JAVA.SE.01

JAVA FUNDAMENTALS

Author: **Ihar Blinou**
Oracle Certified Java Instructor
ihar_blinou@epam.com

Сожержание

1. Введение в язык Java
2. Типы данных, переменные, операторы
3. Простейшие классы и объекты
4. Java Beans
5. Массивы
6. Code conventions
7. Параметризованные классы
8. Перечисления
9. Внутренние классы
10. Документирование кода

ВВЕДЕНИЕ В ЯЗЫК JAVA

Введение в язык Java. Язык программирования Java

Java - это *объектно-ориентированный*, *платформенно-независимый* язык программирования, используемый для разработки информационных систем, работающих в сети *Internet*.

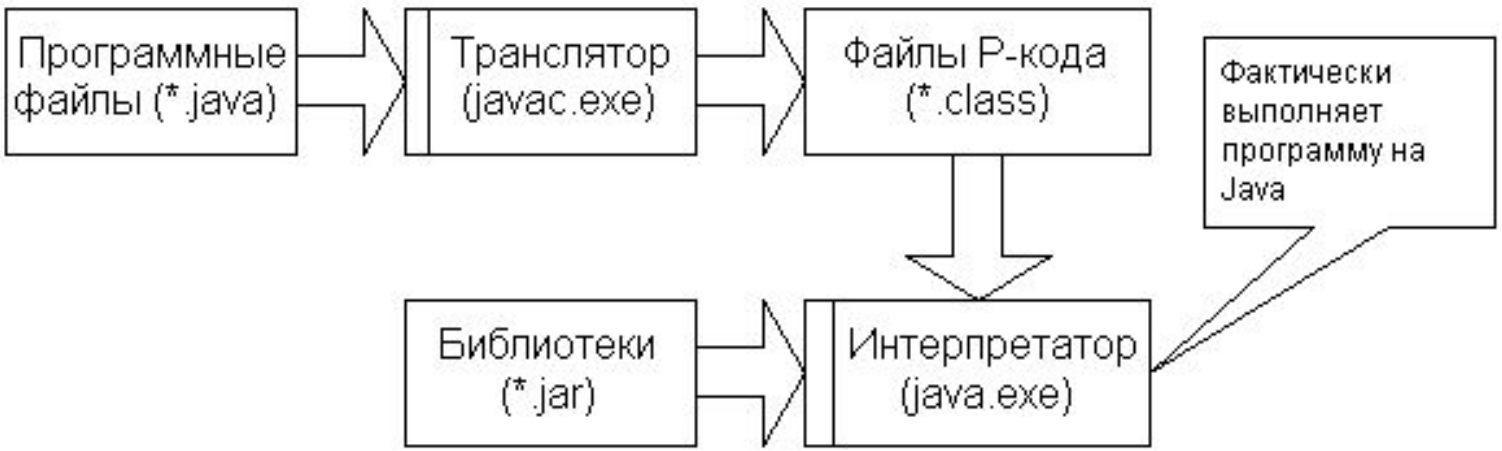
Объектно-ориентированный язык Java, разработанный в Sun Microsystems, предназначен для создания *переносимых* на различные платформы и операционные системы *программ*. Язык Java нашел широкое применение в Интернет-приложениях, добавив на статические и клиентские Web-страницы динамическую графику, улучшив интерфейсы и реализовав вычислительные возможности. Но объектно-ориентированная парадигма и кроссплатформенность привели к тому, что уже буквально через несколько лет после своего создания язык практически покинул клиентские страницы и перебрался на сервера. На стороне клиента его место занял язык JavaScript.

Введение в язык Java. Использование памяти

В Java все объекты программы расположены в **динамической памяти** (heap) и доступны по **объектным ссылкам**, которые в свою очередь хранятся в *стеке*. Это решение исключило непосредственный доступ к памяти, но усложнило работу с элементами массивов.

Необходимо отметить, что объектные ссылки языка Java *содержат информацию о классе* объектов, на которые они ссылаются, так что объектные ссылки - это не указатели, а дескрипторы объектов. Наличие дескрипторов позволяет JVM выполнять проверку совместимости типов на фазе интерпретации кода, генерируя исключение в случае ошибки.

Введение в язык Java. Жизненный цикл программы на Java



Введение в язык Java. Простое линейное приложение. Example 1

```
package _java._se._01._start;

public class First {
    public static void main(String[] args){
        System.out.print("Java ");
        System.out.println("уже здесь!");
    }
}
```

Результат:

```
Java уже
здесь!
```

Введение в язык Java. Простое объектно-ориентированное приложение. Example 2

```
package _java._se._01._start.firstoop;
public class AboutJava {
     public void printReleaseData(){
          System.out.println("Java уже здесь!");
     }
}
```

```
package _java._se._01._start.firstoop;
public class FirstOOPProgram {
     public static void main(String[] args) {
          AboutJava object = new AboutJava();
          object.printReleaseData();
     }
}
```

Результат:

```
Java уже
здесь!
```

Введение в язык Java. Компиляция и запуск приложения из командной строки

Создайте файл Console.java со следующим содержанием

```
package _java._se._01._start;
public class Console {
    public static void main(String[] args) {
        System.out.println("Hello!");
    }
}
```

Скомпилируйте программу командой **`javac.exe Console.java`**

Введение в язык Java. Компиляция и запуск приложения из командной строки

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

Для запуска программы их консоли выполните команду **`java.exe Console`**

Введение в язык Java. Работа с аргументами командной строки

Создайте файл ConsoleArguments.java со следующим содержанием:

```
package _java._se._01._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]);
    }
  }
}
```

Скомпилируйте приложение и запустите его с помощью следующей командной строки **`java.exe CommandArg first second 23 56 23,9`**

Введение в язык Java. Консоль. Простейшие примеры

Взаимодействие с консолью с помощью потока System.in представляет собой один из простейших способов передачи информации в приложение.

В следующем примере рассматривается ввод информации в виде символа из потока ввода, связанного с консолью, и последующего вывода на консоль символа и его числового кода.

Введение в язык Java. Консоль. Простейшие примеры. Example 3

```
package _java._se._01._start;

public class ReadCharRunner {
    public static void main(String[] args) {
         int x;
         try {
        x = System.in.read();
        char c = (char)x;
        System.out.println("Код символа: "+c+" = "+x);
         } catch (java.io.IOException e) {
        e.printStackTrace();
         }
    }
}
```

Результат:

```
v
Код символа: v =
118
```

Введение в язык Java. Консоль. Простейшие примеры. Example 4

```
package _java._se._01._start;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
public class ReadCharRunnerString {
     public static void main(String[] args) {
     InputStreamReader is = new InputStreamReader(System.in);
     BufferedReader bis = new BufferedReader(is);
     try {
     System.out.println("Введите Ваше имя и нажмите <Enter>:" );
          String name = bis.readLine();
          System.out.println("Привет, " + name);
     } catch (IOException e) {
          System.err.print("ошибка ввода " + e);
     }
      }
}
```

Результат:

```
Введите ваше имя и нажмите
<Enter>:
Ivan
Привет, Ivan
```

Введение в язык Java. Консоль. Простейшие примеры. Example 5

```
package _java._se._01._start;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class BRRead {
  public static void main(String[] args)  throws IOException {
    char c = ' ';
    BufferedReader br =  new BufferedReader(new
InputStreamReader(System. in));
    System.out.println("Вводите символы, 'q'-для выхода." );
    do {
    c = (char) br.read();
    System.out.println(c);
    } while (c != 'q');
  }
}
```

Введение в язык Java. Консоль. Простейшие примеры. Example 5

Результат:

```
Вводите символы, 'q'-для
выхода.
abcdef
a
b
c
d
e
f
ghq
g
h
q
```

Введение в язык Java. Консоль. Простейшие примеры. Example 6

```
package _java._se._01._start;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
public class BRReadLines {
 public static void main(String[] args)  throws IOException {
  BufferedReader br =  new BufferedReader(
        new InputStreamReader(System. in));
  String str;
  System.out.println("Вводите строки текста");
  System.out.println("Введите 'стоп' для завершения");
  do {
    str = br.readLine();
    System.out.println(str);
  } while (!str.equalsIgnoreCase("стоп"));
 }
}
```

Введение в язык Java. Консоль. Простейшие примеры. Example 6

Результат:

```
Вводите строки текста
Введите 'стоп' для завершения
first string
first string
second string
second string
стоп
стоп
```

Введение в язык Java. Консоль. Простейшие примеры. Example 7

```
package _java._se._01._start;
import java.util.Scanner;
public class Scan {
    public static void main(String[] args) {
        int i;
        Scanner conin = new Scanner(System.in);
        while (conin.hasNextInt()) {
            i = conin.nextInt();
            System.out.println("i=" + i);
        }
    }
}
```

Результат:

```
23
i=23
56
i=56
s
```

ТИПЫ ДАННЫХ, ПЕРЕМЕННЫЕ, ОПЕРАТОРЫ

Типы данных, переменные, операторы. Примитивные и ссылочные типы

Язык Java является объектно-ориентированным, но существуют типы данных (простые/примитивные), не являющиеся объектами.

- Фактор производительности

Простые типы делятся на 4 группы:

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

Введение в синтаксис языка классов позволяет создавать свои типы, получившие название ссылочных.

Типы данных, переменные, операторы. Примитивные типы

Примитив-ный тип	Размер(бит)	Мин. значение	Макс. значение	Класс-оболочка
boolean	-	-	-	Boolean
char	16	Unicode 0	U2^16-1	Character
byte	8	-128	127	Byte
short	16	-2^15	2^15-1	Short
int	32	-2^31	2^31-1	Integer
long	64	-2^63	2^63-1	Long
float	32	IEEE754	IEEE754	Float
double	64	IEEE754	IEEE754	Double
void	-	-	-	Void

Типы данных, переменные, операторы. Размер типа данных. Значения по умолчанию

- Размер одинаков для всех платформ; за счет этого становится возможной переносимость кода

- Размер boolean неопределен. Указано, что он может принимать значения true или false

Типы данных, переменные, операторы. Размер типа данных. Значения по умолчанию

Неинициализированная явно переменная (член класса) примитивного типа принимает значение в момент создания

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

Типы данных, переменные, операторы. Переменные. Объявление переменных

Характеристики.

- Основное место для хранения данных
- Должны быть явно объявлены
- Каждая переменная имеет тип, идентификатор и область видимости
- Определяются для класса, для экземпляра и внутри метода

Объявление переменных.

- Может быть объявлена в любом месте блока кода
- Должна быть объявлена перед использованием
- Обычно переменные объявляются в начале блока
- Область видимости определяется блоком
- Необходимо инициализировать переменные перед использованием
- Переменные простых типов инициализируются автоматически

Типы данных, переменные, операторы. Переменные. Объявление переменных

Основная форма объявления

тип идентификатор [= значение];

При объявлении переменные могут быть проинициализированы

```
package _java._se._01._types;
public class VariablesExample {
    public static void main(String[] args) {
        int itemsSold = 10;
        float itemCost = 11.0f;
        int i, j, k;
        double interestRate;
    }
}
```

Типы данных, переменные, операторы. Переменные. Объявление переменных

Java не позволяет присваивать переменной значение более длинного типа, если только это не константы. Исключение составляют операторы инкремента, декремента и операторы +=, −=, *=, /=.

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.

