размещены и в каком порядке вызываются?
  - Что представляют собой методы с переменным числом параметров, как передаются параметры в такие методы и что представляет собой такой параметр в методе? Как осуществляется выбор подходящего метода при использовании перегрузки для методов с переменным числом параметров?
  - Чем является класс Object? Перечислите известные вам методы класса Object, укажите их назначение.
  - Что такое хэш-значение? Объясните, почему два разных объекта могут генерировать одинаковые хэш-коды?
  - Что такое объект класса Class? Чем использование метода getClass() и последующего сравнения возвращенного значения с Type.class отличается от использования оператора instanceof?
  - Укажите правила переопределения методов equals(), hashCode() и toString().
  - Что такое абстрактные классы и методы? Зачем они нужны? Бывают ли случаи, когда абстрактные методы содержат тело? Можно ли в абстрактных классах определять конструкторы? Могут ли абстрактные классы содержать неабстрактные методы? Можно ли от абстрактных классов создавать объекты и почему?
  - Что такое интерфейсы? Как определить и реализовать интерфейс в java-программе? Укажите спецификаторы, которые приобретают методы и поля, определенные в интерфейсе. Можно ли описывать в интерфейсе конструкторы и создавать объекты? Можно ли создавать интерфейсные ссылки и если да, то на какие объекты они могут ссылаться?

- Для чего служит интерфейс Clonable? Как правильно переопределить метод clone() класса Object, для того чтобы объект мог создавать свои адекватные копии?
- Для чего служат интерфейсы Comparable и Comparator? В каких случаях предпочтительнее использовать первый, а в каких – второй? Как их реализовать и использовать?

## Лабораторная работа

Общие требования к коду задач, входящих в лабораторную работу:

- При написании приложений обязательно используйте Java Code Convention.
- Не размещайте код всего приложения в одном методе (даже если задача вам кажется маленькой и «там же нечего писать»).
- Обязательно используйте пакеты.
- Не смешивайте в одном классе код, работающий с данными, и логику (даже если вам кажется, что так быстрее). Создавайте разные типы классов: классы, объекты которых хранят данные, и классы, методы которых обрабатывают данные. Размещайте такие классы в разных пакетах.
- Требование к наличию JUnit-тестов является обязательным.
- Именуйте переменные, методы, класс и прочее так, чтобы можно было понять назначение элемента. Не используйте сокращения, только если они не общепринятые.

Задача 1. Решить задачу.

Создать класс *Мяч*. Создать класс *Корзина*. Наполнить корзину мячиками. Определить вес мячиков в корзине и количество синих мячиков.  
Для модульного тестирования приложения создать JUnit-тесты.

Задача 2. Создать и запустить приложение из командной строки.

Скомпилировать и запустить приложение, созданное при решении задачи 1 из командной строки.

Задача 3. Создать запускной jar-файл и запустить приложение, созданное при решении задачи 1.

Задача 4. Переопределить методы equals(), hashCode() и toString().

Не пользуясь средствами автогенерации кода, переопределить для класса Book методы equals(), hashCode() и toString().

```
public class Book {
```

```
private String title;  
private String author;  
private int price;  
private static int edition;  
}
```

### 3. НАСЛЕДОВАНИЕ И ИНТЕРФЕЙСЫ

Вопросы в начале темы:

- Попытайтесь сформулировать, как вы понимаете, что такое наследование?
- Подумайте, можете ли вы определить принципиальные различия (не касающиеся синтаксических особенностей определения в языке) между наследованием класса и реализацией интерфейса?
- Как вы думаете, для чего используется наследование классов в Java-программе? Приведите примеры применения наследования и интерфейсов.

#### Наследование

Наследование как понятие ООП крайне многогранно. Чтобы корректного воспринимать концепцию наследования (и интерфейсов), можно применить слово «обобщение». Используя наследование (классов), мы можем обобщать свойства и поведение **типов**, выстраивая иерархии, где производный класс находится в отношении «есть(является)» к базовому классу. На рис. 4 приведен пример иерархии наследования.

Отношение «есть(является)» описывается, например, следующим образом: студент есть человек, ручка является канцелярским товаром.

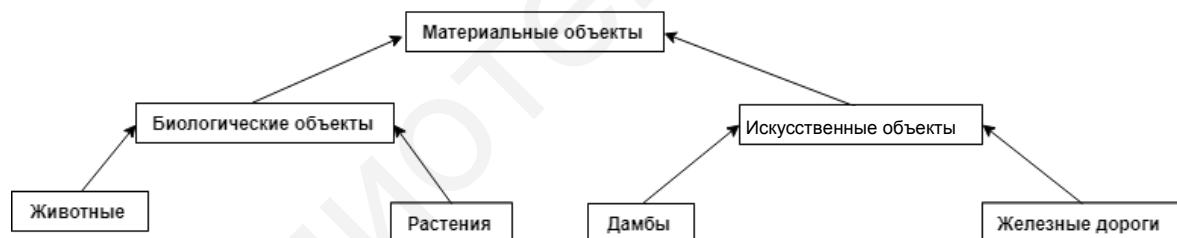


Рис. 4. Пример иерархии наследования

Создать производный класс в Java можно с помощью следующего синтаксиса:

```
[спецификаторы] class имя_класса extends суперкласс [implements  
список_интерфейсов]{  
    /*определение класса*/  
}
```

Базовый класс	Производный класс
<pre>package by.bsuir.unit03.class_syntax_demo; public class Book {}</pre>	<pre>package by.bsuir.unit03.class_syntax_demo; public class EBook extends Book {}</pre>

## Пример наследования от базового класса

Для принятия решения о реализации интерфейса можно использовать отношение, которое лучше всего определяется фразой «умеет делать»: классу А следует реализовать интерфейс Х, если объект класса А умеет делать то, что навязывает ему интерфейс Х: ручка, например, умеет (может) писать.

Определить и реализовать интерфейс в Java можно с помощью следующего синтаксиса:

```
[спецификаторы] interface имя_интерфейса extends  
имя_базового_интерфейса {  
    /*объявление интерфейса*/  
}
```

Определение интерфейса	Реализация интерфейса
<pre>package by.bsuir.unit03.interface_syntax_demo; <b>public interface</b> BookService { }</pre>	<pre>package by.bsuir.unit03.interface_syntax_demo; <b>public class</b> BookServiceImpl <b>implements</b> BookService { }</pre>

Интерфейсы могут наследовать другие интерфейсы, создавая иерархии интерфейсов. Самой известной иерархией интерфейсов в Java является иерархия интерфейсов Collection Framework. Также в Java допустимо только одиночное наследование, а реализуемых интерфейсов в классе может быть несколько.

```
public class HashSet<E>  
    extends AbstractSet<E>  
    implements Set<E>,  
        Cloneable,  
        java.io.Serializable {  
//Определение класса HashSet (Java 1.7)
```



Часть иерархии интерфейсов Collection Framework

На начальных этапах применения обобщения в коде вам может быть трудно определить, что использовать: наследование или интерфейс. Например, «ручка является пишущим предметом» и «ручка умеет писать» являются определенно правильными рассуждениями. Взаимоотношение типов в вашем коде будет зависеть от того, как вы видите и насколько знаете предметную область задачи. В любом случае, даже если после размышлений вы не определились, что применить, воспользуйтесь универсальным советом – в любом непонятном случае сначала попытайтесь использовать интерфейс.

## Связь типов ссылок и типов объектов

Объектная переменная базового типа может ссылаться на объекты как базового, так и производного классов (такая возможность относится к реализации концепции полиморфизма).

## Вызов конструкторов при наследовании

При создании объекта (без нарушений принципов ООП) необходимо вызвать один из конструкторов класса для начальной инициализации свойств объекта. Свойства же производного класса объявлены в целой цепочке классов, и каждый класс из этой цепочки наследования обладает собственным конструктором – и для корректного создания объекта производного класса необходимо обеспечить правильный порядок вызовов конструкторов в цепочке наследования.

При создании объектов производного класса конструктор производного класса вызывает соответствующий конструктор базового класса с помощью ключевого слова **super(параметры)**. Вызов конструктора базового класса из конструктора производного должен быть произведен в первой строке конструктора производного класса. Если конструктор производного класса явно не вызывает конструктор базового, то происходит вызов конструктора по умолчанию базового класса, в этом случае в базовом классе должен быть определен конструктор по умолчанию.

<pre>public class Book {     private String title;     private long price;      public Book() {     }      public Book(String title,</pre>	<pre>public class ProgrammerBook extends Book{     private Language language;      public ProgrammerBook() {         super();     }      public ProgrammerBook(String title, long</pre>
--	---

<pre> <b>long</b> price) {     <b>this.title</b> = title;     <b>this.price</b> = price; } // другие методы } </pre>	<pre> price, Language language) {     super(title, price);     <b>this.language</b> = language; } // другие методы } <b>enum</b> Language{ } </pre>
--	---

## Инициализация статических или полей экземпляра класса при наследовании

При создании объекта производного класса вызываются сначала статические блоки базового класса, а затем производного при условии, что до этого они вызваны не были. Если какие-либо статические блоки класса отработали к моменту создания объекта – их вызов пропускается. Инициализаторы всех статических полей и статические блоки инициализации выполняются в порядке их перечисления в объявлении класса.

Инициализаторы всех полей и блоки инициализации выполняются в порядке их перечисления в объявлении класса перед выполнением конструктора сначала для базового класса, затем для производного.

<pre> <b>public class</b> Book {     <b>private static long</b> countOfBooks;     <b>private String</b> title;     <b>private long</b> price;     <b>static</b> {         countOfBooks = 0;     }     {         title = "";         price = 0;     }     <b>public Book()</b> {     }     <b>public Book(String title, long price)</b> {     } } </pre>	<pre> <b>public class</b> ProgrammerBook <b>extends</b> Book{     <b>private static long</b> countOfProgrammerBooks = 0;     <b>private Language</b> language;     {         language = language.<b>UNKNOWN</b>;     }     <b>public ProgrammerBook()</b> {         super();     }     <b>public ProgrammerBook(String title, long price, Language language)</b> {         super(title, price);         <b>this.language</b> = language;     } } // другие методы </pre>
---	--

<pre>         this.title = title;         this.price = price;     }      // другие методы } </pre>	<pre>     }  <b>enum</b> Language{ UNKNOWN } </pre>
--	---

## Переопределение методов

Переопределение методов является одним из важнейших концептов, реализующих принцип полиморфизма. Для правильного применения данной возможности следует знать технические аспекты переопределения методов в Java и разбираться в философии этого вопроса. Для того чтобы иметь возможность переопределить метод в производном классе, необходимо соблюдение следующих условий:

- наследование его из базового класса;
- атрибуты наследуемого метода должны позволять переопределение;
- метод должен быть доступен для прямого вызова по имени из производного класса.

Для переопределения метода в производном классе необходимо повторить объявление метода из базового класса и дать ему новую реализацию.

<pre> <b>public class</b> Baker {     <b>public</b> Pie cookPie() {         System.out.println("Пекарь печет пирог как умеет");         <b>return new</b> Pie();     } }  <b>public class</b> Pie {} </pre>	<pre> <b>public class</b> ChiefBaker <b>extends</b> Baker{      <b>public</b> Pie cookPie() {         System.out.println("Шеф- пекарь печет умопомрачительно вкусный пирог");         <b>return new</b> Pie();     } } </pre>
---	---

При переопределении метода можно изменить значение возвращаемого типа на производный от используемого в методе базового класса. Таким образом, если метод возвращает значение примитивного типа, то при переопределении метода изменить его нельзя. Также, переопределяя метод, можно расширить область его видимости (изменить атрибут доступа на более широкий).

```
public class ChiefBaker extends Baker{  
  
    public SuperPie cookPie() {  
        System.out.println("Шеф-пекарь печет умопомрачительно  
вкусный пирог");  
        return new SuperPie();  
    }  
}  
  
public class SuperPie extends Pie {  
}
```

Автоматический выбор нужного метода во время выполнения программы называется динамическим связыванием (dynamic binding). Для статических методов в Java полиморфизм неприменим.

## Перегрузка методов при наследовании

Методы с одинаковыми именами, но с различающимися списками параметров и возвращаемыми значениями могут находиться в разных классах одной цепочки наследования и также будут являться перегруженными. Статические методы перегружаются нестатическими, нестатические методы перегружаются статическими.

## Предотвращение переопределения и предотвращение наследования

Чтобы предотвратить переопределение методов, их необходимо объявить терминальными с помощью ключевого слова **final**.

```
public class Book {  
    public final int getPrice(){  
        return price;  
    }...  
}  
public class ProgrammerBook extends Book{  
    ...  
    public final int getPrice(){ // error  
        return price;  
    } ...  
}
```

Классы, объявленные как терминальные, нельзя расширить. Объявить терминальный класс можно следующим образом:

```
public final class Book {}  
public class ProgrammerBook extends Book{} // error
```

Если класс объявлен терминальным, то это не значит, что его поля стали константными.

## Переопределение методов equals, hashCode и toString для производного класса

Для переопределения контрактных методов класса Object в производных классах (не наследуемых напрямую от класса Object) для сравнения его базовой составляющей следует вызывать соответствующие методы базовых классов.

<pre>public class ProgrammerBook ex- tends Book{      private Language language;      public ProgrammerBook() {      }      @Override     public int hashCode() {         final int prime = 31;         int result = <u>super.hashCode()</u>;         result = prime * result + ((lan- guage == null) ? 0 :                                 lan- guage.hashCode());         return result;     } }</pre>	<pre>@Override public boolean equals(Object obj) {     if (this == obj)         return true;     if (!<u>super.equals(obj)</u>)         return false;     if (getClass() != obj.getClass())         return false;     ProgrammerBook other = (Pro- grammerBook) obj;     if (language != other.language)         return false;     return true; } }</pre>
--	---

## Приведение типов при наследовании

На основе описания классов компилятор проверяет, сужает или расширяет возможности класса программист, объявляющий переменную. Если переменной суперкласса присваивается объект подкласса, возможности класса сужаются, и компилятор без проблем позволяет программисту сделать это. Если, наоборот, объект суперкласса присваивается переменной подкласса, возможности класса расширяются, поэтому программист должен подтвердить это с помощью обозначения, предназначенного для приведения типов, указав в скобках имя подкласса (subclass).

```

class Book {
}
class ProgrammerBook extends Book {
}

public class BookInspector {

    public static void main(String[] args) {
        Book book = new ProgrammerBook();
        ProgrammerBook progrBook = new ProgrammerBook();

        Book goodBook = progrBook;
        ProgrammerBook goodProgrBook = (ProgrammerBook) book;

        Book simpleBook = new Book();
        ProgrammerBook simpleProgrBook = (ProgrammerBook) simple-
Book;// error
    }
}

```

При недопустимом преобразовании типов при выполнении программы система обнаружит несоответствие и возбудит исключительную ситуацию. Если ее не перехватить, то работа программы будет остановлена.

Перед приведением типов следует проверить его на корректность. Делается это с помощью оператора **instanceof**.

```

Book simpleBook = ...;
ProgrammerBook simpleProgrBook = ...;

if (simpleBook instanceof ProgrammerBook){
    simpleProgrBook = (ProgrammerBook)simpleBook;
}

```

Компилятор не позволит выполнить некорректное приведение типов. Например, приведение типов

```
Date dt = (Date)SimpleBook;
```

приведет к ошибке на стадии компиляции, поскольку класс **Date** не является подклассом класса **Book**.

## Абстрактные методы и классы

Часто при проектировании иерархии классов верхние классы иерархии становятся все более и более абстрактными, так что реализовывать некоторые методы в них не имеет никакого смысла. Однако удалить их из класса нельзя, т. к. при дальнейшем использовании базовых объектных ссылок на объекты производных классов необходим доступ к переопределенным методам, а он возможен только при наличии в них метода с такой же сигнатурой, как в базовом классе. В таком случае метод следует объявлять абстрактным.

В классе, где метод объявляется абстрактным, его реализация не требуется. Если в классе есть абстрактные методы, то класс следует объявить абстрактным. Абстрактные классы и методы объявляются с ключевым словом **abstract**. При расширении абстрактного класса все его абстрактные методы необходимо определить или подкласс также объявить абстрактным. Нельзя создавать объекты абстрактных классов, однако можно объявлять объектные переменные.

```
abstract class Course {  
    public abstract String getInformation();  
}  
  
class BaseCourse extends Course {  
    public String getInformation() {  
        return "Base course";  
    }  
}  
  
class OptionalCourse extends Course {  
    public String getInformation() {  
        return "Optional course";  
    }  
}  
  
public class CourseInspector {  
    public static void main(String[] args) {  
        Course course1 = new BaseCourse();  
        Course course2 = new OptionalCourse();  
        System.out.println(course1.getInformation());  
        System.out.println(course2.getInformation());  
    }  
}
```

## Интерфейсы. Определение интерфейса

**Интерфейсы** в Java применяются для добавления к классам новых возможностей, которых нет и не может быть в базовых классах. Интерфейсы говорят о том, что класс может делать, но не говорят, как он должен это делать. Интерфейс только гарантирует (определяет контракт), какие методы должен выполнять класс, но как класс выполняет контракт, интерфейс контролировать не может.

*Объявление интерфейса имеет вид:*

```
[спецификаторы] interface имя_интерфейса
    extends имя_базового_интерфейса {
        /*объявление интерфейса*/
    }
```

Поля интерфейса по умолчанию являются **final static**. Все методы по умолчанию открыты (**public**).

```
public interface Square {
    double PI = 3.1415926;
    double square();
}
```

**Реализация интерфейса** происходит в классе с помощью ключевого слова **implements**. Если реализуемых интерфейсов несколько, то они перечисляются через запятую. Интерфейс считается реализованным, когда в классе и/или в его суперклассе реализованы все методы интерфейса.

```
public class Quadrate implements Square {
    private int a;

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

    public double square() {
        return a * a;
    }

    public void print() {
        System.out.println("Square box: " + square());
    }
}
```

```
public class Circle implements Square {
    private int r;
    public Circle(int r) { this.r = r; }

    public double square() {
        return r * r * Square.PI;
    }

    public void print() {
        System.out.println("Square circle: " + square());
    }
}

public class Rectangle implements Square {
    private int a, b;

    public Rectangle(int a, int b) {
        this.a = a;
        this.b = b;
    }

    public double square() {
        return a * b;
    }

    public void print() {
        System.out.println("Square rectangle: " + square());
    }
}

public class SquareInspector {
    public static void main(String[] args) {
        Quadrate box = new Quadrate(4);
        Rectangle rectangle = new Rectangle(2, 3);
        Circle circle = new Circle(3);
        box.print();
        rectangle.print();
        circle.print();
        System.out.println("Box: " + box.square());
        System.out.println("Rectangle: " + rectangle.square());
        System.out.println("Circle: " + circle.square());
    }
}
```

## Свойства интерфейсов

- С помощью оператора **new** нельзя создать экземпляр интерфейса.
- Можно объявлять интерфейсные ссылки.
- Интерфейсные ссылки должны ссылать на объекты классов, реализующих данный интерфейс.
- Через интерфейсную ссылку можно вызвать только методы, определенные в интерфейсе.

```
public class InterfaceProperties {
```

```
    public static void main(String[] args) {
        Quadrate box = new Quadrate(4);
        //box = new Square(); // ERROR
        Square square;
        square = box;
        box.print();
        System.out.println("Box: "+square.square());
        // square.print() // ERROR
        if (box instanceof Square) {
            System.out.println("box implements square");
        }
    }
}
```

- С помощью оператора **instanceof** можно проверять, реализует ли объект определенный интерфейс.
- Если класс не полностью реализует интерфейс, то он должен быть объявлен как **abstract**.

```
public abstract class Ellipse implements Square {
}
```

- Интерфейс может быть расширен при помощи наследования от другого интерфейса, синтаксис в этом случае аналогичен синтаксису наследования классов.

```
interface Collection<E>{  
}
```

```
public interface List<E> extends Collection<E> {  
}
```

## Вложенные интерфейсы

Интерфейсы можно объявить членом (вложить) другого класса или интерфейса.

Когда вложенный интерфейс используется вне области вложения, то он используется вместе с именем класса или интерфейса.

```
public interface Map<K, V> {  
  
    interface Entry<K, V> {  
  
    }  
}
```

## Клонирование объектов. Интерфейс Cloneable

Для создания нового объекта с таким же состоянием используется клонирование объекта. Метод **clone()** класса **Object** объявлен с атрибутом доступа **protected**. Клонирование объекта можно реализовать, имплементировав интерфейс **Cloneable** и реализовав копирование состояний полей и агрегированных объектов.

Интерфейс **Cloneable** не содержит методов, относится к помеченным (**tagged**) интерфейсам, а его реализация гарантирует, что метод **clone()** класса **Object** возвратит точную копию вызвавшего его объекта с воспроизведением значений всех его полей. В противном случае метод генерирует исключение **CloneNotSupportedException**.

```
package java.lang;  
public interface Cloneable {  
  
}  
  
import java.util.Date;  
public class Department implements Cloneable {  
    private Integer name;  
    private Date date = new Date();  
  
    public Object clone() throws CloneNotSupportedException {  
        Department obj = null;  
  
        obj = (Department) super.clone();  
        if (null != this.date) {  
            obj.date = (Date) this.date.clone();  
        }  
    }  
}
```

```

        return obj;
    }
}

import java.util.ArrayList;
import java.util.List;

class Faculty implements Cloneable {
    private String name;
    private int numberDepartments;
    private List<Department> departmentList;

    public Object clone() throws CloneNotSupportedException {
        Faculty obj = null;

        obj = (Faculty) super.clone();
        if (null != this.departmentList) {
            ArrayList<Department> tempList =
                new Ar-
                rayList<Department>(this.departmentList.size());
            for (Department listElem : this.departmentList) {
                tempList.add((Department) listElem.clone());
            }
            obj.departmentList = tempList;
        }
        return obj;
    }
}

```

## Сравнение объектов. Интерфейс Comparable

Метод **sort(...)** класса **Arrays** позволяет упорядочивать массив, переданный ему в качестве параметра. Для элементарных типов правила определения больше/меньше известны.

```

import java.util.Arrays;
public class SortArray {
    public static void main(String[] args) {
        int[] mas = { 3, 6, 5, 1, 2, 9, 8 };
        printArray(mas);
        Arrays.sort(mas);
        printArray(mas);
    }
    public static void printArray(int[] ar) {

```

```

        for (int i : ar) System.out.print(i + " ");
        System.out.println();
    }
}

```

**Естественный порядок сортировки (natural sort order)** – естественный и реализованный по умолчанию (реализацией метода **compareTo** интерфейса **java.lang.Comparable**) способ сравнения двух экземпляров одного класса.