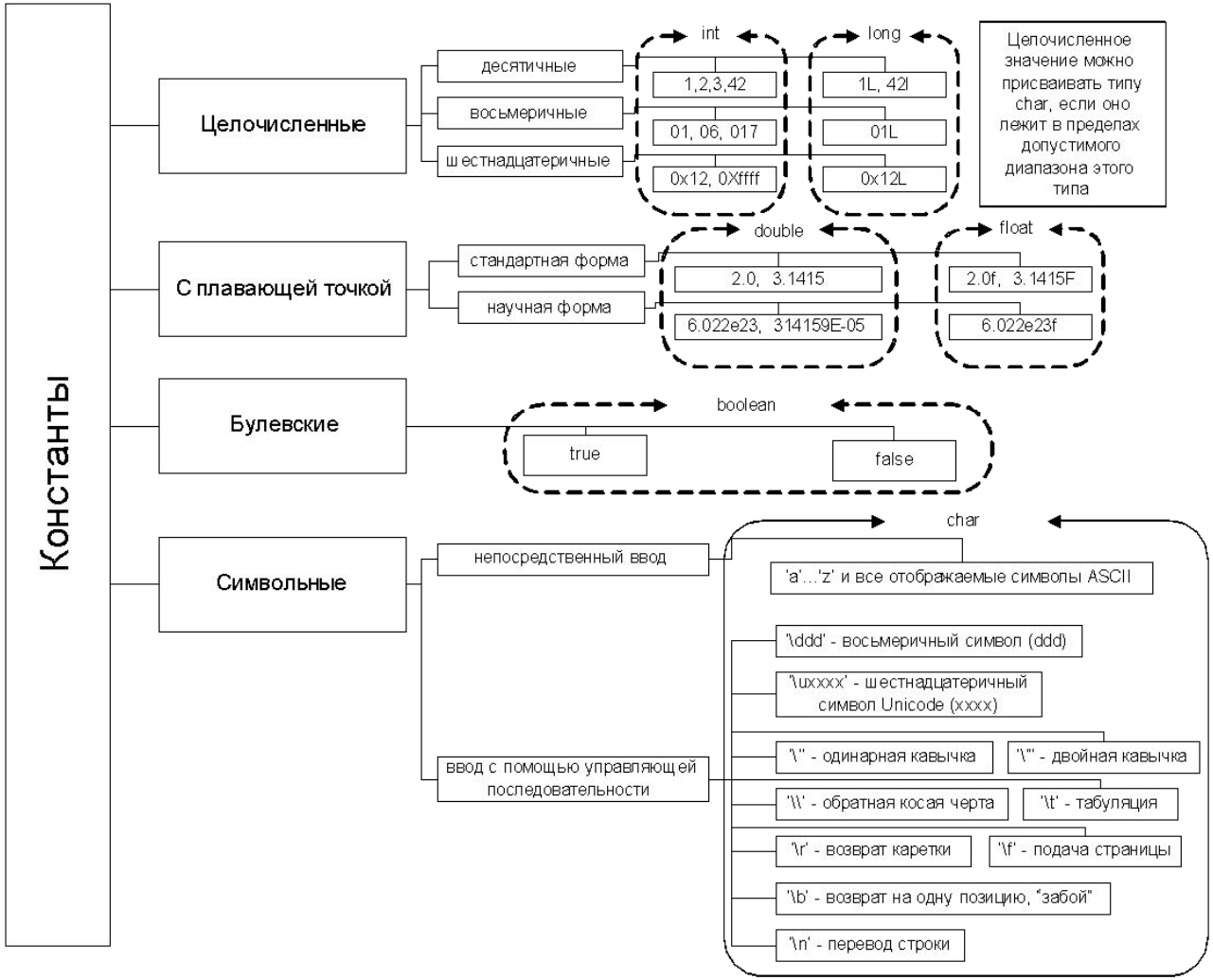
Типы данных, переменные, операторы. Ключевые и зарезервированные языка Java

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

Типы данных, переменные, операторы. Ключевые и зарезервированные языка Java

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

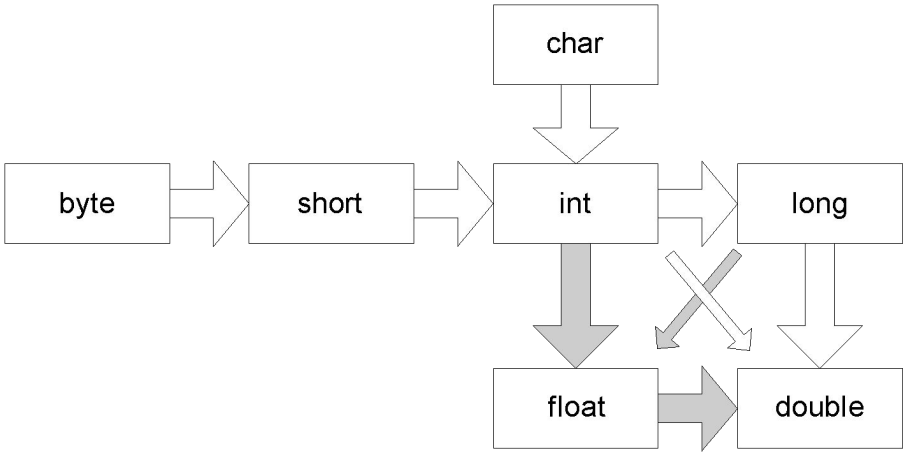
Типы данных, переменные, операторы. Литералы



Типы данных, переменные, операторы. Преобразования типов

Java запрещает смешивать в выражениях величины разных типов, однако при числовых операциях такое часто бывает необходимо. Различают повышающее (разрешенное, неявное) преобразование и понижающее приведение типа.

Повышающее преобразование осуществляется автоматически по следующему правилу. Серыми стрелками обозначены преобразования, при которых может произойти потеря точности.



Типы данных, переменные, операторы. Расширяющее и сужающее преобразование типов

Расширяющее 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 преобразование. . Результирующий тип имеет меньший диапазон значений, чем исходный тип.

```
    long value2 = 1000L;
    int value3 = (int)value2; //обязательно. Иногда это
единственный способ сделать код компилируемым
```

Типы данных, переменные, операторы. Потеря точности при преобразовании типов. Example 8

```
package _java._se._01._types;
public class LoseAccuracy {
public static void main(String[] args) {
     byte b = 10;
     long e1;
     e1 = b;

     long a = 10000000000L;
     int x;
     x = (int)a;
     System.out.println("1 - "+x);

     byte b5 = 50;
     //byte b4 = b5*2;
     byte b4 = (byte)(b5*2);

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

     double d=12.34;
     int x3;
     x3 = (int)d;
     System.out.println("3 - "+x3);
```

Типы данных, переменные, операторы. Потеря точности при преобразовании типов. Example 8

```
        int x4 = 123456789;
        float f = x4;
        double d1 = x4;
        System.out.println("4 - "+f);
        System.out.println("5 - "+d1);

        float f2 = 1.234567890f;
        double d2 = f2;
        System.out.println("6 - "+d2);

        long l2 = 123456789L;
        float f3 = f2;
        System.out.println("7 - "+f3);
        }
}
```

Результат:

```
1 - 1410065408
2 - 127000
3 - 12
4 - 1.23456792E8
5 - 1.23456789E8
6 - 1.2345678806304932
7 - 1.2345679
```

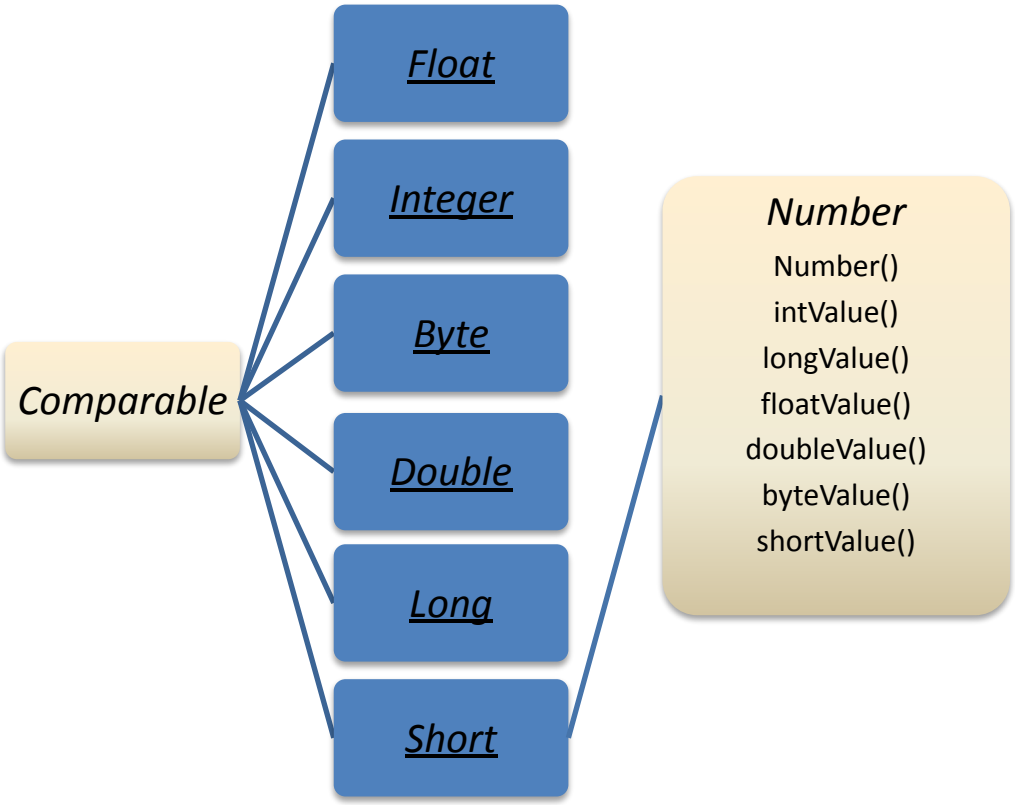
Типы данных, переменные, операторы. Классы-оболочки

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. Указанные классы являются наследниками абстрактного класса **Number** и реализуют интерфейс **Comparable**, представляющий собой интерфейс для работы со всеми скалярными типами.

Объекты этих классов являются константными

Типы данных, переменные, операторы. Классы-оболочки



Типы данных, переменные, операторы. Классы-оболочки. Example 9

```
package _java._se._01._types;
public class IntegerPack {
  public static void main(String[] args) {
    Integer i = new Integer(10);
    System.out.println("In main - before call function - i="  + i);
    changeInteger(i);
    System.out.println("In main - after call function - i="  + i);
  }
  public static void changeInteger(Integer x) {
    System.out.println("In changeInteger - before change - x=" +x);
    x = new Integer(20);
    System.out.println("In changeInteger - after change - x="  +
x);
  }
}
```

Результат:

```
In main - before call function - i=10
In changeInteger - before change - x=10
In changeInteger - after change - x=20
In main - after call function - i=10
```

Типы данных, переменные, операторы. Классы-оболочки

Если требуется создать метод, изменяющий свои числовые параметры, необходимо воспользоваться классами вспомогательных типов из пакета org.omg.CORBA

IntHolder, **BooleanHolder** и др.

Типы данных, переменные, операторы. Классы-оболочки. Example 10

```
package _java._se._01._types;
import org.omg.CORBA.IntHolder;
public class HolderPack {
  public static void main(String[] args) {
    IntHolder i = new IntHolder(10);
    System.out.println("In main - before call function - i="
                       + i.value);
    changeIntHolder(i);
    System.out.println("In main - after call function - i="
                       + i.value);
  }
  public static void changeIntHolder(IntHolder x) {
    System.out.println("In changeIntHolder - before change - x="
                       + x.value);
    x.value++;
    System.out.println("In changeIntHolder - after change - x="
                       + x.value);
  }
}
```

Типы данных, переменные, операторы. Классы-оболочки. Example 10

Результат:

```
In main - before call function - i=10
In changeIntHolder - before change -
x=10
In changeIntHolder - after change - x=11
In main - after call function - i=11
```

Класс Character не наследуется от Number, так как 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.

- **digit(char ch, in radix)** - переводит цифру ch системы счисления с основанием radix в ее числовое значение типа int.
- **forDigit(int digit, int radix)** - производит обратное преобразование целого числа digit в соответствующую цифру (тип char) в системе счисления с основанием radix.

Типы данных, переменные, операторы. Классы-оболочки

- Основание системы счисления должно находиться в диапазоне от **Character.MIN_RADIX** до **Character.MAX_RADIX**.
- Метод toString() переводит символ, содержащийся в классе, в строку с тем же символом.
- Статические методы **toLowerCase(), touppercase(), toTitieCase()** возвращают символ, содержащийся в классе, в указанном регистре. Последний из этих методов предназначен для правильного перевода в верхний регистр четырех кодов Unicode, не выражающихся одним символом.
- Множество статических логических методов проверяют различные характеристики символа, переданного в качестве аргумента метода.

Типы данных, переменные, операторы. Классы-оболочки. Example 11

```
package _java._se._01._types;
public class CharacterTest {
  public static void main(String[] args){
    char ch = '9';
    Character c1 = new Character(ch);
    System.out.println("ch = " + ch);
    System.out.println("c1.charValue() = " + c1.charValue());
    System.out.println("number of 'A' = " + Character.digit('A',
    16));
    System.out.println("digit for 12 = " + Character.forDigit(12,
    16));
    System.out.println("c1 = " + c1.toString() );
    System.out.println("ch isDefined? " +
    Character.isDefined(ch));
    System.out.println("ch isDigit? " + Character.isDigit(ch));
    System.out.println("ch isIdentifierIgnorable? " +
    Character.isIdentifierIgnorable(ch));
    System.out.println("ch isJavaIdentifierPart? " +
    Character.isJavaIdentifierPart(ch));
```

Типы данных, переменные, операторы. Классы-оболочки. Example 11

```
    System.out.println("ch isJavaIdentifierStart? "  +
    Character.isJavaIdentifierStart(ch));
    System.out.println("ch isLetter? " +
    Character.isLetter(ch));
    System.out.println("ch isLetterOrDigit? "  +
    Character.isLetterOrDigit(ch));
  }
}
```

Результат:

```
ch = 9
cl.charValue() = 9
number of 'A' = 10
digit for 12 = c
cl = 9
ch isDefined? true
ch isDigit? true
ch isIdentifierIgnorable? false
ch isJavaIdentifierPart? true
ch isJavaIdentifierStart? false
ch isLetter? false
ch isLetterOrDigit? true
```

Типы данных, переменные, операторы. 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.

Типы данных, переменные, операторы. Big-классы. Example 12

```
package _java._se._01._types;
import java.math.BigDecimal;
import java.math.BigInteger;
public class BigNumbers {
  public static void main(String[] args) {
    BigInteger numI1, numI2, bigNumI;
    BigDecimal numD1, numD2, bigNumD;
    numI1 = BigInteger.valueOf(100000000);// преобразование числа
в большое
    // число
    numI2 = BigInteger.valueOf(200000);
    bigNumI = BigInteger.valueOf(1);
    for (int i = 0; i < 10000000; i++)
     bigNumI =
bigNumI.add(numI1.multiply(numI2).multiply(numI2));
    System.out.println("bigNumI = " + bigNumI);
  }
}
```

Результат:

```
bigNumI =
40000000000000000000000001
```

Типы данных, переменные, операторы. Упаковка/распаковка

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.

Типы данных, переменные, операторы. Упаковка/распаковка

Допускается участие объектов в арифметических операциях, однако не следует этим злоупотреблять, поскольку упаковка/распаковка является ресурсоемким процессом.

```
package _java._se._01._types;
public class NeewProperties {
    public static void main(String[] args) {
        Integer j = 71; // создание объекта+упаковка
        Integer k = ++j; // распаковка+операция+упаковка
        int i = 2;
        k = i + j + k;
    }
}
```

Типы данных, переменные, операторы. Упаковка/распаковка

Несмотря на то, что значения базовых типов могут быть присвоены объектам классов-оболочек, сравнение объектов между собой происходит по ссылкам.

Метод **equals()** сравнивает не значения объектных ссылок, а значения объектов, на которые установлены эти ссылки. Поэтому вызов **oa.equals(ob)** возвращает значение **true**.

Значение базового типа может быть передано в метод **equals()**. Однако ссылка на базовый тип не может вызывать методы.

Типы данных, переменные, операторы. Упаковка/распаковка. Example 12

```
package _java._se._01._types;
public class ComparePack {
  public static void main(String[] args) {
     int i = 128;// заменить на 127 !!!
     Integer oa = i;  // создание объекта+упаковка
     Integer ob = i;
     System.out.println("oa==i " + (oa == i));// true
     System.out.println("ob==i " + (ob == i));// true
     System.out.println("oa==ob " + (oa == ob));// false
     System.out.println("equals ->" + oa.equals(i) + ob.equals(i)
             + oa.equals(ob));// true
  }
}
```

Результат:

```
oa==i true
ob==i true
oa==ob false
equals ->truetruetrue
```

Типы данных, переменные, операторы. Упаковка/распаковка

При инициализации объекта класса-оболочки значением базового типа преобразование типов необходимо указывать явно.

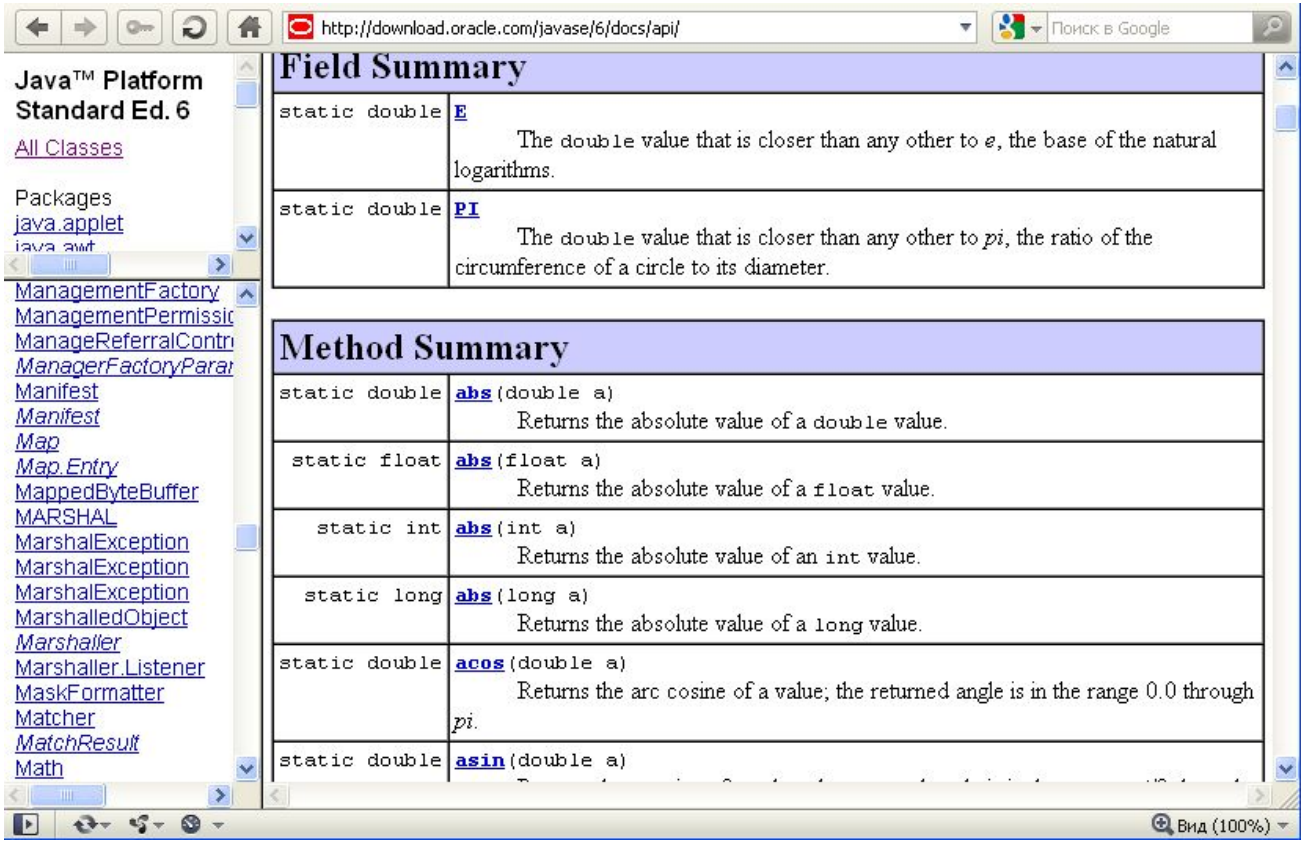
Возможно создавать объекты и массивы, сохраняющие различные базовые типы без взаимных преобразований, с помощью ссылки на класс Number.

```
Number n1 = 1;
Number n2 = 7.1;
Number array[] = {71, 7.1, 7L};
```

При автоупаковке значения базового типа возможны ситуации с появлением некорректных значений и непроверяемых ошибок.

Типы данных, переменные, операторы. Класс Math

Для организации математических вычислений в Java существует класс Math.



Типы данных, переменные, операторы. 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.

Типы данных, переменные, операторы. Статический импорт. Example 13

```
package _java._se._01._types;
import static java.lang.Math.sqrt;
import static java.lang.Math.pow;
public class StaticImport {
    private int i = 20;
    private int j = 40;

    public void staticImport() {
        double x, y;
        x = pow(i, 2);
        y = sqrt(j) / 2;
        System.out.println("x=" + x + " y=" + y);
    }

    public static void main(String[] args) {
        StaticImport obj = new StaticImport();
        obj.staticImport();
    }
}
```

Результат:

```
x=400.0
y=3.1622776601683795
```

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

Битовые операторы

|	Или	**>>**	Сдвиг вправо
|=	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		

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

Логические операторы

||	Или	**&&**	И
!	Унарное отрицание		

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.

Типы данных, переменные, операторы. Операторы

Операции над целыми числами: +, −, *, %, /, ++,-- и битовые операции &, |, ^, ~ аналогичны операциям большинства языков программирования.

Деление на ноль целочисленного типа вызывает исключительную ситуацию, переполнение не контролируется.

Типы данных, переменные, операторы. Операторы

Операции над числами с плавающей точкой практически те же, что и в других языках, но по стандарту IEEE 754 введены понятие бесконечности +Infinity и –Infinity и значение NaN (Not a Number). Результат деления положительного числа на 0 равен положительной бесконечности, отрицательного – отрицательной бесконечности. Вычисление квадратного корня из отрицательного числа или деление 0/0 – не число. Проверить, что какой-то результат равен не числу можно с помощью методов Double.isNan(<arg>) или Float.isNaN(<arg>), возвращающих значение типа boolean.

Типы данных, переменные, операторы. Приоритет операций

№	Операция	Порядок выполнения
1	**[ ] . () (вызов метода)**	Слева направо
2	**! ~ ++ -- +(унарный) -(унарный) () (приведение) new**	Справа налево
3	*** / %**	Слева направо
4	**+ -**	Слева направо
5	**<< >> >>>**	Слева направо
6	**< <= > >= instanceof**	Слева направо
7	**== !=**	Слева направо
8	**&**	Слева направо
9	**^**	Слева направо
10	**|**	Слева направо
11	**&&**	Слева направо
12	**||**	Слева направо
13	**?:**	Слева направо
14	**= += -= *= /= %= |= ^= <<= >>= >>>=**	Справа налево

Типы данных, переменные, операторы. Вычисления с плавающей точкой

Все вычисления, которые проводятся над числами с плавающей точкой следуют стандарту IEEE 754. В Java есть три специальных числа с плавающей точкой

Положительная бесконечность
Отрицательная бесконечность
Не число

В языке Java существуют константы

- **Double.POSITIVE_INFINITY;**
- **Float.POSITIVE_INFINITY;**
- **Double.NEGATIVE_INFINITY;**
- **Float.NEGATIVE_INFINITY;**
- **Double.NaN;**
- **Float.NaN;**

Типы данных, переменные, операторы. Вычисления с плавающей точкой. Example 14

```
package _java._se._01._types;
public class DoubleCalc {
  public static void main(String[] args) {
    double i = 7.0;
    double j, z, k;
    j = i / 0;
    z = -i / 0;
    k = Math.sqrt(-i);
    if (j == Double.POSITIVE_INFINITY)
        System.out.println("Мы получили положительную бесконечность.");
    if (z == Double.NEGATIVE_INFINITY)
        System.out.println("Мы получили отрицательную бесконечность.");
    if (Double.isNaN(k))
        System.out.println("Мы получили не число.");
    System.out.println("j=" + j + " z=" + z + " k=" + k);
  }
}
```

Результат:

```
Мы получили положительную бесконечность.
Мы получили отрицательную бесконечность.
Мы получили не число.
j=Infinity z=-Infinity k=NaN
```