- **int compareTo(E other)** – сравнивает **this**-объект с **other** и возвращает отрицательное значение если **this<other**, 0 – если они равны и положительное значение если **this>other**.

```
public interface Comparable<T> { int compareTo (T other); }
```

Метод **compareTo** должен выполнять следующие условия:

- **sgn(x.compareTo(y)) == -sgn(y.compareTo(x));**
- если **x.compareTo(y)** выбрасывает исключение, то и **y.compareTo(x)** должен выбрасывать то же исключение;
- если **x.compareTo(y)>0** и **y.compareTo(z)>0**, тогда **x.compareTo(z)>0**;
- если **x.compareTo(y)==0** и **x.compareTo(z)==0**, то и **y.compareTo(z)==0**;
- **x.compareTo(y)==0** тогда и только тогда, когда **x.equals(y)** ; (правило рекомендуемо, но не обязательно).

```

public class Person implements Comparable<Person> {
    private String firstName;
    private String lastName;
    private int age;
    public Person(String firstName, String lastName, int age) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }
    public int getAge() {
        return age;
    }
    public int compareTo(Person anotherPerson) {
        int anotherPersonAge = anotherPerson.getAge();
        return this.age - anotherPersonAge;
    }
}

```

## Сравнение объектов. Интерфейс Comparator

При реализации интерфейса **Comparator<T>** существует возможность сортировки списка объектов конкретного типа по правилам, определенным для этого типа. Для этого необходимо реализовать метод **int compare(T ob1, T ob2)**, принимающий в качестве параметров два объекта, для которых должно быть определено возвращаемое целое значение, знак которого и определяет правило сортировки.

Интерфейс **java.util.Comparator** содержит два метода:

- **int compare(T o1, T o2)** – сравнение, аналогичное **compareTo**;
- **boolean equals(Object obj)** – **true** если **obj** это **Comparator** и у него такой же принцип сравнения.

```
public abstract class GeometricObject {  
    public abstract double getArea();  
}  
  
public class RectangleGO extends GeometricObject {  
    private double sideA;  
    private double sideB;  
  
    public RectangleGO(double a, double b) {  
        sideA = a;  
        sideB = b;  
    }  
  
    @Override  
    public double getArea() {  
        return sideA * sideB;  
    }  
}  
  
public class CircleGO extends GeometricObject {  
    private double radius;  
  
    public CircleGO(double r) {  
        radius = r;  
    }  
  
    @Override  
    public double getArea() {  
        // TODO Auto-generated method stub  
        return 2 * 3.14 * radius * radius;  
    }  
}
```

```

import java.util.Comparator;

public class GeometricObjectComparator
    implements Comparator<GeometricObject> {
    public int compare(GeometricObject o1, GeometricObject o2) {
        double area1 = o1.getArea();
        double area2 = o2.getArea();
        if (area1 < area2) {
            return -1;
        } else if (area1 == area2) {
            return 0;
        } else {
            return 1;
        }
    }
}

```

```

import java.util.Comparator;
import java.util.Set;
import java.util.TreeSet;

public class TreeSetWithComparator {
    public static void main(String[] args) {
        Comparator<GeometricObject> comparator = new GeometricObjectComparator();
        Set<GeometricObject> set = new TreeSet<GeometricObject>(comparator);
        set.add(new RectangleGO(4, 5));
        set.add(new CircleGO(40));
        set.add(new CircleGO(40));
        set.add(new RectangleGO(4, 1));
        System.out.println("A sorted set of geometric objects");
        for (GeometricObject elements : set) {
            System.out.println("area = " + elements.getArea());
        }
    }
}

```

## Параметризованные классы. Определение параметризованного класса

С помощью шаблонов можно создавать **параметризованные** (родовые, generic) **классы и методы**, что позволяет использовать более строгую типизацию. Пример класса-шаблона с двумя параметрами:

```
public class Message<T1, T2> {  
    T1 id;  
    T2 name;  
}
```

T1, T2 – фиктивные типы, которые используются при объявлении атрибутов класса. Компилятор заменит все фиктивные типы на реальные и создаст соответствующий им объект. Объект класса **Message** можно создать, например, следующим образом:

```
Message <Integer, String> ob = new Message <Integer, String>();
```

```
public class Optional <T> {  
    private T value;  
    public Optional() {  
    }  
    public Optional(T value) {  
        this.value = value;  
    }  
    public T getValue() {  
        return value;  
    }  
    public void setValue(T val) {  
        value = val;  
    }  
    public String toString() {  
        if (value == null) return null;  
        return value.getClass().getName() + " " + value;  
    }  
}
```

```
public class OptionalDemo {  
    public static void main(String[] args) {  
        Optional<Integer> ob1 = new Optional<Integer>();  
        ob1.setValue(1);  
  
        // ob1.setValue("2");// ERROR  
        int v1 = ob1.getValue();  
        System.out.println(v1);  
  
        // параметризация типом String  
        Optional<String> ob2 = new Optional<String>("Java");  
        String v2 = ob2.getValue();  
        System.out.println(v2);  
    }  
}
```

```
// ob1 = ob2; //ERROR

Optional ob3 = new Optional();

System.out.println(ob3.getValue());
ob3.setValue("Java SE 6");

System.out.println(ob3.toString());

ob3.setValue(71);
System.out.println(ob3.toString());

ob3.setValue(null);
}

}
```

## Применение **extends** при параметризации

*Объявление generic-типа в виде <T>, несмотря на возможность использовать любой тип в качестве параметра, ограничивает область применения разрабатываемого класса. Переменные такого типа могут вызывать только методы класса Object. Доступ к другим методам ограничивает компилятор, предупреждая возможные варианты возникновения ошибок.*

```
public class OptionalExt<T extends Тип> {
    private T value;
}
```

Чтобы расширить возможности параметризованных членов класса, можно ввести ограничения на используемые типы при помощи следующего объявления класса:

```
public class OptionalExt<T extends Тип> {
    private T value;
}
```

Такая запись говорит о том, что в качестве типа **T** разрешено применять только классы, являющиеся наследниками (производными) класса Тип, и соответственно появляется возможность вызова методов ограничивающих (bound) типов.

## Метасимвол «?»

Если возникает необходимость в метод параметризованного класса одного допустимого типа передать объект этого же класса, но параметризованного другим типом, то при определении метода следует применить метасимвол «?».

**<?>**  
**<? extends Number>**

```
public class Mark<T extends Number> {  
    public T mark;  
  
    public Mark(T value) {  
        mark = value;  
    }  
    public T getMark() {  
        return mark;  
    }  
    public int roundMark() {  
        return Math.round(mark.floatValue());  
    }  
/* вместо */// public boolean sameAny (Mark<T> ob) {  
public boolean sameAny(Mark<?> ob) {  
    return roundMark() == ob.roundMark();  
}  
public boolean same(Mark<T> ob) {  
    return getMark() == ob.getMark();  
}  
}
```

```
public class MarkInspector {  
    public static void main(String[] args) {  
        // Mark<String> ms = new Mark<String>("7"); //ошибка  
        // компиляции  
        Mark<Double> md = new Mark<Double>(71.4D); // 71.5d  
        System.out.println(md.sameAny(md));  
        Mark<Integer> mi = new Mark<Integer>(71);  
        System.out.println(md.sameAny(mi));  
        // md.same (mi); //ошибка компиляции  
        System.out.println(md.roundMark());  
    }  
}
```

Метод **sameAny(Mark<?> ob)** может принимать объекты типа **Mark**, инициализированные любым из допустимых для этого класса типов, в то время как метод с параметром **Mark<T>** мог бы принимать объекты с инициализацией того же типа, что и вызывающий метод объект.

## Использование **extends** с метасимволом «?»

С помощью ссылки **List<? extends T>** невозможно добавлять элементы в коллекцию, так как невозможно гарантировать, что в список добавятся объекты допустимого типа. Гарантируется только чтение объектов типа **T** или его подклассов

## Использование **super** с метасимволом «?»

При чтении из коллекции с помощью ссылки типа **List<? super T>** нельзя гарантировать тип возвращаемого объекта иным, кроме как тип **Object**, т. к. ссылка, параметризованная данным образом может ссылаться на коллекции, параметризованные типом **T** и его базовыми типами. Добавление элементов в коллекцию элементов возможно, элементы должны иметь тип **T** или тип его подклассов.

```
import java.util.ArrayList;
import java.util.List;

class MedicalStaff {
}
class Doctor extends MedicalStaff {
}
class HeadDoctor extends Doctor {
}
class Nurse extends MedicalStaff {
}

public class MetaGenericInspector {
    public static void main(String[] args) {
        List<? extends Doctor> list1 = new ArrayList<MedicalStaff>(); // error
        List<? extends Doctor> list2 = new ArrayList<Doctor>();
        List<? extends Doctor> list3 = new ArrayList<HeadDoctor>();

        List<? super Doctor> list7 = new ArrayList<HeadDoctor>(); // error
        List<? super Doctor> list6 = new ArrayList<Doctor>();
        List<? super Doctor> list5 = new ArrayList<MedicalStaff>();
```

```

List<? super Doctor> list4 = new ArrayList<Object>();

list5.add(new Object()); // error
list5.add(new MedicalStaff()); // error
list5.add(new Doctor());
list5.add(new HeadDoctor());

Object object = list5.get(0);
MedicalStaff medicalDtaff = list5.get(0); // error
Doctor doctor = list5.get(0); // error
HeadDoctor headDoctor = list5.get(0); // error
}

}

```

## Параметризованные методы

Параметризованный (generic) метод определяет базовый набор операций, которые будут применяться к разным типам данных, получаемых методом в качестве параметра.

```

<T extends Тип> Тип method(T arg) {}
<Т> Тип method(T arg) {}

```

Описание типа должно находиться перед возвращаемым типом. Запись первого вида означает, что в метод можно передавать объекты, типы которых являются подклассами класса, указанного после **extends**. Второй способ объявления метода никаких ограничений на передаваемый тип не ставит.

```

public class GenericMethod {
    public static <T extends Number> byte asByte(T num) {
        long n = num.longValue();
        if (n >= -128 && n <= 127) return (byte)n;
        else return 0;
    }
    public static void main(String [] args) {
        System.out.println(asByte(7));
        System.out.println(asByte(new Float("7.f")));
        // System.out.println(asByte(new Character('7'))); // ошибка
    }
}

```

Параметризованные методы применяются, когда необходимо разработать базовый набор операций, который будет работать с различными ти-

пами данных. Описание типа всегда находится перед возвращаемым типом. Параметризованные методы могут размещаться как в обычных, так и в параметризованных классах.

Параметр метода может не иметь никакого отношения к параметру класса. Метасимволы применимы и к generic-методам.

```
public static <T> T  
    copy(List<? super T> dest, List<? extends T> src)  
    {...}
```

Параметризованные методы можно перегружать.

```
import java.util.Date;  
public class GenericMethodOverload {  
    public static <Type> void method(Type obj) {  
        System.out.println("<Type> void method(Type obj)");  
    }  
  
    public static void method(Number obj) {  
        System.out.println("void method(Number obj)");  
    }  
  
    public static void method(Integer obj) {  
        System.out.println("void method(Integer obj)");  
    }  
  
    public static void method(String obj) {  
        System.out.println("void method(String obj)");  
    }  
  
    public static void main(String[] args) {  
        Number number = new Integer(1);  
        Integer integer = new Integer(2);  
        method(number);  
        method(integer);  
        method(23);  
        method("string");  
        method(new Date());  
    }  
}
```

## Ограничения при использовании параметризации

Параметризованные поля *не могут быть статическими, статические методы не могут иметь параметризованные классом поля.*

```

public class GenericRestriction<T> {
    private static T x; // error
    private T y;

    public static <Type> void method(Type obj) {
        T var; // error
        Type typeVar;
    }
}

```

## Перечисления (enums). Синтаксис

- dayOfWeek: SUNDAY, MONDAY, TUESDAY, ...
- month: JAN, FEB, MAR, APR, ...
- gender: MALE, FEMALE
- title: MR, MRS, MS, DR
- appletState: READY, RUNNING, BLOCKED, DEAD

```

public enum Season {
    WINTER, SPRING, SUMMER, FALL
}

```

В отличие от статических констант перечисления предоставляют типизированный, безопасный способ задания фиксированных наборов значений, являются классами специального вида, не могут иметь наследников, сами в свою очередь наследуются от **java.lang.Enum** и реализуют **java.lang.Comparable** (следовательно, могут быть сортированы) и **java.io.Serializable**. Перечисления не могут быть абстрактными и содержать абстрактные методы (кроме случая, когда каждый объект перечисления реализовывает абстрактный метод), но могут реализовывать интерфейсы.

```

public class SeasonInspector {
    public static void main(String[] args) {
        Season season = Season.WINTER;
        System.out.println(season);
        // prints WINTER
        season = Season.valueOf("SPRING");
        // sets season to Season.SPRING
    }
}

```

## Создание объектов перечисления

Экземпляры объектов **перечисления нельзя создать с помощью new**, каждый объект перечисления уникален, создается при загрузке пере-

числения в виртуальную машину, поэтому допустимо сравнение ссылок для объектов перечислений, **можно использовать оператор switch**. Как и обычные классы, объекты перечисления могут реализовывать поведение, содержать вложенные классы. Элементы перечисления по умолчанию **public, static и final**.

```
public enum Day {  
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,  
    SATURDAY;  
  
    public boolean isWeekend() {  
        switch (this) {  
            case SUNDAY:  
            case SATURDAY:  
                return true;  
            default:  
                return false;  
        }  
    }  
}  
  
System.out.println( Day.MONDAY + " isWeekEnd(): " +  
    Day.MONDAY.isWeekend() );
```

## Методы перечисления

Каждый класс перечисления неявно содержит следующие методы:

- **static enumType[] values()** – возвращает массив, содержащий все элементы перечисления в порядке их объявления;
  - **static T valueOf(Class<T> enumType, String arg)** – возвращает элемент перечисления, соответствующий передаваемому типу и значению передаваемой строки;
  - **static enumType valueOf(String arg)** – возвращает элемент перечисления, соответствующий значению передаваемой строки;
- Статические методы **valueOf** выбрасывают исключение **IllegalArgumentException**, если в них передается элемент с неправильным именем.

Каждый класс перечисления неявно содержит следующие методы:

- **int ordinal()** – возвращает позицию элемента перечисления.
- **String toString()**
- **boolean equals(Object other)**

```
public enum Shape {
```

```
    RECTANGLE("red"), TRIANGLE("green"), CIRCLE("blue");
```

```

String color;

Shape(String color){
    this.color = color;
}

public String getColor(){
    return color;
}

public class ShapeInspector {

    public static void main(String[] args) {
        double x = 2, y = 3;
        Shape[] arr = Shape.values();
        for (Shape sh : arr)
            System.out.println(sh + " " + sh.getColor());
    }
}

```

## **Анонимные классы перечисления**

Отдельные элементы перечисления **могут реализовывать** свое **собственное поведение**.

```

public enum Direction {
    FORWARD(1.0) {
        public Direction opposite() {
            return BACKWARD;
        }
    },
    BACKWARD(2.0) {
        public Direction opposite() {
            return FORWARD;
        }
    };
}

private double ratio;

```

```

Direction(double r) {
    ratio = r;
}

```

```

public double getRatio() {
    return ratio;
}

public static Direction byRatio(double r) {
    if (r == 1.0)
        return FORWARD;
    else if (r == 2.0)
        return BACKWARD;
    else
        throw new IllegalArgumentException();
}

```

## Сравнение переменных перечисления

На равенство переменные перечислимого типа можно сравнить с помощью операции `==` в операторе **if** или с помощью оператора **switch**.

```

enum Faculty {
    MMF, FPMI, GEO
}

```

```

public class SwitchWithEnum {
    public static void main(String[] args) {
        Faculty current;
        current = Faculty.GEO;
        switch (current) {
            case GEO:
                System.out.print(current);
                break;
            case MMF:
                System.out.print(current);
                break;
            // case LAW : System.out.print(current); //ошибка компиляции!
            default:
                System.out.print("вне case: " + current);
        }
    }
}

```

## Классы внутри классов

В Java можно объявлять классы внутри других классов и даже внутри методов. Они делятся на внутренние нестатические, вложенные статические и анонимные классы. Такая возможность используется, если класс более нигде не используется, кроме как в том, в который он вложен. Более того, использование внутренних классов позволяет создавать простые и понятные программы, управляющие событиями.

### Внутренние (inner) классы

Методы внутреннего класса имеют прямой доступ ко всем полям и методам внешнего класса.

```
import java.util.Date;
public class Outer1 {
    private String str;
    Date date;

    Outer1() {
        str = "string in outer";    date = new Date();
    }
    class Inner {
        public void method() {
            System.out.println(str);
            System.out.println(date.getTime());
        }
    }
}
```

Доступ к элементам внутреннего класса возможен только из внешнего класса через объект внутреннего класса.

```
import java.util.Date;
public class Outer2 {
    private Inner inner;
    private String str;
    private Date date;

    Outer2() {
        str = "string in outer";
        date = new Date();
        inner = new Inner();
    }
}
```

```
class Inner {  
    public void method() {  
        System.out.println(str);  
        System.out.println(date.getTime());  
    }  
}  
  
public void callMethodInInner() {  
    inner.method();  
}  
}
```

Внутренние классы не могут содержать **static**-поляй, кроме **final static**.

```
import java.util.Date;  
public class Outer3 {  
    private Inner inner;  
    private String str;  
    private Date date;  
  
    class Inner {  
        private int i;  
        public static int static_pole; // ERROR  
        public final static int pubfsi_pole = 22;  
        private final static int prfsi_polr = 33;  
  
        public void method() {  
            System.out.println(str);  
            System.out.println(date.getTime());  
        }  
    }  
  
    public void callMethodInInner() {  
        inner.method();  
    }  
}
```

Доступ к таким полям можно получить извне класса, используя конструкцию:

**имя\_внешнего\_класса.имя\_внутреннего\_класса.  
имя\_статической\_переменной**

```
Outer outer = new Outer();
System.out.println(Outer.Inner.pubfsi_pole);
```

Доступ к переменной типа **final static** возможен во внешнем классе через имя внутреннего класса.

```
public class Outer5 {
    Inner inner;
    Outer5() {
        inner = new Inner();
    }
    class Inner {
        public final static int pubfsi_pole = 22;
        private final static int prfsi_polr = 33;
    }
    public void callMethodInInner() {
        System.out.println(Inner.prfsi_polr);
        System.out.println(Inner.pubfsi_pole);
    }
}
```

Внутренние классы могут быть производными от других классов, могут быть базовыми (в пределах внешнего класса) и могут реализовывать интерфейсы.

Если необходимо создать объект внутреннего класса где-нибудь, кроме метода внешнего класса, то нужно определить тип объекта как

**имя\_внешнего\_класса.имя\_внутреннего\_класса**

Объект в этом случае создается по правилу:

**ссылка\_на\_внешний\_объект.new констру-
тор\_внутреннего\_класса([параметры]);**

```
Outer.Inner1 obj1 = new Outer().new Inner1();
Outer.Inner2 obj2 = new Outer().new Inner2();
```

```
obj1.print();
obj2.print();
```

Внутренний класс может быть объявлен внутри метода или логического блока внешнего класса; видимость класса регулируется видимостью того блока, в котором он объявлен; однако класс сохраняет доступ ко всем полям и методам внешнего класса, а также константам, объявленным в текущем блоке кода.

```
public class Outer6 {  
    public void method() {  
        final int x = 3;  
        class Inner1 {  
            void print() {  
                System.out.println("Inner1");  
                System.out.println("x=" + x);  
            }  
        }  
        Inner1 obj = new Inner1();  obj.print();  
    }  
    public static void main(String[] args) {  
        Outer6 out = new Outer6();      out.method();  
    }  
}
```

Локальные внутренние классы не объявляются с помощью модификаторов доступа.

```
public class Outer7 {  
    public void method() {  
        public class Inner1 {  
        } // ОШИБКА  
    }  
}
```

Ссылка на внешний класс имеет вид

**имя\_внешнего\_класса.this**

```
public class Outer8 {  
    int count = 0;  
  
    class InnerClass {  
        int count = 10000;  
  
        public void display() {  
            System.out.println("Outer: " + Outer8.this.count);  
            System.out.println("Inner: " + count);  
        }  
    }  
}
```

## Статические (nested) классы

Статический вложенный класс для доступа к нестатическим членам и методам внешнего класса должен создавать объект внешнего класса.

```
public class Outer9 {  
    private int x = 3;  
    static class Inner1 {  
        public void method() {  
            Outer9 out = new Outer9();  
            System.out.println("out.x=" + out.x);  
        }  
    }  
}
```

Вложенный класс имеет доступ к статическим полям и методам внешнего класса.

```
public class Outer10 {  
    private int x = 3;  
    private static int y = 4;  
    public static void main(String[] args) {  
        Inner1 in = new Inner1();  
        in.method();  
    }  
    public void meth() {  
        Inner1 in = new Inner1();  
        in.method();  
    }  
    static class Inner1 {  
        public void method() {  
            System.out.println("y=" + y);  
            // System.out.println("x=" + x); // ERROR  
            Outer10 out = new Outer10();  
            System.out.println("out.x=" + out.x);  
        }  
    }  
}
```

Статический метод вложенного класса вызывается при указании полного относительного пути к нему.

```

public class Outer11 {
    public static void main(String[] args) {
        Inner1.method();
    }

    public void meth() {
        Inner1.method();
    }

    static class Inner1 {
        public static void method() {
            System.out.println("inner static method");
        }
    }
}

public class Outer12 {
    public static void main(String[] args) {
        Outer11.Inner1.method();
    }
}

```

Подкласс вложенного класса не наследует возможность доступа к членам внешнего класса, которым наделен его суперкласс.

```

public class Outer13 {
    private static int x = 10;

    public static void main(String[] args) {
        Inner1.method();
    }

    public void meth() {
        Inner1.method();
    }

    static class Inner1 {
        public static void method() {
            System.out.println("inner1 outer.x=" + x);
        }
    }
}

```

```
public class Outer14 extends Outer13.Inner1{
    public static void main(String[] args) {
    }

    public void outer2Method() {
        // System.out.println("x="+x); // ERROR
    }
}
```

Класс, вложенный в интерфейс, статический по умолчанию.

```
public interface InterfaceWithClass {
    int x = 10;

    class InnerInInterface {
        public void meth() {
            System.out.println("x=" + x);
        }
    }
}
```

Вложенный класс может быть базовым, производным, реализующим интерфейсы.

```
public class ExtendNested extends Outer15.Inner {
}

class Outer15 {
    static class Inner {
    }
    static class Inner2 extends Inner {
    }
    class Inner3 extends Inner {
    }
}
```

## Анонимные (anonymous) классы

Анонимный класс расширяет другой класс или реализует внешний интерфейс при объявлении одного-единственного объекта; остальным будет соответствовать реализация, определенная в самом классе.

```

import java.util.Date;
public class AnonymEx {
    public void ex(){
        Date d=new Date(){
            @Override
            public String toString(){
                return "new version toString method";
            }
        };
        System.out.println(d);
    }
}

```

Объявление анонимного класса выполняется одновременно с созданием его объекта с помощью операции **new**. Анонимные классы допускают вложенность друг в друга. Конструкторы анонимных классов ни определить, ни переопределить нельзя.

```

public class SimpleClass {
    public void print() {
        System.out.println("This is Print() in SimpleClass");
    }
}

public class AnonymInspector {
    public void printSecond() {
        SimpleClass myCl = new SimpleClass() {
            public void print() {
                System.out.println("!!!!!!");    newMeth();
            }
            public void newMeth() {
                System.out.println("New method");
            }
        };
        SimpleClass myCl2 = new SimpleClass();

        myCl.print(); // myCl.newMeth(); // Error
        myCl2.print();
    }
    public static void main(String[] args) {
        AnonymInspector myS = new AnonymInspector();
        myS.printSecond();
    }
}

```

Объявление анонимного класса в перечислении отличается от простого анонимного класса, поскольку инициализация всех элементов происходит при первом обращении к типу.

```
enum Color {
    Red(1),
    Green(2),
    Blue(3) {
        int getNumColor() {
            return 222;
        }
    };
    Color(int _num) {
        num_color = _num;
    }

    int getNumColor() {
        return num_color;
    }

    private int num_color;
}

public class EmunWithAnonym {
    public static void main(String[] args) {
        Color color;
        color = Color.Red;
        System.out.println(color.getNumColor());
        color = Color.Blue;
        System.out.println(color.getNumColor());
        color = Color.Green;
        System.out.println(color.getNumColor());
    }
}
```

## Вопросы для закрепления знаний

- Укажите, как вызываются конструкторы при создании объекта производного класса? Что в конструкторе класса делает оператор `super()`? Возможно ли в одном конструкторе использовать операторы `super()` и `this()`?