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 булевское 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 (boolexpr) { /*операторы*/ }
2. do { /*операторы*/ }
   while (boolexp);
3. for(exp1; boolexp; exp3){ /*операторы*/ }
4. for((Тип exp1 : exp2){ /*операторы*/ }
```

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.

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

Для цикла **for** не рекомендуется в цикле 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.

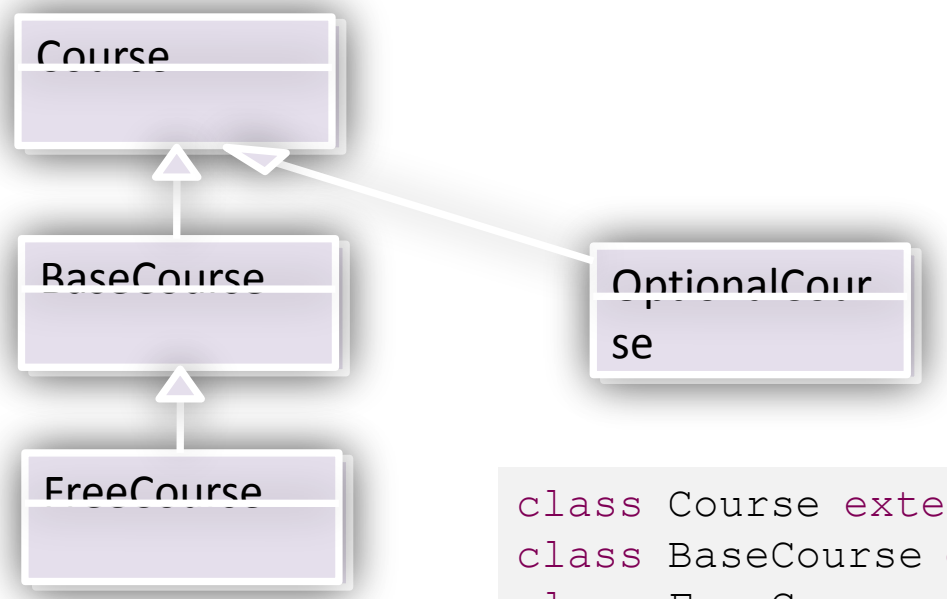
Оператор **switch**:

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 */
}
```

Типы данных, переменные, операторы. 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 {}
```

Типы данных, переменные, операторы. Instanceof

Объект подкласса может быть использован всюду, где используется объект суперкласса

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

Типы данных, переменные, операторы. Instanceof. Example 15

```
package _java._se._01._types;

public class InstanceofTest {

  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.");
        }
  }
}
```

Типы данных, переменные, операторы. Instanceof. Example 15

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

Результат:

```
BaseCourse
OptionalCourse
BaseCourse
```

Типы данных, переменные, операторы. Ссылочные типы данных. Базовые элементы работы со строками.

Создание переменной ссылочного типа:

```
String s1 = new String(“World”);
```

Для класса String можно использовать упрощенный синтаксис

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

Типы данных, переменные, операторы. Ссылочные типы данных. Базовые элементы работы со строками.

- 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**()
- Длина строки определяется с помощью метода **length**() - **int len = str.length**();

Типы данных, переменные, операторы. Ссылочные типы данных. Базовые элементы работы со строками. Example 16

```
package _java._se._01._types;
public class ComparingStrings {
  public static void main(String[] args) {
    String s1, s2;
    s1 = "Java";
    s2 = s1; /*
          * переменная ссылается на ту же строку
          */
    System.out.println("сравнение ссылок " + (s1 == s2)); // результат true
    // создание нового объекта добавлением символа
    s1 += '2';
    // s1-="a"; //ошибка, вычитать строки нельзя
    // создание нового объекта копированием
    s2 = new String(s1);
    System.out.println("сравнение ссылок " + (s1 == s2)); // результат false
    System.out.println("сравнение значений " + s1.equals(s2)); // результат
                                // true
  }
}
```

Результат:

```
сравнение ссылок true
сравнение ссылок false
сравнение значений true
```

Типы данных, переменные, операторы. Ссылочные типы данных. Базовые элементы работы со строками

Перевести строковое значение в величину типа **int** или **double** можно с помощью методов **parseInt()** и **parseDouble()** классов **Integer** и **Double**. Обратное преобразование возможно при использовании метода **valueOf()** класса **String**. Кроме того, любое значение можно преобразовать в строку путем конкатенации его (**+**) с пустой строкой (**“”**).

Типы данных, переменные, операторы. Ссылочные типы данных. Базовые элементы работы со строками. Example 17

```
package _java._se._01._types;
public class StrToNum {
     public static void main(String[] args) {
          String strInt = "123";  String strDouble = "123.456";
          int x;  double y;
          x = Integer.parseInt(strInt);
          y = Double.parseDouble(strDouble);
          System.out.println("x=" + x);
          System.out.println("y=" + y);
          strInt = String.valueOf(x + 1);
          strDouble = String.valueOf(y + 1);
          System.out.println("strInt=" + strInt);
          System.out.println("strDouble=" + strDouble);
          String str;
          str = "num=" + 345;
          System.out.println(str);
     }
}
```

Результат:

```
x=123
y=123.456
strInt=124
strDouble=124.456
num=345
```

Типы данных, переменные, операторы. Ссылочные типы данных. Базовые элементы работы со строками. Example 18

Для преобразования целого числа в десятичную, двоичную, шестнадцатеричную и восьмеричную строки используются методы **toString()**, **toBinaryString()**, **toHexString()** и **toOctalString()**.

```
package _java._se._01._types;
public class StrToNum2 {
    public static void main(String[] args) {
        System.out.println(Integer.toString(262));
        System.out.println(Integer.toBinaryString(262));
        System.out.println(Integer.toHexString(267));
        System.out.println(Integer.toOctalString(267));
    }
}
```

Результат:

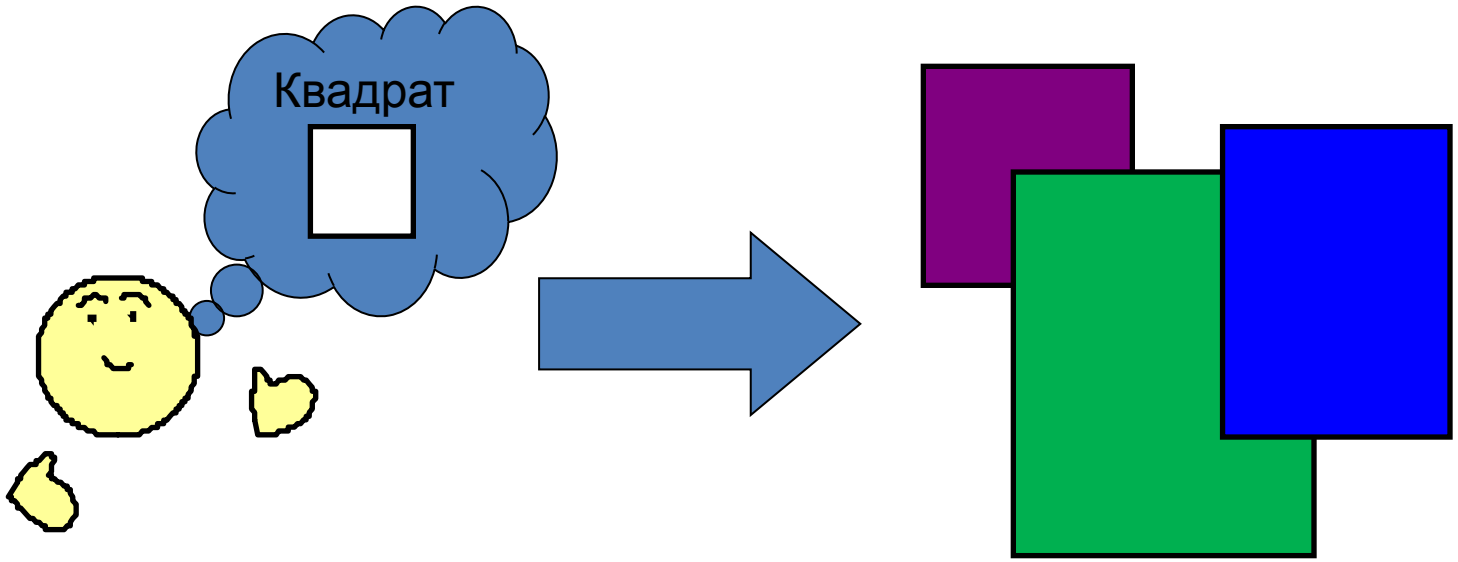
```
262
100000110
10b
413
```

ПРОСТЕЙШИЕ КЛАССЫ И ОБЪЕКТЫ

Простейшие классы и объекты. Определения

Объект – некоторая КОНКРЕТНАЯ сущность моделируемой предметной области

Класс – шаблон или АБСТРАКЦИЯ сущности предметной области



Простейшие классы и объекты. Определения

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.

Каждый класс имеет свое имя, отличающее его от других классов, и относится к определенному пакету. Имя класса в пакете должно быть уникальным. Физически пакет представляет собой каталог, в который помещаются программные файлы, содержащие реализацию классов.

Классы позволяют разбить поведение сложных систем на простое взаимодействие взаимосвязанных объектов.

Простейшие классы и объекты. Свойства и методы класса

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

Простейшие классы и объекты. Свойства и методы класса

Определение класса включает:

- Модификатор доступа
- Ключевое слово class
- Свойства класса
- Конструкторы
- Методы
- Статические свойства
- Статические методы

Простейшие классы и объекты. Свойства и методы класса

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

Методы определяются только внутри класса. Указывается:

- Модификатор доступа
- Слово static
- Тип возвращаемого значения
- Аргументы

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

Простейшие классы и объекты. Свойства и методы класса

Объявление класса имеет вид:

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

Создание объекта имеет вид:

```
имя_класса имя_объекта= new конструктор_класса([аргументы]);
```

Простейшие классы и объекты. Свойства и методы класса. Блоки кода

- Блоки кода обрамляются в фигурные скобки “{“ “}”
- Охватывают определение класса
- Определения методов
- Логически связанные разделы кода

```
package _java._se._01._easyclass;
import java.util.Date;
public class SimpleProgram {
     private Date today;
     public Date getToday() {
          return today;
     }
     public static final int PROGRAM_SIZE = 560;
     public static void main(String[] args) {
          SimpleProgram object = new SimpleProgram();
          System.out.println(object.getToday());
          System.out.println(object. PROGRAM_SIZE);
     }
}
```

Простейшие классы и объекты. Атрибуты доступа

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** (класс доступен в данном пакете). Данное слово при объявлении вообще не используется и не является ключевым словом языка, мы его используем для обозначения.

Простейшие классы и объекты. Конструкторы

Конструктор – это метод, который автоматически вызывается при создании объекта класса и выполняет действия *только* по *инициализации объекта*;

Конструктор имеет **то же имя**, что и класс;

Вызывается не по имени, а только **вместе с ключевым словом new** при создании экземпляра класса;

Конструктор **не возвращает значение**, но может иметь параметры и быть перегружаемым.

Простейшие классы и объекты. Конструкторы. Example 19

```
package _java._se._01._easyclass;

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) {
          x = size;
          y = size;
     }
     public int getX() {
          return x;
     }
     public int getY() {
          return y;
     }
}
```

Простейшие классы и объекты. Конструкторы. Example 19

```
package _java._se._01._easyclass;
public class Point2DUse {

  public static void main(String[] args) {
     Point2D obj1 = new Point2D(1, 2);
     Point2D obj2 = new Point2D(3);
     System.out.println(obj1.getX() + " " + obj1.getY());
     System.out.println(obj2.getX() + " " + obj2.getY());
  }
}
```

Результат:

```
1 2
3 3
```

Простейшие классы и объекты. 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*.

Простейшие классы и объекты. Пакеты

При компиляции поиск пакетов осуществляется в:

- рабочем каталоге
- используя параметр переменной среды CLASSPATH
- указывая местонахождение пакета параметром компилятора -classpath

Простейшие классы и объекты. Пакеты

Пакеты регулируют права доступа к классам и подклассам.

Сущности (интерфейсы, классы, методы, поля), помеченные ключевым словом public, могут использоваться любым классом.

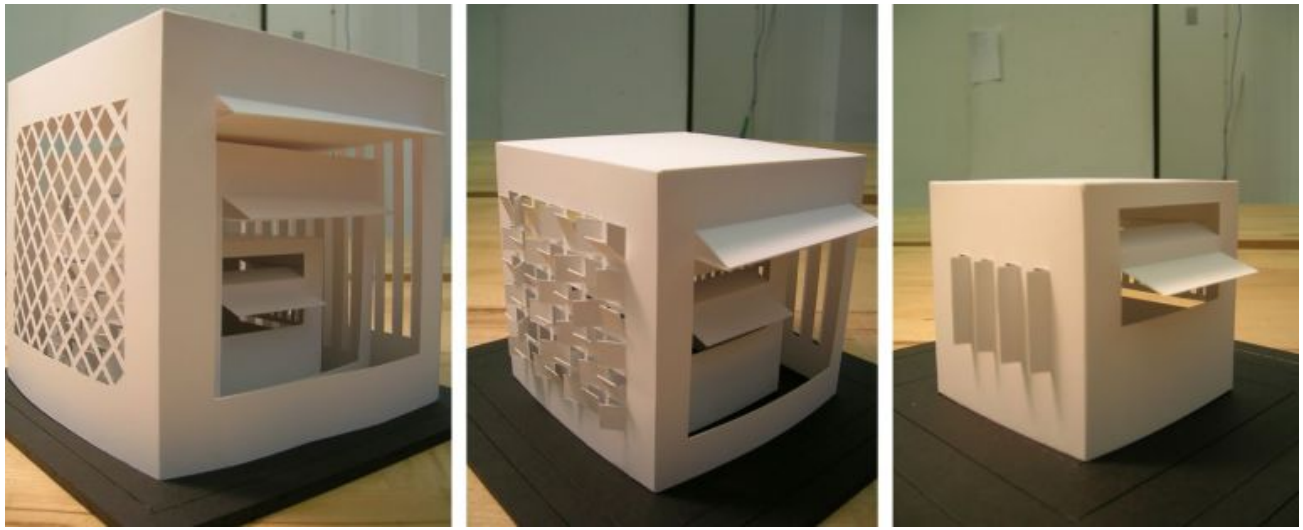
Закрытые сущности могут использоваться только определившим их классом.

Если ни один модификатор доступа неуказан, то сущность (т.е. класс, метод или переменная) является доступной всем методам в том же самом *пакете.*

Простейшие классы и объекты. Пакеты

Для подключения пакета используется ключевое слово **import**.

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



Простейшие классы и объекты. Пакеты. Example 20

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

```
package _java._se._01._easyclass.mypackage.package2;
import _java._se._01._easyclass.mypackage.package1.Class1;
public class Class3 {
     public static void main(String[] args) {
          Class1 cl1 = new Class1();
     }
}
```

```
package _java._se._01._easyclass.mypackage.package1;
public class Class4 {
     Class2 obj = new Class2();

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

JAVA BEANS

JavaBeans. Определение

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

С точки зрения ООП, *компонент JavaBean* – это классический самодостаточный *объект*, который, будучи написан один раз, может быть многократно использован при построении новых апплетов, сервлетов, полноценных приложений, а также других компонент JavaBean.

JavaBeans. Определение

Отличие от других технологий заключается в том, что компонент JavaBean 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)*

JavaBeans. Свойства Bean

Свойства компоненты 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**.

JavaBeans. Свойства Bean. Example 21

```
package _java._se._01._beans;
import java.awt.Color;

public class BeanExample {
     private Color color;

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

     public Color getColor(){
       return color;
     }
}
```

JavaBeans. Свойства Bean

Следует заметить, что согласно спецификации Bean, аналогичные *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[].

JavaBeans. Свойства Bean. Example 22

```
package _java._se._01._beans;
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);
     }
}
```

JavaBeans. Свойства Bean. Example 23

Атрибуту типа **boolean** *в классе Bean* должны соответствовать несколько иные методы: **is** и **set**

```
package _java._se._01._beans;
public class BeanBoolExample {
     private boolean ready;
     public void setReady(boolean newStatus){
          ready = newStatus;
     }
     public boolean isReady(){
          return ready;
     }
}
```

Формально к свойствам компонента Bean следует отнести также инициируемые им события. Каждому из этих событий в компоненте Bean также должно соответствовать два метода - add и remove.

JavaBeans. Example 24

```
package _java._se._01._beans;
public class UserBean {
     public int numericCode; // нарушение инкапсуляции
     private String password;

     public int getNumericCode() {
          return numericCode;
     }

     public void setNumericCode(int value) {
          if (value > 0) { numericCode = value; }
          else { numericCode = 1; }
     }

     public String getPassword() {
     // public String getPass() { // некорректно - неполное имя
          return password;
     }
     public void setPassword(String pass) {
          if (pass != null) { password = pass; }
          else { password = pass;}
     }
}
```

JavaBeans. Использование

Может показаться, что нет никакой разницы, предоставляем ли мы доступ извне непосредственно к свойству компоненты Bean, или же для достижения того же самого результата используем методы set и get. Принципиально важное отличие заключается в том, что в последнем случае мы *получаем возможность контролировать все изменения этого свойства*. Например, мы можем связать с методом set определенный программный код, который будет автоматически оповещать другие компоненты приложения, если кто-то попытается изменить значение этого свойства.

JavaBeans. Синхронизация

Заметим, что реализуя тот или иной метод, разработчик должен учитывать, что создаваемый им компонент Bean должен будет функционировать 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. Наиболее *тривиальный способ синхронизации* таких запросов заключается в том, чтобы пометить все методы класса Bean директивой **synchronized**.

МАССИВЫ

Массивы. Определения

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.

Массивы. Определения

Значения элементов неинициализированных массивов, для которых выделена память, устанавливается в нуль.

Многомерных массивов в Java не существует, но можно объявлять массивы массивов. Для задания начальных значений массивов существует специальная форма инициализатора.

Массивы. Определения

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.**

Массивы. Объявление и инициализация. Example 25

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.

```
package _java._se._01._array;
public class CreateArray {
  public static void main(String[] args) {
     // примитивный тип, размер массива задан явно
     int[] price = new int[10];
     // неявное задание размера
     int[] rooms = new int[] { 1, 2, 3 };
     // содержит ссылочные переменные
     Item[] items = new Item[10];
     Item[] undefinedItems = new Item[] { new Item(1), new Item(2),
               new Item(3) };
  }
}
class Item {
     public Item(int i) {
     }
}
```

Массивы. Объявление и инициализация. Example 25

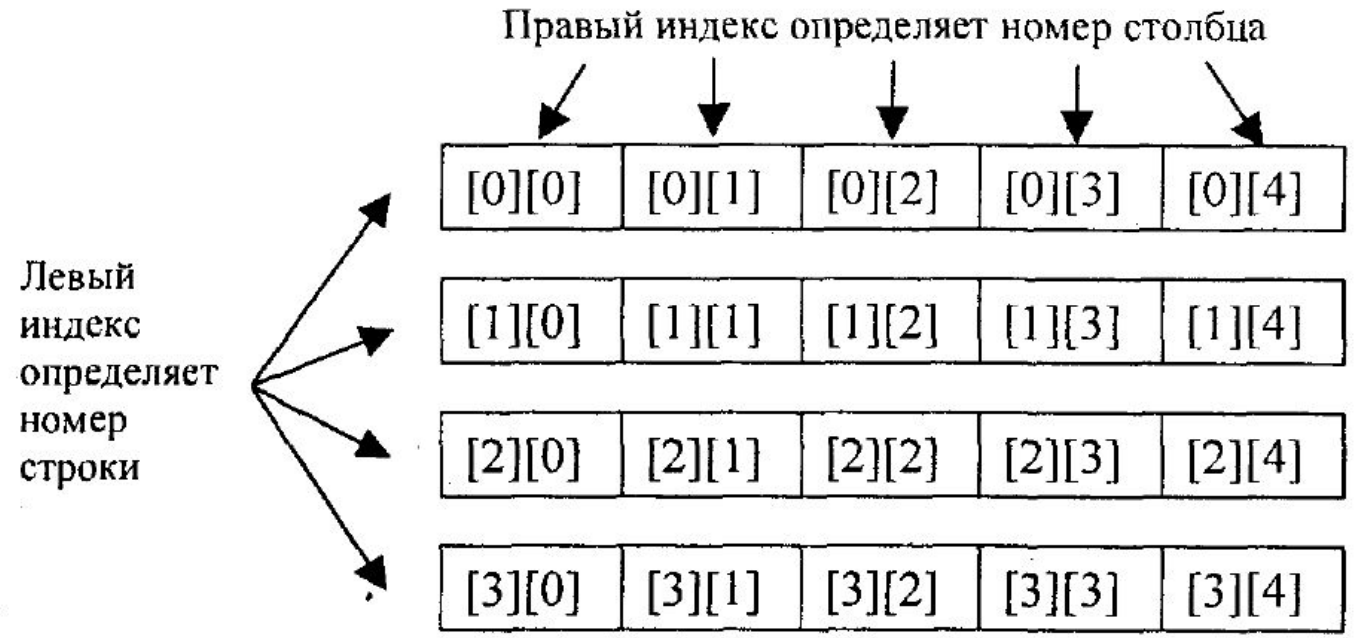
```
package _java._se._01._array;
public class FindReplace {
  public static void main(String[] args) {
     int myArray[];
     int mySecond[] = new int[100];
     int a[] = { 5, 10, 0, -5, 16, -2 };
     int max = a[0];
     for (int i = 0; i < a.length; i++) if (max < a[i]) max = a[i];
     for (int i = 0; i < a.length; i++) {
          if (a[i] < 0) a[i] = max;
          mySecond[i] = a[i];
          System.out.println("a[" + i + "]= " + a[i]);
     }
     myArray = a; // установка ссылки на массив a
  }
}
```

Результат:

```
a[0]= 5
a[1]= 10
a[2]= 0
a[3]= 16
a[4]= 16
a[5]= 16
```

Массивы. Массив массивов

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



Определение: `int twoD[] [] = new int [4][5]`

Массивы. Массив массивов

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 }
    };
```

Массивы. Работа с массивами

Члены объектов-массивов:

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

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

Массивы. Работа с массивами. Example 26

```
package _java._se._01._array;

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]);
  }
}
```

Результат:

```
false   true
```

Массивы. Работа с массивами. Example 27

```
package _java._se._01._array;

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);
        }
    }
}
```

Массивы. Работа с массивами. Example 27

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

Результат:

```
true
java.lang.ArrayStoreException: _java._se._01._array.Point
```

Массивы. Ошибки времени выполнения. Example 28

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

```
package _java._se._01._array;
public class ArrayIndexError {
    public static void main(String[] args) {
        int array[] = new int[] { 1, 2, 3 };
        System.out.println(array[3]);
    }
}
```

Результат:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 3
        at _java._se._01._array.ArrayIndexError.main(ArrayIndexError.java:6)
```

Массивы. Ошибки времени выполнения. Example 29

Попытка поместить в массив неподходящий элемент пресекается виртуальной машиной:

```
package _java._se._01._array;
public class ArrayTypeError {
    public static void main(String[] args) {
        Object x[] = new String[3];
        // попытка поместить в массив содержимое
//несоответствующего типа
        x[0] = new Integer(0);
    }
}
```

Результат:

```
Exception in thread "main" java.lang.ArrayStoreException: java.lang.Integer
        at _java._se._01._array.ArrayTypeError.main(ArrayTypeError.java:7)
```

CODE CONVENTIONS

Code conventions. Code conventions for Java Programming

Содержание: имена файлов, организация структуры файлов, структурированное расположение текста, комментарии, объявления, операторы, пробельные символы, соглашение об именовании, практики программирования.

80% стоимости программного обеспечения уходит на поддержку.

Едва ли программное обеспечение весь свой жизненный цикл будет поддерживаться автором..

Code conventions улучшает удобочитаемость программного кода, позволяя понять новый код долее быстро и полностью.

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

Code conventions. Best Practices

- *Объявляйте локальные переменные сразу перед использованием*
  - Определяется их область видимости.
  - Уменьшается вероятность ошибок и неудобочитаемости.

- *Поля необходимо объявлять 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.

Code conventions. Best Practices

- *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.

Code conventions. Best Practices

- *С осторожностью используйте System.Exit(0) с многопоточными приложениями.*
  - Нормальный способ завершения программы должен завершать работу всех используемых потоков.

- *Используйте интерфейсы для определения констант.*
  - Создание класса для констант является оправданным, только если это широко используемые константы.

- *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.

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

Code conventions. Best Practices

- *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.

Code conventions. Best Practices

- *Избегайте пустых блоков catch*
  - В этом случае когда происходит исключение, то 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.

Code conventions. Best Practices

- *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.

Code conventions. Best Practices

- *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.

Code conventions. Best Practices

- *Declarations. Длина строк кода*
  - Не используйте строки длиной более 80 символов.
- *Объявление переменных*
  - Не присваивайте одинаковые значения нескольким переменных одним оператором.

    fooBar.fChar = barFoo.lchar = ‘c’;c// AVOID!!!
- *При декларировании переменных объявляйте по одной переменной в строке кода*
  - Такое объявление позволяет писать понятные комментарии.

Code conventions. Best Practices

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

```
argv++; // Correct
argc-; // Correct
argv++; argc-; II AVOID!
```

Code conventions. Соглашение об именовании

- Имена файлов
  - Customer.java
  - Person.class
- Имена пакетов
  - java.util
  - javax.swing
- Имена классов
  - Customer
  - Person

Code conventions. Соглашение об именовании

- Имена свойств класса
  - firstName
  - Id
- Имена методов
  - getName
  - isAlive
- Имена констант
  - SQUARE_SIZE

Также могут использоваться цифры 1..9, _, $

ПАРАМЕТРИЗОВАННЫЕ КЛАССЫ

Параметризованные классы. Назначение и синтаксис

С помощью шаблонов можно создавать параметризованные (родовые, generic) классы и методы, что позволяет использовать более строгую типизацию, например при работе с коллекциями.