- Объясните, как вы понимаете утверждения: «ссылка базового класса может ссылаться на объекты своих производных типов» и «объект производного класса может быть использован везде, где ожидается объект его базового типа». Верно ли обратное и почему?
- Что такое переопределение методов? Как вы думаете, зачем они нужны? Можно ли менять возвращаемый тип при переопределении методов? Можно ли менять атрибуты доступа при переопределении методов? Можно ли переопределить методы в рамках одного класса?
- Определите правило вызова переопределенных методов. Можно ли статические методы переопределить нестатическими и наоборот?
- Какие свойства имеют финальные методы и финальные классы? Как вы думаете, зачем их использовать?
- Укажите правила приведения типов при наследовании. Напишите примеры явного и неявного преобразования ссылочных типов. Объясните, какие ошибки могут возникать при явном преобразовании ссылочных типов.
- Что такое перечисления в Java? Как объявить перечисление? Чем являются элементы перечислений? Кто и когда создает экземпляры перечислений? Могут ли перечисления реализовывать интерфейсы или содержать абстрактные методы? Могут ли перечисления содержать статические методы?
- Можно ли самостоятельно создать экземпляр перечисления, ссылку типа перечисления? Как определить, что в двух переменных содержится один и тот же элемент перечисления и почему именно так?
- Что такое параметризованные классы? Для чего они необходимы? Приведите пример параметризованного класса и пример создания объекта параметризованного класса. Объясните, ссылки какого типа могут ссылаться на объекты параметризованных классов. Можно ли создать объект, параметризовав его примитивным типом данных?
- Какие ограничения на вызов методов существуют у параметризованных полей? Как эти ограничения снимает использование при параметризации ключевого слова extends?
- Как параметризуются статические методы, как определяется конкретный тип параметризованного метода? Можно ли методы экземпляра класса параметризовать отдельно от параметра класса, и если да, то как тогда определять тип параметра?
- Что такое wildcard? Приведите пример его использования.
- Для чего используется параметризация <? extends Type>, <? super Type>?

## Лабораторная работа №1

Общие требования к коду задач, входящих в лабораторную работу:

- Напишите код, решающий задачи, приведенные ниже.
- Обязательно используйте пакеты.
- Именуйте переменные, методы, класс и прочее так, чтобы можно было понять назначение элемента. Не используйте сокращения, только если они не общепринятые.

Задача 1. Переопределить методы equals(), hashCode() и toString().

Не пользуясь средствами автогенерации кода, переопределить для класса ProgrammerBook методы equals(), hashCode() и toString().

```
public class ProgrammerBook extends Book{  
    private String language;  
    private int level;  
}
```

Задача 2. Переопределить метод clone.

Не пользуясь средствами автогенерации кода, переопределить для класса Book из задачи 1 метод clone().

Задача 3. Реализовать интерфейс Comparable.

Добавить в класс Book из задачи 1 поле isbn. Реализовать в классе Book интерфейс Comparable так, чтобы книги приобрели сортировку по умолчанию согласно номеру isbn. Написать тесты JUnit, проверяющие корректность сортировки.

Задача 4. Реализовать интерфейс Comparator.

Реализовать для класса Book из задачи 1 компараторы, позволяющие сортировать книги по следующим параметрам: по названию; по названию, а потом по автору; по автору, а потом по названию; по автору, названию и цене. Напишите тесты JUnit, проверяющие корректность сортировок.

## Лабораторная работа №2

Основные требования к лабораторной работе:

- При реализации приложения примените шаблон MVC.
- При написании приложения используйте Java Code Convention.

Важно:

- Приложение должно быть объектно-, а не процедурно-ориентированным!

Общие пояснения к практическому заданию:

- Корректно спроектируйте и реализуйте предметную область задачи.
- Для создания объектов из иерархии классов используйте шаблон Factory.
- Реализуйте проверку данных, вводимых пользователем, но не на стороне клиента.
- Особое условие: для проверки корректности переданных данных применяйте регулярные выражения.
- Меню выбора действия пользователем можно не реализовывать, используйте заглушку.
- Особое условие: переопределите, где необходимо, методы `toString()`, `equals()` и `hashCode()`.

### Варианты задания

**Цветочная композиция.** Реализовать приложение, позволяющее создавать цветочные композиции (объект, представляющий собой цветочную композицию). Составляющими цветочной композиции являются цветы и упаковка.

**Подарки.** Реализовать приложение, позволяющее создавать подарки (объект, представляющий собой подарок). Составляющими целого подарка являются сладости и упаковка.

**Сборник литературных произведений.** Реализовать приложение, позволяющее создавать сборники произведений (объект, представляющий собой сборник). Составляющими сборника являются литературные произведения (стихи, проза и т. д.) и форма сборника (журнал, книга и т. д.).

**Подписка.** Реализовать приложение, позволяющее создавать подписки на группу печатных изданий (объект, представляющий собой подписку). Составляющими подписки являются печатные издания и варианты подписки.

**Игрушки для Нового года.** Реализовать приложение, позволяющее создавать наборы украшений к Новому году (объект, представляющий собой набор украшений). Составляющими новогоднего набора являются елочные украшения и упаковка.

## Лабораторная работа №3

Задание: создать однопоточное консольное приложение «Учет книг в домашней библиотеке».

Общие требования к заданию:

- Система учитывает книги как в электронном, так и в бумажном варианте.
- Существующие роли: пользователь, администратор.
- Пользователь может просматривать книги в каталоге книг, осуществлять поиск книг в каталоге.
- Администратор может модифицировать каталог.
- При добавлении описания книги в каталог оповещение о ней рассыпается на e-mail всем пользователям.
- При просмотре каталога желательно реализовать постраничный просмотр.
- Пользователь может предложить добавить книгу в библиотеку, переслав ее администратору на e-mail.
- Каталог книг хранится в текстовом файле.
- Данные аутентификации пользователей хранятся в текстовом файле. Пароль не хранится в открытом виде.

Требования к коду лабораторной работы:

- При реализации приложения придерживайтесь layered architecture. (При несоблюдении этого требования лабораторная защита не будет!)
- Для доступа к данным обязательна реализация слоя DAO.
- Код должен быть документирован (javadoc), документация сгенерирована.
- При написании приложения обязательно использовать Java Code Convention.

## 4. LAYERED ARCHITECTURE – «РАССЛОЕНИЕ» СИСТЕМЫ

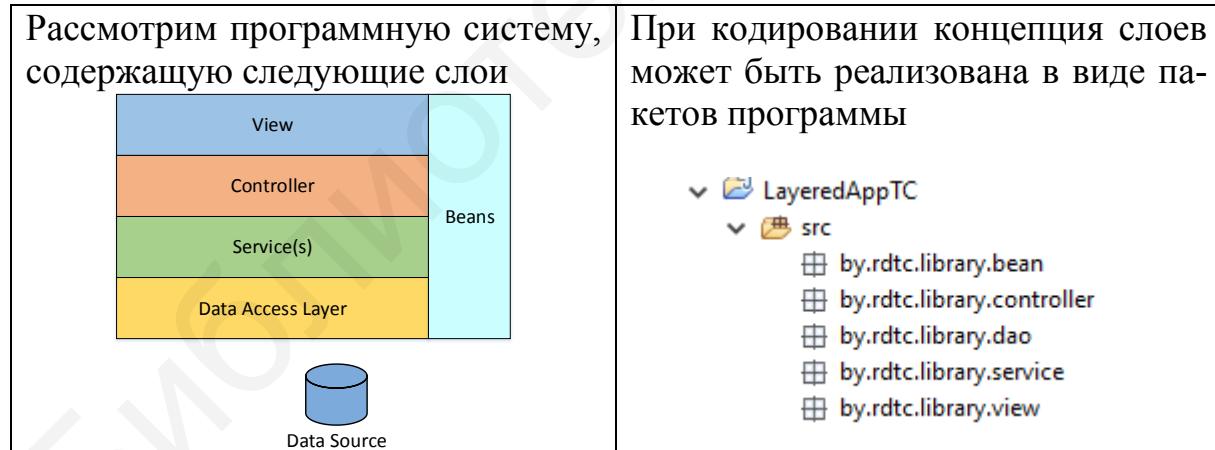
Вопросы в начале темы:

- Как вы думаете , зачем нужны архитектурные паттерны проектирования?
- Дайте объяснение паттерну MVC.

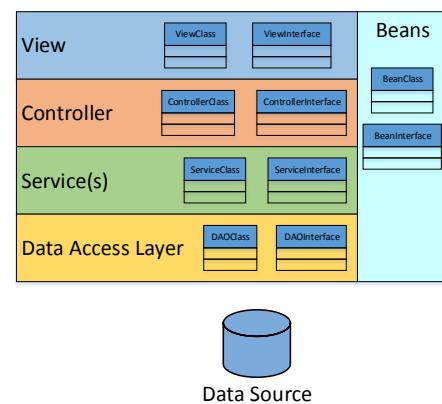
«Расслоение» системы – концепция слоев (layers) – одна из общеупотребительных моделей, используемых разработчиками программного обеспечения для разделения сложных систем на более простые части. Описывая систему в терминах архитектурных слоев, удобно воспринимать составляющие ее подсистемы в виде «слоеного пирога».

Разделение системы на слои представляет целый ряд преимуществ:

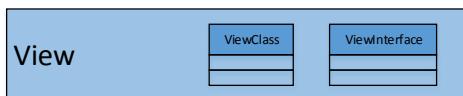
- отдельный слой можно воспринимать как единое самодостаточное целое, не особенно заботясь о наличии других слоев;
- можно выбирать альтернативную реализацию базовых слоев;
- зависимость между слоями можно свести к минимуму;
- каждый слой является удачным кандидатом на стандартизацию;
- созданный слой может служить основой для нескольких различных слоев более высокого уровня;
- слои способны удачно инкапсулировать многое (но не все);
- наличие избыточных слоев нередко снижает производительность системы.



Внутри каждого слоя располагаются артефакты (классы, интерфейсы и др.), обеспечивающие выполнение слоем своих непосредственных функций.



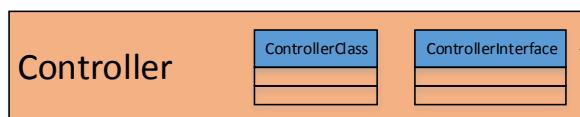
## View Layer



Слой View (Вид) отвечает за внешнее взаимодействие приложения или модуля.

Например, Вид может отвечать за взаимодействие (ввод данных и отображение информации) с пользователем.

## Controller Layer



Слой Controller (Контроллер) является слоем управления и содержит компоненты, отвечающие за управление приложением.

Компоненты этого слоя не имеют права выполнять какую-либо логику. Если проводить аналогию, то слой Контроллер можно ассоциировать с директором (или целым отделом директоров) фирмы, выпускающей, например, окна и двери. Контроллер решает, может ли приложение выполнить пришедший запрос, разрешено ли тому, кто прислал запрос, его выполнять, и **кто** будет этот запрос выполнять. Так, директор фирмы решает, будет ли фирма браться за этот заказ и какому мастеру получить его выполнение. В такой ассоциации слой Вид можно представлять **заказчиком**, который запрашивает у Контроллера какие-либо действия и ожидает от Контроллера какого-либо результата. На основании полученного от контроллера ответа Вид может сгенерировать следующий запрос.