Пример класса-шаблона с двумя параметрами:

```
package java.se._01.generics;
public class Message < T1, T2 >{
    T1 id;
    T2 name;
}
```

Здесь 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:

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

Параметризованные классы. Назначение и синтаксис. Example 30

```
package _java._se._01._generics;
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;
     }
}
```

Параметризованные классы. Назначение и синтаксис. Example 30

```
package _java._se._01._generics;
public class OptionalDemo {
     public static void main(String[] args) {
         // параметризация типом Integer
         Optional<Integer> ob1 = new Optional<Integer>();
    ob1.setValue(1);

    // ob1.setValue("2");// ошибка компиляции: недопустимый тип
    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; //ошибка компиляции – параметризация не ковариантна
```

Параметризованные классы. Назначение и синтаксис. Example 30

```
    // параметризация по умолчанию – Object
    Optional ob3 = new Optional();
    System.out.println(ob3.getValue());
    ob3.setValue("Java SE 6");
    System.out.println(ob3.toString()); /* выводится тип
объекта, а не тип параметризации */
    b3.setValue(71);
    System.out.println(ob3.toString());
    ob3.setValue(null);
     }
}
```

Результат:

```
1
Java
null
java.lang.String Java SE 6
java.lang.Integer 71
```

Параметризованные классы. Использование extends

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.

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

```
package _java._se._01._generics;
public class OptionalExt <T extends Тип> {
    private T value;
}
```

Параметризованные классы. Использование extends

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

Параметризованные классы. Метасимволы

Часто возникает необходимость в метод параметризованного класса одного допустимого типа передать объект этого же класса, но параметризованного другим типом.

В этом случае при определении метода следует применить метасимвол “**?**”.

<?>

Метасимвол также может использоваться с ограничением **extends** для передаваемого типа.

<? extends Number>

Параметризованные классы. Метасимволы. Example 31

```
package _java._se._01._generics;
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 ();
     }
}
```

Параметризованные классы. Метасимволы. Example 31

```
package _java._se._01._generics;

public class Runner {

     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 ());
     }
}
```

Результат:

```
true
true
71
```

Параметризованные классы. Метасимволы

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.

Параметризованные классы. 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) {}

<T> *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.

Параметризованные классы. Параметризованные методы. Example 32

```
package _java._se._01._generics;

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 компиляции
     }
}
```

Результат:

```
7
7
```

Параметризованные классы. Ограничения на использование

Нельзя явно вызвать конструктор параметризованного класса, так как компилятор не знает, какой конструктор может быть вызван и какой объем памяти должен быть выделен при создании объекта,

Параметризованные поля *не могут быть статическими*, *статические методы не могут иметь параметризованные поля* и обращаться к ним также запрещено.

Параметризованные классы. Применение

Параметризованные методы применяются когда необходимо разработать базовый набор операций, который будет работать с различными типами 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.

ПЕРЕЧИСЛЕНИЯ (ENUMS)

Перечисления. Синтаксис

Examples:

- 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
}
```

Перечисления. Определения

В отличие от статических констант, 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. Определения

Не могут быть абстрактными и содержать абстрактные методы (кроме случая, когда каждый объект перечисления реализовывает абстрактный метод), но могут реализовывать интерфейсы.

Enums переопределяют *toString*() and определяют *valueOf*()

```
Season season = Season.WINTER;
System.out.println(season );
        // prints WINTER
season = Season.valueOf("SPRING");
        // sets season to Season.SPRING
```

Перечисления. Создание объектов перечисления

Экземпляры объектов **перечисления нельзя создать с помощью new**, каждый объект перечисления уникален, создается при загрузке перечисления в виртуальную машину, поэтому допустимо сравнение ссылок для объектов перечислений, **можно использовать switch**

Как и обычные классы могут реализовывать поведение, содержать вложенные классы.

Enums по умолчанию **public**, **static** и **final**

Перечисления. Создание объектов перечисления. Example 33

```
package _java._se._01._enums;
public enum Days {
     SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY;
     public boolean isWeekend() {
          switch(this) {
               case SUNDAY:
               case SATURDAY:
                    return true;
               default:
                    return false;
          }
     }
}
```

```
System.out.println( Days.MODAY+” isWeekEnd(): “ + Days.MONDAY.isWeekend() );
```

Перечисления. 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, выбрасывает **IllegalArgumentException** если нет элемента с указанным именем)

Перечисления. Методы перечисления

Каждый класс перечисления неявно содержит следующие методы:

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

Перечисления. Методы перечисления. Example 34

```
package _java._se._01._enums;

public enum Shape {
     RECTANGLE, TRIANGLE, CIRCLE;
     public double square(double x, double y) {
          switch (this) {
          case RECTANGLE:
               return x * y;
          case TRIANGLE:
               return x * y / 2;
          case CIRCLE:
               return Math.pow(x, 2) * Math.PI;
          }
          throw new EnumConstantNotPresentException(
               this.getDeclaringClass(), this.name());
     }
}
```

Перечисления. Методы перечисления. Example 34

```
package _java._se._01._enums;
public class Runner {
  public static void main(String[] args) {
    double x = 2, y = 3;
    Shape[] arr = Shape.values();
    for (Shape sh : arr)
      System.out.printf("%10s = %5.2f%n", sh, sh.square(x, y));
  }
}
```

Результат:

```
 RECTANGLE =  6,00
  TRIANGLE =  3,00
    CIRCLE = 12,57
```

Перечисления. Конструкторы и анонимные классы для перечисления. Example 35

Класс перечисления **может иметь конструктор** (private либо package), который вызывается для каждого элемента при его декларации. Отдельные элементы перечисления **могут реализовывать** свое **собственное поведение**.

```
package _java._se._01._enums;
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();
     }
}
```

Перечисления. Сравнение переменных перечисления. Example 36

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

```
package _java._se._01._enums;
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);
           }
      }
}
```

Перечисления. Сравнение переменных перечисления. Example 36

```
package _java._se._01._enums;

public enum Faculty {
     MMF,
     FPMI,
     GEO
}
```

Результат:

```
GEO
```

ВНУТРЕННИЕ КЛАССЫ

Внутренние классы. Определение

В Java можно объявлять классы внутри других классов и даже внутри методов. Они делятся на внутренние нестатические, сложенные 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 содавать простые и понятные программы, управляющие событиями.

Внутренние классы. Inner (нестатические). Example 37

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

```
package _java._se._01._innerclasses.inner.outer1;
import java.util.Date;
public class Outer {
     private String str;
     Date date;

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

Внутренние классы. Inner (нестатические)

- Доступ к элементам внутреннего класса возможен только из внешнего класса через объект внутреннего класса. То есть, чтобы класс Outer мог вызвать какой-либо метод класса Inner в классе Outer необходимо создать объект класса Inner и вызывать методы уже через этот объект.

Внутренние классы. Inner (нестатические). Example 38

```
package _java._se._01._innerclasses.inner.outer2;
import java.util.Date;
public class Outer {
     Inner inner;
     private String str;
     Date date;
     Outer() {
          str = "string in outer";
          date = new Date();
          inner = new Inner();
     }
     class Inner {
          public void method() {
               System.out.println(str);
               System.out.println(date.getDate());
          }
     }
     public void callMethodInInner() {
          inner.method();
     }
}
```

Внутренние классы. Inner (нестатические). Example 38

```
package _java._se._01._innerclasses.inner.outer2;

public class OuterInnerTest {
     public static void main(String[] args) {
          Outer outer = new Outer();
          outer.callMethodInInner();
     }
}
```

Результат:

```
string in outer
3
```

Внутренние классы. Inner (нестатические)

- Объект внутреннего класса имеет ссылку на объект своего внешнего класса. Ссылка эта неявная. Предположим что имя это ссылки ref_outer (естественно, никакой реальной ссылки с таким именем по умолчанию во внутреннем классе не предусматривается), тогда любой доступ к элементам внешнего класса из внутреннего выглядит следующим образом. Именно эта неявная ссылка и позволяет методам внутреннего класса иметь прямой доступ к полям и методам внешнего класса.

Внутренние классы. Inner (нестатические). Example 39

```
package _java._se._01._innerclasses.inner.outer3;

import java.util.Date;

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

Внутренние классы. Inner (нестатические)

- Внутренние классы не могут содержать static-полей, кроме final static

Внутренние классы. Inner (нестатические). Example 40

```
package _java._se._01._innerclasses.inner.outer4;
import java.util.Date;
public class Outer {
      Inner inner;
      private String str;
      Date date;
      Outer() {
            str = "string in outer";
            date = new Date();
            inner = new Inner();
      }
      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.getDate());
            }
      }
      public void callMethodInInner() {
            inner.method();
      }
}
```

Внутренние классы. Inner (нестатические). Example 41

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

имя_внешнего_класса.имя_внутреннего класса.
имя_статической_переменной

```
package _java._se._01._innerclasses.inner.outer4;
public class OuterInnerTest {
    public static void main(String[] args) {
        Outer outer = new Outer();
        System.out.println(Outer.Inner.pubfsi_pole);
    }
}
```

Внутренние классы. Inner (нестатические). Example 42

- Также доступ к переменной типа final static возможен во внешнем классе через имя внутреннего класса:

```
package _java._se._01._innerclasses.inner.outer5;
public class Outer {
     Inner inner;
     Outer() {
          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. Inner (нестатические). Example 43

- Внутренние классы могут быть производными от других классов. Внутренние классы могут быть базовыми

```
package _java._se._01._innerclasses.inner.outer6;
public class Outer {
     private int privI = 1;
     protected int protI = 2;
     public int pubI = 3;
     class Inner1 {
          private int inner1_privI = 11;
          protected int inner1_protI = 22;
          public int inner1_pubI = 33;
          public void print() {
System.out.println(privI + " " + protI + " "
+ pubI + " " + inner1_privI + " " + inner1_protI + " " + inner1_pubI);
          }
     }
```

Внутренние классы. Inner (нестатические). Example 43

```
     class Inner2 extends Inner1 {
          private int inner2_privI = 111;
          protected int inner2_protI = 222;
          public int inner2_pubI = 333;
          public void print() {
System.out.println(privI + " " + protI + " " + pubI
+ " " + inner1_protI + " " + inner1_pubI);
System.out.println(inner2_privI + " "
+ inner2_protI + " "     + inner2_pubI);
          }
     }

     class Inner3 extends Outer2 {
          private int inner3_privI = 1111;
          protected int inner3_protI = 2222;
          public int inner3_pubI = 3333;
          public void print() {
System.out.println(privI + " " + protI
+ " " + pubI + " " + outer2_protI + " " + outer2_pubI);
System.out.println(inner3_privI + " "
+ inner3_protI + " "     + inner3_pubI);
          }
     }
}
```

Внутренние классы. Inner (нестатические). Example 43

```
class Outer2 {
     private int outer2_privI = 11111;
     protected int outer2_protI = 22222;
     public int outer2_pubI = 33333;
}
```

Внутренние классы. Inner (нестатические). Example 44

- Внутренние классы могут реализовывать интерфейсы

```
package _java._se._01._innerclasses.inner.outer7;
public class Outer {
  class Inner1 implements MyInterfaceInner, MyInterfaceOuter {
    public void interfaceInnerPrint() {
        System.out.println("interfaceInnerPrint");
    }
    public void interfacePrint() {
        System.out.println("interfacePrint");
    }
  }
  interface MyInterfaceInner {
    void interfaceInnerPrint();
  }
}
interface MyInterfaceOuter {
    void interfacePrint();
}
```

Внутренние классы. Inner (нестатические). Example 45

- Внутренние классы могут быть объявлены с параметрами final, abstract, public, protected, private

```
package
_java._se._01._innerclasses.inner.outer8;
public class Outer {
     public class Inner1 {
     }
     private class Inner2 {
     }
     protected class Inner3 {
     }
     abstract private class Inner4 {
     }
     final protected class Inner5 {
     }
}
```

Внутренние классы. Inner (нестатические). Example 46

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

имя_внешнего_класса.имя_внутреннего_класса

```
package _java._se._01._innerclasses.inner.outer9;
public class Outer {
    public class Inner1{
        void print(){
            System.out.println("Inner1");
        }
    }
    protected class Inner2{
        void print(){
            System.out.println("Inner1");
        }
    }
}
```

Внутренние классы. Inner (нестатические). Example 46

```
package _java._se._01._innerclasses.inner.outer9;
public class Outer2 {
     public static void main(String[] args) {
          Outer.Inner1 obj1 = new Outer().new Inner1();
          Outer.Inner2 obj2 = new Outer().new Inner2();
          obj1.print();
          obj2.print();
     }
}
```

Внутренние классы. Inner (нестатические)

- 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.

Внутренние классы. Inner (нестатические). Example 47

```
package _java._se._01._innerclasses.inner.outer10;
public class Outer {
    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) {
        Outer out = new Outer();
        out.method();
    }
}
```

Внутренние классы. Inner (нестатические). Example 48

- Локальные внутренние классы не объявляются с помощью модификаторов доступа.

```
package
_java._se._01._innerclasses.inner.outer11;
public class Outer {
     public void method() {
          public class Inner1 {
          } // ОШИБКА
     }
}
```

Внутренние классы. Inner (нестатические)

- Правила для внутренних классов.

1) ссылка на внешний класс имеет вид

имя_внешнего_класса.this

Для получения доступа из внутреннего класса к экземпляру его внешнего класса необходимо в ссылке указать имя класса и ключевое слово **this**, поставив между ними точку (например, OuterClass.this). Ключевое слово **this** обеспечивает доступ к потенциально спрятанным методам и полям, в которых внутренние и внешние классы используют метод или переменную с одинаковыми именами.

Внутренние классы. Inner (нестатические)

Например, в следующем определении класса и у внешнего и у внутреннего классов присутствует переменная **count**. Для получения доступа к переменной внешнего класса, необходимо в ссылке на переменную перед ее именем приписать префикс this и имя внешнего класса.

Внутренние классы. Inner (нестатические). Example 49

```
package _java._se._01._innerclasses.inner.outer12;
class OuterClass {
  int count = 0;
  class InnerClass {
    int count = 10000;
    public void display() {
     System.out.println("Outer: " + OuterClass.this.count);
     System.out.println("Inner: " + count);
     }
  }
}
```

Внутренние классы. Inner (нестатические)

- Правила для внутренних классов

2) конструктор внутреннего класса можно создать явным способом

ссылка_на_внешний_объект.new
конструктор_внутренего_класса([параметры]);

Вложенные классы. Nested (статические). Example 50

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

```
package _java._se._01._innerclasses.nested.outer1;

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

Вложенные классы. Nested (статические). Example 51

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

```
package _java._se._01._innerclasses.nested.outer2;
public class Outer {
     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
               Outer out = new Outer();
               System.out.println("out.x=" + out.x);
          }
     }
}
```

Вложенные классы. Nested (статические). Example 52

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

```
package _java._se._01._innerclasses.nested.outer3;
public class Outer {
     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");
          }
     }
}
```

```
package _java._se._01._innerclasses.nested.outer3;
public class Outer2 {
     public static void main(String[] args) {
          Outer.Inner1.method();
     }
}
```

Вложенные классы. Nested (статические). Example 53

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

```
package _java._se._01._innerclasses.nested.outer4;
public class Outer {
     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);
          }
     }
}
```

Вложенные классы. Nested (статические). Example 53

```
package _java._se._01._innerclasses.nested.outer4;
public class Outer2 extends Outer.Inner1 {
     public static void main(String[] args) {
     }
     public void outer2Method() {
          // System.out.println("x="+x); // ERROR
     }
}
```

Вложенные классы. Nested (статические). Example 54

- Класс, вложенный в интерфейс, статический по умолчанию

```
package _java._se._01._innerclasses.nested.outer5;
public interface MyInterface {
     int x = 10;
     class InnerInInterface {
          public void meth() {
               System.out.println("x=" + x);
          }
     }
}
```

Вложенные классы. Nested (статические). Example 55

- Вложенный класс может быть базовым, производным, реализующим интерфейсы

```
package _java._se._01._innerclasses.nested.outer6;
public class Outer {
     private static int x = 10;
     public static void main(String[] args) {
          Inner2.methodInner1();
          Inner2.methodInner2();
          Inner3.methodInner1();
          Outer out = new Outer();
          out.meth();
     }
     public void meth() {
          Inner3 in3 = new Inner3();
          in3.methodInner3();
          in3.methodInner1();
     }
     static class Inner1 {
          public static void methodInner1() {
               System.out.println("inner1 outer.x=" + x);
          }
     }
```

Вложенные классы. Nested (статические). Example 55

```
    static class Inner2 extends Inner1 {
        public static void methodInner2() {
            methodInner1();
            System.out.println("inner1 outer.x=" + x);
        }
    }
    class Inner3 extends Inner1 implements MyInterface {
        public void methodInner3() {
            methodInner1();
            System.out.println("inner1 outer.y=" + y);
            System.out.println("inner1 outer.x=" + x);
        }
    }
}
interface MyInterface {
    int y = 123;
}
```

Анонимные классы. Anonymous

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

Анонимные классы. Anonymous. Example 56

```
package _java._se._01._innerclasses.anonymous.outer1;
public class MyClass {
     public void print() {
          System.out.println("This is Print() in MyClass");
     }}

class MySecondClass {
     public void printSecond() {
          MyClass myCl = new MyClass() {
               public void print() {
                    System.out.println("!!!!!!!");
                    newMeth();
               }
               public void newMeth() {
                    System.out.println("New method");
               }
          };
          MyClass myCl2 = new MyClass();
          myCl.print();// myCl.newMeth(); // Error
          myCl2.print();
     }
     public static void main(String[] args) {
          MySecondClass myS = new MySecondClass();
          myS.printSecond();
     }}
```

Анонимные классы. Anonymous. Example 57

- Объявление анонимного класса выполняется одновременно с созданием его объекта с помощью операции new

```
package _java._se._01._innerclasses.anonymous.outer1;
public class MySecondClass {
     public void printSecond() {
          System.out.println("MySecondClass.java::printSecond");
     }
     public static void main(String[] args) {
          MySecondClass myS = new MySecondClass() {
               public void printSecond() {
                    System.out.println("Oi-oi-oi");
               }
          };
          myS.printSecond();
          new MySecondClass() {
               public void printSecond() {
                    System.out.println("Ai-ai-ai");
               }
          }.printSecond();
     }
}
```

Анонимные классы. Anonymous

- Конструкторы анонимных классов ни определить, ни переопределить нельзя

Анонимные классы. Anonymous. Example 58

```
package _java._se._01._innerclasses.anonymous.outer2;
import _java._se._01._innerclasses.anonymous.outer1.MyClass;
public class MySecondClass {
     public MySecondClass() {
          System.out.println("Constructor");
     }
     public void printSecond() {
          System.out.println("MySecondClass.java::printSecond");
     }
     public static void main(String[] args) {
          new MySecondClass() {
               // public MySecondClass() {} // ERROR
               // public MySecondClass(String str){} // ERROR
               public void printSecond() {
                    System.out.println("Ai-ai-ai");
                    new MyClass() {
                         public void print() {
System.out.println("print in MyClass in printSecond in MySecondClass");
                         }
                    }.print();
               }
          }.printSecond();
     }
}
```

Анонимные классы. Anonymous. Example 59

- Анонимные классы допускают вложенность друг в друга

```
package _java._se._01._innerclasses.anonymous.outer3;
public class MyClass {
     public void print() {
          System.out.println("This is Print() in MyClass");
     }
}
```

Анонимные классы. Anonymous. Example 59

```
package _java._se._01._innerclasses.anonymous.outer3;
public class MySecondClass {
     public void printSecond() {
          System.out.println("MySecondClass.java::printSecond");
     }
     public static void main(String[] args) {
          new MySecondClass() {
               public void printSecond() {
                    System.out.println("Ai-ai-ai");
                    new MyClass() {
                         public void print() {
System.out.println("print in MyClass in printSecond in MySecondClass");
                         }
                    }.print();
               }
          }.printSecond();
     }
}
```

Анонимные классы. Anonymous. Example 60

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

```
package _java._se._01._innerclasses.anonymous.outer4;
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;
}
```

Анонимные классы. Anonymous. Example 60

```
public class TestColor {
    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());
    }
}
```

ДОКУМЕНТИРОВАНИЕ КОДА (JAVADOC)

Javadoc. Основание для ведения документации

- Возобновление работы над проектом после продолжительного перерыва
- Переход проекта от одного человека (группы) к другому человеку (группе)
- Опубликование проекта для Open Source сообщества
- Совместная работа большой группы людей над одним проектом

Javadoc. Требования к документам

- Не документировать очевидные вещи (setter'ы и getter'ы, циклы по массивам и листам, вывод логов и прочее)

```
package java.se._01.javadoc;
public class DocRequirement {
	/**  Проверка: редактируема ли даннная ячейка.
	*
	*    <p>В случае если данная ячейка редактируема - возвращается true</p>
	*
	*    <p>В случае если данная ячаейка не редактируема - возвращается
false</p>
	*
	*    @param column номер колонки для проверки
	*    @return результат проверки
	**/
	public boolean isCellEditable(int column) {
		return column % 2 == 0 ? true : false;
	}
}
```

Javadoc. Требования к документам

- Поддерживать документацию в актуальном состоянии

```
package java.se._01.javadoc;
public class Parsing {
      /**
      * Произвести парсинг истории операций над невстроенной БД.
      * @throws XMLConfigurationParsingException
      **/
      public void parseHistoryNotEmbeddedDB()  throws
XMLConfigurationParsingException {
      return;
      /*    InputStream is =
      *     Thread.currentThread().GetContextClassLoader().
      getResourceAsStream("ru/magnetosoft/magnet/em/cfg/db-configuration-not-embedd
ed.xml");
      *     String configXml = readStringFromStream(is);
      *     XmlConfigurationParserImpl parser = new
      *     XmlConfigurationParserImpl(configXml); IEmConfiguration res =
      *     parser.parse(); assertNotNull(res);
      *     assertFalse(res.getOperationHistoryStorageConfiguration().isEmbedded();
      *     assertEquals("HSQLDB",
      *     res.getOperationHistoryStorageConfiguration().getStorageDBType());
      */
      }
}
```

Javadoc. Требования к документам

- Описывать входящие параметры, если нужно

```
package java.se._01.javadoc;
public class EnterParamsDoc {
     /**  Создание нового экземпляра ядра.
     *
     *    @param contextName
     *    @param objectRelationManager
     *    @param xmlObjectPersister
     *    @param ohm
     *    @param snm
     *    @param initializationLatch
     *    @return
     */
     public static EmEngine newInstance(String contextName,
          IXmlObjectRelationManager objectRelationManager,
          IXmlObjectPersister xmlObjectPersister,
          OperationHistoryManager ohm,
          ISearchNotificationManager snm,
          CountDownLatch initializationLatch) {
     ...
     }
}
```

Javadoc. Синтаксис javadoc-комментария

- Обыкновенный комментарий

```
/* Calculates the factorial */
int factorial(int x) {
…
```

- Javadoc-комментарий (он может включать в себя HTML тэги и специальные javadoc тэги, которые позволяют включать дополнительную информацию и ссылки)

```
/** Calculates the factorial */
public double factorial(int x) {
```

Javadoc. Структура javadoc-комментария

- Структура каждого javadoc-комментария такова:
  - первая строчка, которая попадает в краткое описание класса (отделяется точкой и пустой строкой);
  - основной текст, который вместе с HTML тэгами копируется в основную документацию;
  - входящие параметры (если есть);
  - выбрасываемые исключения (если есть);
  - возвращаемое значение (если есть);
  - служебные javadoc-тэги.

Javadoc. Структура javadoc-комментария

```
package java.se._01.javadoc;

import java.se._01.javadoc.exception.EntityManagerException;
import java.se._01.javadoc.exception.XmlMagnetException;

public class WriterDocExample {
      /**
      * Произвести запись нового объекта.
      *
      *     Произвести запись нового объекта. Тип для сохранения может быть
      *     подклассом List (для реализации возможности работы с несколькими
      *     объектами) или единичным объектом. В случае если произошла какая-либо
      *     ошибка - выбрасывается исключение. В данном случае с базой не происходит
      *     никаких изменений и ни один объект не  затрагивается пред
      *     операцией. Конкретный тип ошибки можно определить проверкой
      *     возвращенного исключения.
      *     @param object
      *     сохраняемый объект/объекты.
      *     @return сохраненный объект/объекты
      *     @throws XmlMagnetException
      *     @throws EntityManagerException
      */
      public Object insert(Object object) throws XmlMagnetException,
EntityManagerException{
            return new Object();
      }
}
```

Method Summary

java.lang.Object	insert(java.lang.Object object) Произвести запись нового объекта.

Method Detail

insert