## Service Layer

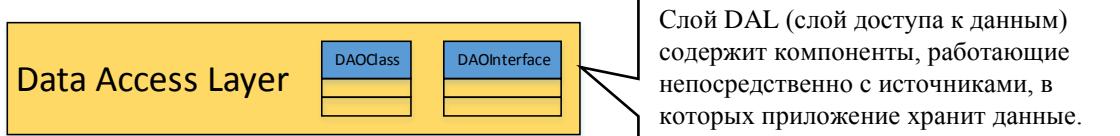


Слой Service (Сервис, Логика) реализует всю функциональность приложения.

Именно ради действий, реализованных на слое Сервис, мы и используем приложение. Если приложение должно прогнозировать котировки акций на следующей неделе, то алгоритмы прогноза будут реализованы здесь. Если необходимо в системе зарегистрировать пользователя, то код регистрации будет находиться непосредственно на этом слое.

Если говорить про фирму «окна – двери», то на слое сервисов у этой фирмы будут находиться конкретные бригады, выполняющие конкретные заказы. Необходимо окно – это делает бригада номер один, дверь входная – бригада номер два и т. д.

## Data Access Layer

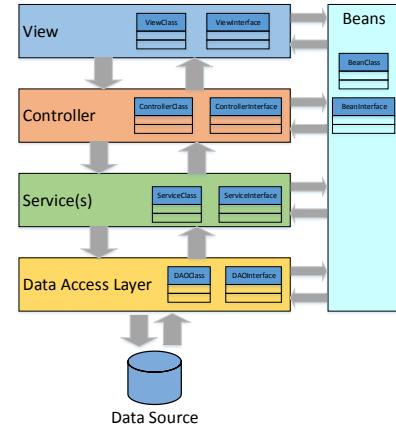


Слой DAL (слой доступа к данным) содержит компоненты, работающие непосредственно с источниками, в которых приложение хранит данные.

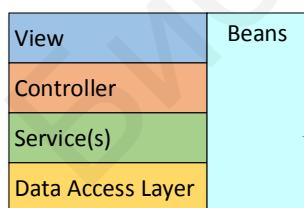
99,9 % приложений используют различные системы хранения данных. Самой распространенной на текущий момент является база данных. Код, «общающийся» с базой данных (и с любыми другими источниками), зачастую очень специфичен, а нередко и громоздок. Слой DAL решает две задачи. С одной стороны, он закрывает от слоя Сервисов знания о том, как непосредственно надо работать с источником; если при реализации сервисов надо получить информацию из источника или сохранить ее, код слоя сервисов будет обращаться к слою DAL с «просьбой» это сделать. С другой стороны, введение слоя доступа к данным позволяет модифицировать и менять источник, даже если это требует изменения программного кода для доступа к нему; слой Сервисов в таком случае будет в безопасности и не потребует модификаций.

*Чаще вместо аббревиатуры DAL вы будете слышать аббревиатуру DAO (Data Access Object). DAO – паттерн проектирования, многочисленные вариации которого очень часто используются при реализации слоя доступа к данным.*

А вот для фирмы «окна – двери» слой доступа к данным будет реализован служащими, которые должны доставлять для бригад сервисов рабочие материалы и инструменты и убирать не- нужное на склад, ведь работающей бригаде все равно в какой партии доставили нужное им по типу дерево или стекло.



## Beans Layer



Слой Beans (Слой компонентов данных) содержит компоненты, способные хранить и передавать требуемую информацию.

Чтобы сказать одному компоненту программы, что ему необходимо сделать, или чтобы получить результат работы компонента, приложение должно уметь обмениваться (передавать) данными. Однако таким компонентам запрещено обрабатывать данные, они могут только хранить и передавать.

## Взаимодействие слоев

Взаимодействие слоев в приложении строго регламентировано. Слой может общаться (т. е. передавать и получать данные) только от слоев, расположенных в архитектуре непосредственно выше и ниже искомого. Если перевести это на язык программирования, то код, расположенный в слое контроллера, не имеет права обращаться к коду, расположенному на слое DAL, и, соответственно, получать от этого кода какие-то данные напрямую; контроллер даже не должен знать, что внизу под сервисами что-то существует.

Попытка представить картину расслоения всех выполняемых задач в реальной жизни может привести, например, к ситуации, представленной на рис. 5.

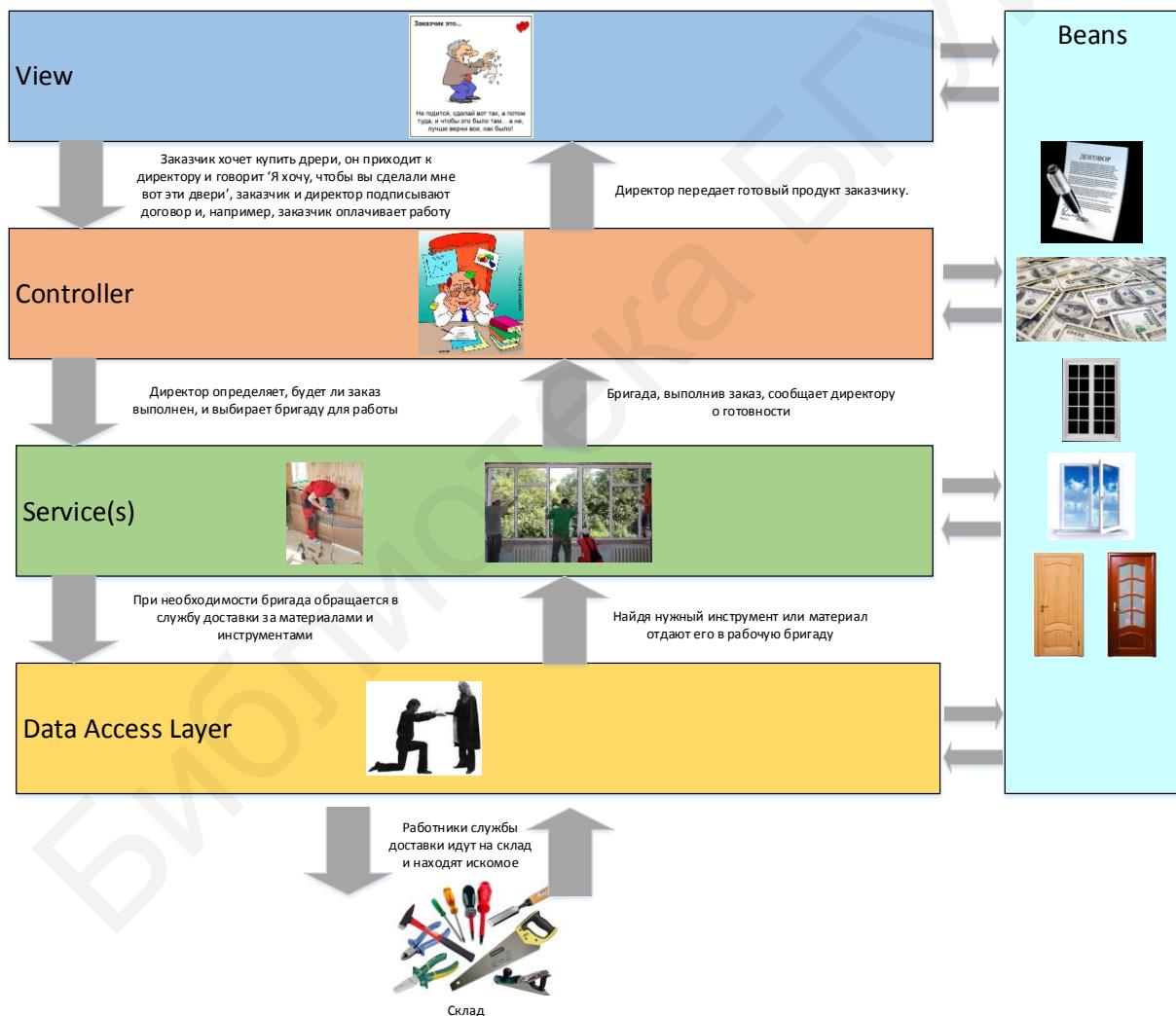


Рис. 5. Пример взаимодействия слоев

## DAO (code)

Теперь переведем архитектурные рассуждения в код и разработаем шаблон для предметной темы «Библиотека».

Сначала устроим мозговой штурм и определим запросы, которые в приложении будут использоваться для обращения к источнику данных. Причем не важно, какой конкретно источник данных будет использоваться, запросы следует определять как «Потребуется проверить, есть ли логин и пароль пользователя в системе» и т. д.

Множество запросов можно разбить на смысловые группы и оформить их в коде в виде интерфейсов.

Например,

```
package by.rdtc.library.dao;  
  
import by.rdtc.library.bean.User;  
  
public interface UserDAO {  
    void signIn(String login, String password);  
    void registration(User user);  
}
```

```
package by.rdtc.library.dao;  
  
import by.rdtc.library.bean.Book;  
  
public interface BookDAO {  
    void addBook(Book book);  
    void deleteBook(long id-  
Book);  
    void delete(Book book);  
}  
  
▼ └─ by.rdtc.library.dao  
    > └─ BookDAO.java  
    > └─ UserDAO.java
```

Также при создании интерфейсов понадобятся bean-классы, объекты которых будут содержать требуемую информацию о пользователе или книге.

```
package by.rdtc.library.bean;  
  
public class User {  
    //stub  
    //надеюсь, вы знаете, какой код тут должен быть написан  
}  
package by.rdtc.library.bean;  
  
public class Book {  
    //stub  
}
```

```
▼ └─ by.rdtc.library.bean  
    > └─ Book.java  
    > └─ User.java
```

Следующим шагом будет написание реализации интерфейсов для слоя data access, причем источник данных в этом случае уже необходимо определить. В нашем случае это будет реляционная база данных.

```
package by.rdtc.library.dao.impl;

import by.rdtc.library.bean.User;
import by.rdtc.library.dao.UserDAO;

public class SQLUserDAO implements UserDAO{

    @Override
    public void signIn(String login, String password) {
        // именно в этом методе мы связываемся с базой данных и проверяем корректность логина и пароля
    }

    @Override
    public void registration(User user) {
    }
}
```

---

```
package by.rdtc.library.dao.impl;

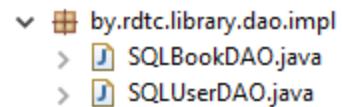
import by.rdtc.library.bean.Book;
import by.rdtc.library.dao.BookDAO;

public class SQLBookDAO implements BookDAO{

    @Override
    public void addBook(Book book) {
    }

    @Override
    public void deleteBook(long idBook) {
    }

    @Override
    public void delete(Book book) {
    }
}
```



Теперь создадим фабрику, которая будет предлагать получить ссылки на объект, класс которого реализует требуемый DAO-интерфейс.

```
import by.rdtc.library.dao.BookDAO;
import by.rdtc.library.dao.UserDAO;
import by.rdtc.library.dao.impl.SQLBookDAO;
import by.rdtc.library.dao.impl.SQLUserDAO;

public final class DAOFactory {
    private static final DAOFactory instance = new DAOFactory();

    private final BookDAO sqlBookImpl = new SQLBookDAO();
    private final UserDAO sqlUserImpl = new SQLUserDAO();

    private DAOFactory(){}
    public static DAOFactory getInstance(){
        return instance;
    }

    public BookDAO getBookDAO(){
        return sqlBookImpl;
    }

    public UserDAO getUserDAO(){
        return sqlUserImpl;
    }
}
```

Класс DAOFactory представляет собой singleton, у которого две задачи. Первая – это закрыть от пользователя слоя конкретную реализацию. Вторая – не создавать каждый раз новые объекты типа SQLBookDAO и SQLUserDAO, т. к. многократное создание этих объектов является грубой ошибкой.