```
public java.lang.Object insert(java.lang.Object object)
                        throws java.se._01.javadoc.exception.XmlMagnetException,
                               java.se._01.javadoc.exception.EntityManagerException
```

Произвести запись нового объекта. Произвести запись нового объекта. Тип для сохранения может быть подклассом List (для реализации возможности работы с несколькими объектами) или единичным объектом. В случае если произошла какая-либо ошибка - выбрасывается исключение. В данном случае с базой не происходит никаких изменений и ни один объект не затрагивается пред операцией. Конкретный тип ошибки можно определить проверкой возвращенного исключения.

Parameters:

object - сохраняемый объект/объекты.

Returns:

сохраненный объект/объекты

Throws:

java.se._01.javadoc.exception.XmlMagnetException
java.se._01.javadoc.exception.EntityManagerException

Javadoc. Типы тегов

- **Блочные теги**
  - Начинается с @tag и оканчивается с началом следующего тега
  - Пример

    @param x a value

- **Строчные теги**
  - Ограничены фигурными скобками
  - Могут встречаться в теле других тегов
  - Пример

    Use a {@link java.lang.Math#log} for positive numbers.

Javadoc. Тег @param

- Описывает параметров методов и конструкторов
- Синтаксис

  @param <имя параметра> <описание>
- Пример

  @param x a value

Javadoc. Тег @return

- Описывает возвращаемое значение метода
- Синтаксис

  @return <описание>
- Пример

  @return the factorial of <code>x</code>

Javadoc. Тег @throws

- Описывает исключения, генерируемые методом/конструктором
- Синтаксис

  @throws <класс исключения> <описание>
- Пример

  @throws IllegalArgumentException if <code>x</code> is less than zero

Javadoc. Тэг @see

- Ссылка на дополнительную информацию
- Синтаксис
  - @see <имя класса>
  - @see [<имя класса>]#<имя члена>
  - @see "<Текст ссылки>"
- Примеры
  - @see Math#log10
  - @see "The Java Programming language Specifiecation, p. 142"

Javadoc. Тэг @version

- Текущая версия класса/пакета
- Синтаксис

  @version <описание версии>
- Пример

  @version 5.0

Javadoc. Тег @since

- Версия в которой была добавлена описываемая сущность
- Синтаксис

  @since <описание версии>
- Пример

  @since 5.0

Javadoc. Тэг @deprecated

- Помечает возможности, которые не следует использовать
- Синтаксис

  @deprecated <комментарий>
- Пример

  @deprecated replaced by {@link #setVisible}

Javadoc. Тэг @author

- Описывает автора класса/пакета
- Синтаксис

  @author <имя автора>
- Пример

  @author Josh Bloch

  @author Neal Gafter

Javadoc. Тэг {@link}

- Ссылка на другую сущность
- Синтаксис

  {@link <класс>#<член> <текст>}
- Примеры

  {@link java.lang.Math#Log10 Decimal Logarithm}
  {@link Math}
  {@link Math#Log10}
  {@link #factorial() calculates factorial}

Javadoc. Тэг {@docRoot}

- Заменяется на ссылку на корень документации
- Синтаксис

  {@docRoot}
- Пример

  Copyright

Javadoc. Тэг {@value}

- Заменяется на значение поля
- Синтаксис

  {@value <имя класса>#<имя поля>}
- Пример

  Default value is {@value #DEFAULT_TIME}

Javadoc. Тэг {@code}

- Предназначен для вставки фрагментов кода
- Внутри тэга HTML не распознается
- Синтаксис

  {@code <код>}
- Пример

  Is equivalent of {@code Math.max(a, b)}.

Javadoc. Описание пакета

- Есть возможность применять комментарии для пакетов. Для этого необходимо поместить файл package.html в пакет с исходными текстами.
- Данный файл должен быть обычным HTML-файлом с тегом <body>.
  - Первая строчка файла до точки идет в краткое описание пакета, а полное идет вниз – под список всех классов и исключений.

Этот функционал позволяет описать что-то, что невозможно описать с помощью конкретных классов.

Javadoc. Применение тегов

<table>
<tr><th>Пакеты</th><th>Классы</th><th>Методы и конструкторы</th><th>Поля</th></tr>
<tr><td colspan="4">@see
@since
{@link}
{@docRoot}</td></tr>
<tr><td></td><td colspan="3">@deprecated</td></tr>
<tr><td colspan="2">@author
@version</td><td>@param
@return
@throws</td><td>{@value}</td></tr>
</table>

Javadoc. Наследование Javadoc

- Если какая-то часть информации о методе не указана, то описание копируется у ближайшего предка.

- Копируемая информация:
  - Описание
  - @param
  - @returns
  - @throws

Javadoc. Компиляция Javadoc

- Инструмент

  Javadoc

- Применение

  javadoc <опции> <список пакетов> <список файлов>

- Пример

  javadoc JavadocExample1.java

Javadoc. Основные опции Javadoc

-sourcepath <path>	Местоположения исходных фалов
-classpath <path>	Местоположение используемых классов
-d <dir>	Каталог для документации
-public	Подробность информации
-protected	
-package	
-private	
-version	Информация о версии
-author	Информация об авторе

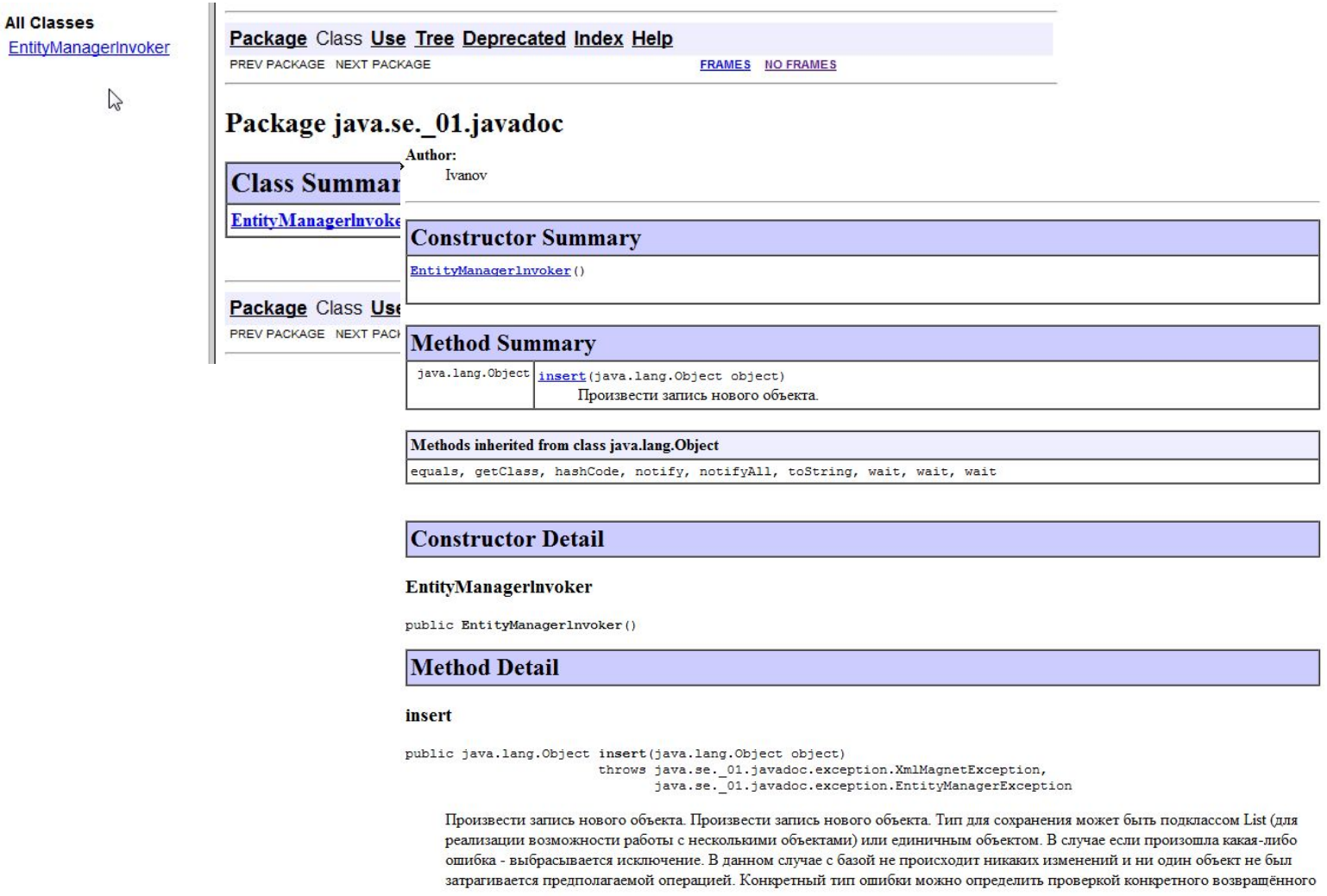
Javadoc. Example 61

```
package java.se._01.javadoc;
import java.se._01.javadoc.exception.EntityManagerException;
import java.se._01.javadoc.exception.XmlMagnetException;
/**   Представитель модуля EntityManger на клиентской стороне.
*
* <p>
*     Данный класс представляет средства доступ к возможностям модуля
*     EntityManager, минуя прямые вызовы веб -сервисов.
*     </p>
* <p>
*     Он самостоятельно преобразовывает ваши Java Beап'ы в XML и производит
*     обратную операцию, при получении ответа от модуля.
*     </p>
* <p>
*     Для получения экземпляра данного класса предназначены статические методы
*     {@link #getlnstance(InputStream)}  и {@link #getlnstance(String)}
*     </p>
*     Screated 09.11.2006
*     (Aversion $Revision 738 $
*     @author MalyshkinF
*     @since 0.2.2
*/
public class EntityManagerlnvoker {
```

Javadoc. Example 61

```
    /**
     * Произвести запись нового объекта.
     *
     * Произвести запись нового объекта. Тип для сохранения может быть
     *     подклассом List (для реализации возможности работы с несколькими
     *     объектами) или единичным объектом. В случае если произошла какая -либо
     *     ошибка - выбрасывается исключение. В данном случае с базой не
происходит
     *     никаких изменений и ни один объект не был затрагивается предполагаемой
     *     операцией. Конкретный тип ошибки можно определить проверкой конкретного
     *     возвращённого исключения.
     *     @param object
     *     сохраняемый объект/объекты.
     *     @return сохраненный объект/объекты
     *     @throws XmlMagnetException ошибка в процессе парсинга XML
     *     @throws EntityManagerException ошибка связанная с другой работой
клиента
     */
    public Object insert(Object object)  throws XmlMagnetException,
    EntityManagerException {  return new Object(); }
}
```

Javadoc. Example 61



All Classes

EntityManagerInvoker

Package Class Use Tree Deprecated Index Help

PREV PACKAGE NEXT PACKAGE FRAMES NO FRAMES

Package java.se._01.javadoc

Author:
Ivanov

Class Summar

EntityManagerInvoke

Package Class Use

PREV PACKAGE NEXT PACI

Constructor Summary

EntityManagerInvoker()

Method Summary

java.lang.Object	insert(java.lang.Object object) Произвести запись нового объекта.

Methods inherited from class java.lang.Object

equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

EntityManagerInvoker

```
public EntityManagerInvoker()
```

Method Detail

insert

```
public java.lang.Object insert(java.lang.Object object)
                        throws java.se._01.javadoc.exception.XmlMagnetException,
                               java.se._01.javadoc.exception.EntityManagerException
```

Произвести запись нового объекта. Произвести запись нового объекта. Тип для сохранения может быть подклассом List (для реализации возможности работы с несколькими объектами) или единичным объектом. В случае если произошла какая-либо ошибка - выбрасывается исключение. В данном случае с базой не происходит никаких изменений и ни один объект не был затрагивается предполагаемой операцией. Конкретный тип ошибки можно определить проверкой конкретного возвращённого

СПАСИБО ЗА ВНИМАНИЕ!

ВОПРОСЫ?

Java.SE.01

Java Fundamentals

Author: **Ihar Blinou, PhD**
Oracle Certified Java Instructor
Ihar_blinou@epam.com