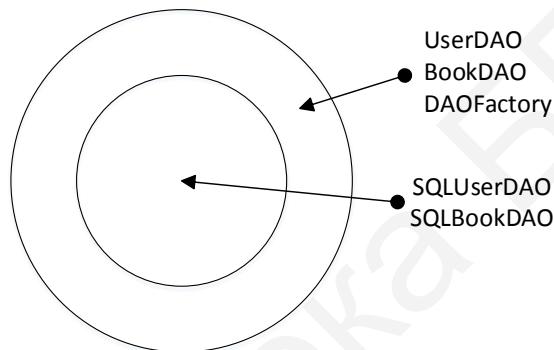
Теперь, если понадобится добавить книгу в библиотеку, то необходимо будет использовать следующий код (тогда при внесении изменения в одно место кода не будет возникать необходимость менять все остальное):

```
DAOFactor daoObjectFactory = DAOFactory.getInstance();
BookDAO bookDAO = daoObjectFactory.getBookDAO();
bookDAO.addBook(book);
```

Реализацию слоя можно представить в виде круговых секций.



Обращаясь к слою, правильно обращаться только к его интерфейсной части. Для приведенного слоя DAO размещение компонентов в кругах выглядит следующим образом.



*Внимание. Представленная реализация – это не единственная возможность написать слой DAO. Существуют и другие решения; главное, чтобы в этих решениях не нарушались основные концепты расслоения системы.*

### Service (code)

Каждый слой может взаимодействовать только со слоем, лежащим непосредственно сверху и снизу. Так слой сервисов может взаимодействовать со слоем доступа к данным и контроллером.

Напишем слои сервисов аналогично структуре слоя DAO. Сначала создадим интерфейсы.

```
package by.rdtc.library.service;  
import by.rdtc.library.bean.User;  
  
public interface ClientService {  
    void singIn(String login, String password);  
    void singOut(String login);
```

```
    void registration(User user);  
}  
  
package by.rdtc.library.service;  
  
import by.rdtc.library.bean.Book;  
  
public interface LibraryService {  
    void addNewBook(Book book);  
    void addEditedBook(Book book);  
}
```

Потом реализуем интерфейсы.

```
package by.rdtc.library.service.impl;  
  
import by.rdtc.library.bean.User;  
import by.rdtc.library.dao.UserDAO;  
  
import by.rdtc.library.dao.factory.DAOFactory;  
import by.rdtc.library.service.ClientService;  
  
public class ClientServiceImpl implements ClientService {  
  
    @Override  
    public void singIn(String login, String password) {  
        // проверяем параметры  
  
        // реализуем функционал логинации пользователя в системе  
        DAOFactory daoObjectFactory = DAOFactory.getInstance();  
        UserDAO userDAO = daoObjectFactory.getUserDAO();  
        userDAO.signIn(login, password);  
        //....  
    }  
  
    @Override  
    public void singOut(String login) {  
    }  
  
    @Override  
    public void registration(User user) {  
    }  
}
```

```
package by.rdtc.library.service.impl;

import by.rdtc.library.bean.Book;
import by.rdtc.library.service.LibraryService;

public class LibraryServiceImpl implements LibraryService{

    @Override
    public void addNewBook(Book book) { }

    @Override
    public void addEditedBook(Book book) { }

}
```

Каждый открытый метод реализации слоя сервисов имеет обязанность проверять входящие параметры (кто бы и где бы до него это не делал)!

Далее предоставим возможность получения доступа к реализации, не открывая имена конкретных классов.

```
package by.rdtc.library.service.factory;

import by.rdtc.library.service.ClientService;
import by.rdtc.library.service.LibraryService;
import by.rdtc.library.service.impl.ClientServiceImpl;
import by.rdtc.library.service.impl.LibraryServiceImpl;

public final class ServiceFactory {
    private static final ServiceFactory instance = new ServiceFactory();

    private final ClientService clientService = new ClientServiceImpl();
    private final LibraryService libraryService = new LibraryServiceImpl();

    private ServiceFactory() {}

    public static ServiceFactory getInstance(){
        return instance;
    }

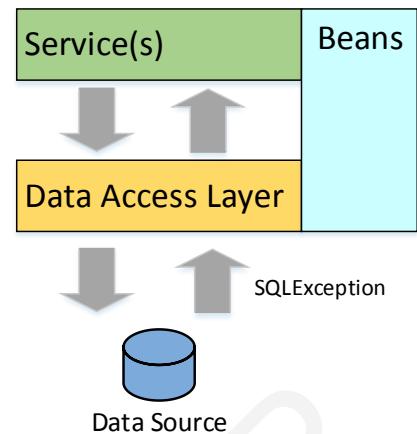
    public ClientService getClientService(){
        return clientService;
    }
}
```

```

public LibraryService getLibraryService(){
    return libraryService;
}

}

```



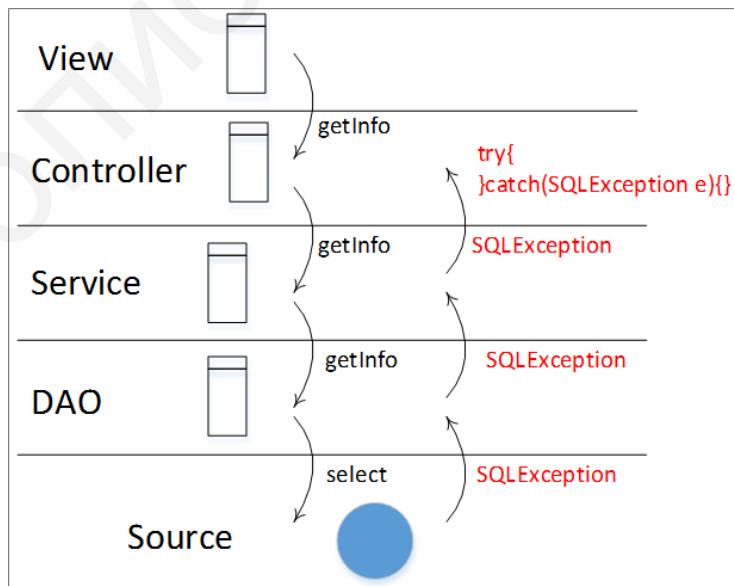
Итак, слой DAO и слой Service у нас есть. Слой Service, обращаясь к DAO, использует только интерфейсную его часть, и не перегружен знанием конкретной реализации, используемой при доступе к данным.

### Обработка исключений

Далее изучим, как обрабатываются исключения, при необходимости их передачи другому слою.

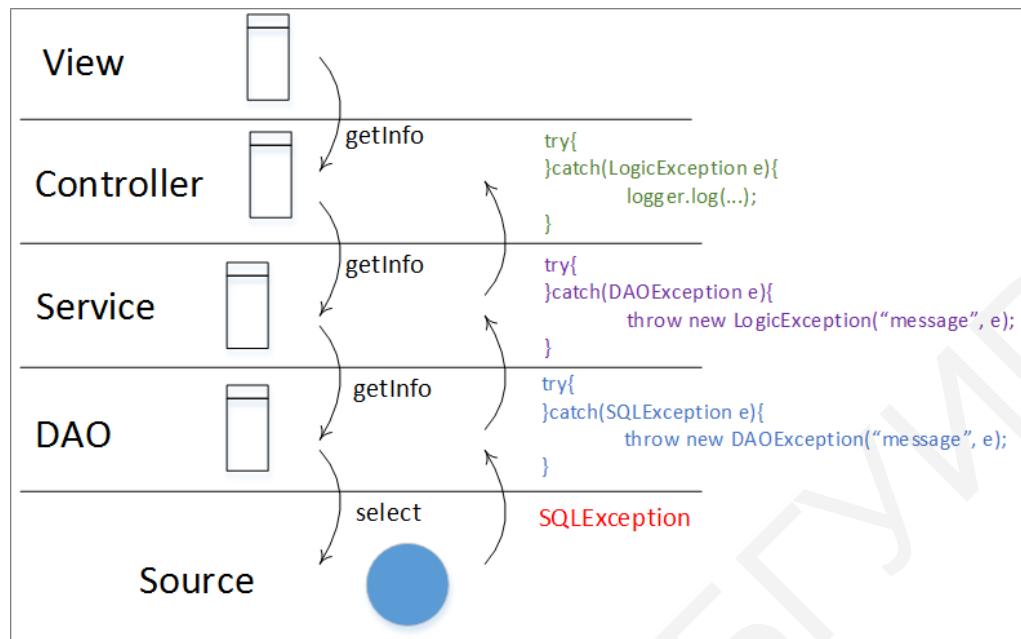
Итак, слой сервисов обратился к слою DAO, слои DAO в свою очередь к базе данных (БД). Однако при выполнении запроса к БД выбросилось исключение SQLException, которое слой DAO не может исправить. Что с ним делать?

Можно отказаться от обработки исключения и отдать его сервисам. Однако это нарушит концепцию слоев приложения, сервисы узнают о чем-то, что характерно для слоя, лежащего ниже, чем DAO.

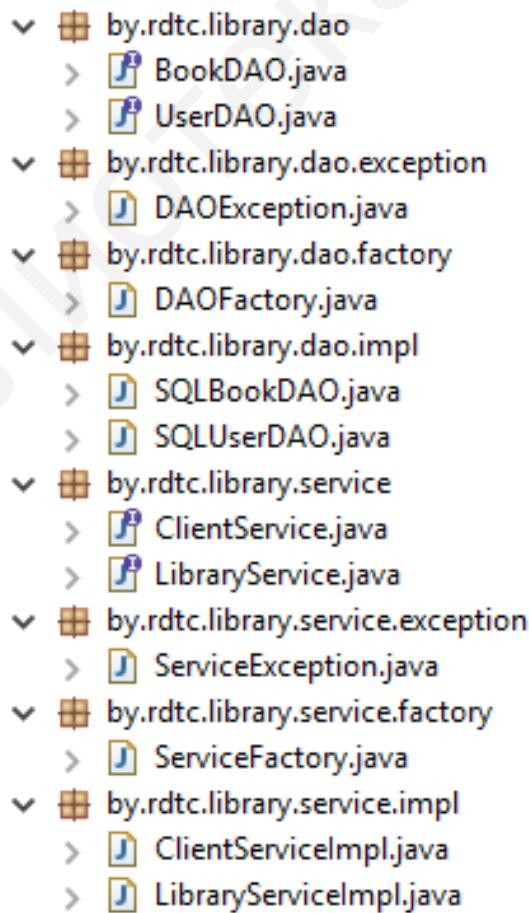


Поэтому для каждого слоя разрабатывается как минимум одно собственное исключение и при необходимости пробросить исключение на

слой выше генерируется исключение именно того слоя, который его и выбрасывает.



Изменим код нашего приложения так, чтобы предоставить возможность корректной обработки исключений.



## *DAO*

```
package by.rdtc.library.dao;

import by.rdtc.library.bean.Book;
import by.rdtc.library.dao.exception.DAOException;

public interface BookDAO {
    void addBook(Book book) throws DAOException;
    void deleteBook(long idBook) throws DAOException;
    void delete(Book book) throws DAOException;
}

package by.rdtc.library.dao;

import by.rdtc.library.bean.User;
import by.rdtc.library.dao.exception.DAOException;

public interface UserDAO {
    void signIn(String login, String password) throws DAOException;
    void registration(User user) throws DAOException;
}

package by.rdtc.library.dao.exception;

public class DAOException extends Exception{
    private static final long serialVersionUID = 1L;

    public DAOException(){
        super();
    }

    public DAOException(String message){
        super(message);
    }

    public DAOException(Exception e){
        super(e);
    }

    public DAOException(String message, Exception e){
        super(message, e);
    }
}
```

```
package by.rdtc.library.dao.factory;

import by.rdtc.library.dao.BookDAO;
import by.rdtc.library.dao.UserDAO;
import by.rdtc.library.dao.impl.SQLBookDAO;
import by.rdtc.library.dao.impl.SQLUserDAO;
public final class DAOFactory {
    private static final DAOFactory instance = new DAOFactory();

    private final BookDAO sqlBookImpl = new SQLBookDAO();
    private final UserDAO sqlUserImpl = new SQLUserDAO();

    private DAOFactory(){}
    public static DAOFactory getInstance(){
        return instance;
    }
    public BookDAO getBookDAO(){
        return sqlBookImpl;
    }
    public UserDAO getUserDAO(){
        return sqlUserImpl;
    }
}

package by.rdtc.library.dao.impl;

import by.rdtc.library.bean.Book;
import by.rdtc.library.dao.BookDAO;
import by.rdtc.library.dao.exception.DAOException;

public class SQLBookDAO implements BookDAO{
    @Override
    public void addBook(Book book) throws DAOException{
    }
    @Override
    public void deleteBook(long idBook) throws DAOException{
    }
}
```

```
    @Override
    public void delete(Book book) throws DAOException{

    }
}

package by.rdtc.library.dao.impl;

import by.rdtc.library.bean.User;
import by.rdtc.library.dao.UserDAO;
import by.rdtc.library.dao.exception.DAOException;

public class SQLUserDAO implements UserDAO{

    @Override
    public void signIn(String login, String password) throws DAOException{
        // именно в этом методе мы связываемся с базой данных и проверяем корректность логина и пароля
    }

    @Override
    public void registration(User user) throws DAOException{
    }
}

package by.rdtc.library.service;

import by.rdtc.library.bean.User;
import by.rdtc.library.service.exception.ServiceException;

public interface ClientService {
    void singIn(String login, char[] password) throws ServiceException;
    void singOut(String login) throws ServiceException;
    void registration(User user) throws ServiceException;
}

package by.rdtc.library.service;

import by.rdtc.library.bean.Book;
import by.rdtc.library.service.exception.ServiceException;
```

```
public interface LibraryService {  
    void addNewBook(Book book) throws ServiceException;  
    void addEditedBook(Book book) throws ServiceException;  
}  
  
package by.rdtc.library.service.exception;  
  
public class ServiceException extends Exception {  
    private static final long serialVersionUID = 1L;  
  
    public ServiceException(){  
        super();  
    }  
  
    public ServiceException(String message){  
        super(message);  
    }  
  
    public ServiceException(Exception e){  
        super(e);  
    }  
  
    public ServiceException(String message, Exception e){  
        super(message, e);  
    }  
}
```

```
package by.rdtc.library.service.factory;  
  
import by.rdtc.library.service.ClientService;  
import by.rdtc.library.service.LibraryService;  
import by.rdtc.library.service.impl.ClientServiceImpl;  
import by.rdtc.library.service.impl.LibraryServiceImpl;  
  
public final class ServiceFactory {  
    private static final ServiceFactory instance = new ServiceFactory();  
  
    private final ClientService clientService = new ClientServiceImpl();  
    private final LibraryService libraryService = new LibraryServiceImpl();  
  
    private ServiceFactory(){ }  
  
    public static ServiceFactory getInstance(){
```

```
    return instance;
}

public ClientService getClientService(){
    return clientService;
}

public LibraryService getLibraryService(){
    return libraryService;
}

}

package by.rdtc.library.service.impl;

import by.rdtc.library.bean.User;
import by.rdtc.library.dao.UserDAO;
import by.rdtc.library.dao.exception.DAOException;
import by.rdtc.library.dao.factory.DAOFactory;
import by.rdtc.library.service.ClientService;
import by.rdtc.library.service.exception.ServiceException;

public class ClientServiceImpl implements ClientService {
    @Override
    public void singIn(String login, String password) throws ServiceException{
        // проверяем параметры
        if(login == null || login.isEmpty()){
            throw new ServiceException("Incorrect login");
        }

        // реализуем функционал логинации пользователя в системе
        try{
            DAOFactory daoObjectFactory = DAOFactory.getInstance();
            UserDAO userDAO = daoObjectFactory.getUserDAO();
            userDAO.signIn(login, password);
        }catch(DAOException e){
            throw new ServiceException(e);
        }
        //....
    }
}
```

```

@Override
public void singOut(String login) throws ServiceException{
}

@Override
public void registration(User user) throws ServiceException{

}
}

package by.rdtc.library.service.impl;

import by.rdtc.library.bean.Book;
import by.rdtc.library.service.LibraryService;
import by.rdtc.library.service.exception.ServiceException;

public class LibraryServiceImpl implements LibraryService{

    @Override
    public void addNewBook(Book book) throws ServiceException{

    }

    @Override
    public void addEditedBook(Book book) throws ServiceException{

    }
}

```

### *Controller (Code)*

Теперь нужно реализовать слой управления. Определим, что данные на слое Controller будут приходить в виде форматированной строки. Ответ клиенту Контроллер также будет осуществлять в виде форматированной строки. Например:

```

package by.rdtc.library.controller;

public class Controller {

    public String executeTask(String request){
        return null;//stub
    }
}

```

Предположим, что название команды будет приходить в начале строки до первого пробела.

Тогда обработка различных команд слоем управления может выглядеть следующим образом:

```
package by.rdtc.library.controller;

public class Controller {
    private final char paramDelimeter = ' ';

    public String executeTask(String request){

        String command;
        command = request.substring(0, re-
quest.indexOf(paramDelimeter));
        command = command.toUpperCase();

        String response = null;
        switch(command){
            case "SIGN_IN":
                // do action (call services and others) and create a response
                break;
            case "ADD_BOOK":
                // do action and create a response
                break;
            default:
                response = "We can't execute this command";
        }

        return response;
    }
}
```

Однако использование оператора **switch** не является хорошей идеей в принципе. Он очень громоздок и тяжело модифицируется. Заменим эту конструкцию паттерном Command.

Сначала создадим интерфейс Command:

```
package by.rdtc.library.controller.command;

public interface Command {
    public String execute(String request);
}
```

Затем создадим реализации этого интерфейса – конкретные команды.

```
package by.rdtc.library.controller.command.impl;

import by.rdtc.library.controller.command.Command;
import by.rdtc.library.service.ClientService;
import by.rdtc.library.service.exception.ServiceException;
import by.rdtc.library.service.factory.ServiceFactory;

public class SingIn implements Command{
    @Override
    public String execute(String request) {
        String login = null;
        String password = null;

        String response = null;

        // get parameters from request and initialize the variables login and
password

        ServiceFactory serviceFactory = ServiceFactory.getInstance();
        ClientService clientService = serviceFactory.getClientService();
        try {
            clientService.singIn(login, password);
            response = "Welcome";
        } catch (ServiceException e) {
            // write log
            response = "Error during login procedure";
        }
        return response;
    }
}

package by.rdtc.library.controller.command.impl;

import by.rdtc.library.controller.command.Command;

public class Register implements Command{
    @Override
    public String execute(String request) {
        // stub
        return null;
    }
}
```

```

package by.rdtc.library.controller.command.impl;
import by.rdtc.library.controller.command.Command;

public class AddBook implements Command {
    @Override
    public String execute(String request) {
        //stub
        return null;
    }
}

```

Реализуем удобный механизм доступа к экземплярам команд (чтобы не раскрывать детали реализации и не множить постоянно одни и те же объекты).

```

package by.rdtc.library.controller;

import java.util.HashMap;
import java.util.Map;

import by.rdtc.library.controller.command.Command;
import by.rdtc.library.controller.command.CommandName;
import by.rdtc.library.controller.command.impl.AddBook;
import by.rdtc.library.controller.command.impl.Register;
import by.rdtc.library.controller.command.impl.SingIn;
import by.rdtc.library.controller.command.impl.WrongRequest;

final class CommandProvider {
    private final Map<CommandName, Command> repository = new
    HashMap<>();

    CommandProvider(){
        repository.put(CommandName.SIGN_IN, new SingIn());
        repository.put(CommandName.REGISTRATION, new Register());
        repository.put(CommandName.ADD_BOOK, new AddBook());
        repository.put(CommandName.WRONG_REQUEST, new WrongRequest());
        //...
    }

    Command getCommand(String name){
        CommandName commandName = null;
        Command command = null;
        try{
            commandName = Command-

```

```

Name.valueOf(name.toUpperCase());
        command = repository.get(commandName);
    }catch(IllegalArgumentException | NullPointerException e){
        //write log
        command = repository.get(CommandName.WRONG_REQUEST);
    }
    return command;
}
}

```

Класс Controller теперь сможет диспетчеризировать множество команд.

```

package by.rdtc.library.controller;

import by.rdtc.library.controller.command.Command;

public final class Controller {
    private final CommandProvider provider = new CommandProvider();

    private final char paramDelimeter = ' ';
    public String executeTask(String request){
        String commandName;
        Command executionCommand;

        commandName = request.substring(0, request.indexOf(paramDelimeter));
        executionCommand = provider.getCommand(commandName);

        String response;
        response = executionCommand.execute(request);

        return response;
    }
}

```

В такой реализации класс Controller выступает в роли front controllerа (в него поступают все запросы к системе), а объекты – команды – уже в роли конкретных команд. Задачей контроллера становится нахождение команды, которая отвечает за управление выполнением конкретного запроса и передачу запроса в эту команду. Задачей же команды является извлечение параметров, приведение их в виду, который запрашивают методы сервисов, обращение к слою сервисов для выполнения запроса, получение

ответа от сервиса и формирование своего ответа – того, который подходит для передачи клиенту (в Вид).

### *View (without code)*

Реализацию слоя View (Вид) рассматривать нет необходимости, т. к. она может быть предназначена и для консольных приложений, и для приложения с графическим интерфейсом. Основное правило – все общение слоя Вид с другими частями приложения идет через экземпляр класса Controller (и его также не надо создавать в нескольких экземплярах). Вид формирует запрос – строку определенного вида, передает ее контроллеру и ожидает ответ. Получив ответ, обрабатывает его в соответствии со своими задачами.

## **ЛИТЕРАТУРА**

1. Эккель, Б. Философия Java / Б. Эккель. – М. : Питер, 2016. – 809 с.
2. Джошуа, Б. Java. Эффективное программирование / Б. Джошуа. – М. : ЛОРИ, 2014. – 220 с.
3. Фаулер, М. Шаблоны корпоративных приложений / М. Фаулер ; пер. с англ. – М. : Изд. дом «Вильямс», 2012. – 544 с.
4. Блинov, И. Н. Java. Методы программирования : учеб.-метод. пособие / И. Н. Блинov, В. С. Романчик. – Минск : Четыре четверти, 2013. – 896 с.

*Учебное издание*

**Смолякова Ольга Георгиевна**

**ОСНОВЫ РАЗРАБОТКИ ВЕБ-ПРИЛОЖЕНИЙ  
НА ЯЗЫКЕ ПРОГРАММИРОВАНИЯ JAVA**

УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ

Редактор *М. А. Зайцева*  
Корректор *Е. Н. Батурчик*

Компьютерная правка, оригинал-макет *Е. Г. Бабичева*

Подписано в печать 16.10.2019. Формат 60×84 1/16. Бумага офсетная. Гарнитура «Таймс». Отпечатано на ризографе. Усл. печ. л. 7,79. Уч.-изд. л. 7,6. Тираж 100 экз. Заказ 54.

Издатель и полиграфическое исполнение: учреждение образования  
«Белорусский государственный университет информатики и радиоэлектроники».

Свидетельство о государственной регистрации издателя, изготовителя,

распространителя печатных изданий №1/238 от 24.03.2014,

№2/113 от 07.04.2014, №3/615 от 07.04.2014.

Ул. П. Бровки, 6, 220013, г. Минск