java.lang **Оболочки примитивных типов**

Примитивные типы **byte, short, int, long, float, double, char, boolean** не являются частью объектной иерархии. Они передаются по значению в методы и не могут быть переданы по ссылке. Также нет способа для двух методов ссылаться на один и тот же экземпляр int - переменной.

В Java определены классы, которые соответствуют каждому из примитивных типов. Эти классы инкапсулируют примитивные типы в классы (помещают примитивные типы в оболочки классов).

Number

Абстрактный класс **Number** определяет суперкласс, который реализован классами, являющимися оболочками для числовых типов **byte, short, int, long, float и double**. **Number** имеет абстрактные методы, которые возвращают значение объекта в каждом из разных числовых форматов.
double doubleValue(), float floatValue(), byte byteValue(), int intValue(), long longValue() , short shortValue().
Значения, возвращаемые этими методами, могут быть округлены.
Number имеет шесть конкретных подклассов, которые содержат явные значения каждого из числовых типов:
Double, Float, Byte, Short, Integer, Long.

Double и **Float -** это оболочки для значений с плавающей точкой типов double и float соответственно.

Конструкторы для Float: **Float(double num), Float(float num)**
Float(String str) throws NumberFormatException.

Конструкторы для Double: **Double(double num),**
Double(String str) throws NumberFormatException

Объекты Double могут быть сконструированы из значения double или строки, содержащей значение с плавающей точкой.

MAX_VALUE	Максимальное положительное значение.
MIN_NORMAL	Минимальное положительное нормальное значение.
MIN_VALUE	Минимальное положительное значение.
NaN	Не число.
POSITIVE_INFINITY	Положительная бесконечность.
NEGATIVE_INFINITY	Отрицательная бесконечность.
SIZE	Размер помещенного в оболочку значения в битах.
TYPE	Объект Class для float и double.

Методы класса Double

byte byteValue() - возвращает значение вызывающего объекта как byte.

static int compare(double num1, double num2) - сравнивает значения numl и num2. Возвращает 0, если значения эквивалентны. Возвращает отрицательное значение, если num1 меньше num2. Возвращает положительное значение, если num1 больше num2.

int compareTo(Double d) - сравнивает числовое значение вызывающего объекта с d. Возвращает 0, если значения эквивалентны. Возвращает отрицательное значение, если вызывающий объект имеет меньшее значение. Возвращает положительное значение, если вызывающий объект имеет большее значение.

double doubleValue() - возвращает значение вызывающего объекта как double.

int intValue() - возвращает значение вызывающего объекта как int.

boolean equals(Object DoubleObj) - возвращает true, если вызывающий объект Double эквивалентен DoubleObj.

float floatValue() - возвращает значение вызывающего объекта как float.

boolean isInfinite() - возвращает true, если вызывающий объект содержит бесконечное значение. В противном случае возвращает false.

boolean isNaN() - возвращает true, если вызывающий объект содержит нечисловое значение. В противном случае возвращает false.

long longValue() - возвращает значение вызывающего объекта как long.

static double parseDouble(String str) throws NumberFormatException - возвращает double-эквивалент числа с основанием 10, содержащегося в строке str.

short shortValue() - возвращает значение вызывающего объекта как short

static String toString() - возвращает строковый эквивалент вызывающего объекта.

Методы класса Integer

byte byteValue()

int compareTo (Integer i)

double doubleValue()

boolean equals(Object IntegerObj)

float floatValue()

int intValue()

long longValue()

short shortValue()

String toString()

static int parseInt(String str) throws NumberFormatException

static int signum(int num) – возвращает -1, если num отрицательное, 0 - если ноль, и 1 - если положительное.

static String toBinaryString(int num) - возвращает строку, содержащую двоичный эквивалент num.

static Integer valueOf (int num) - возвращает объект Integer, содержащий значение, переданное в num.

static String toHexString (int num) - возвращает строку, содержащую шестнадцатеричный эквивалент num.

static String toOctalString(int num) - возвращает строку, содержащую восьмеричный эквивалент num.

```
class DoubleDemo {
 public static void main(String args[])  {
    Double dl = new Double(3.14159) ;
    Double d2 = new Double(“314159E-5“) ;
    System.out.println(dl + “ = “ + d2 + “ ->“ + dl.equals(d2));
      }   }

class InfNaN {
public static void main(String args[]) {
    Double d1 = new Double(1/0.);
    Double d2  = new Double(0/0.);
  System.out.println(d1+“:“+dl.islnfinite()+“,”+d1.isNaN());
  System.out.println(d2+“:“+d2.islnfinite()+“,“+“+d2.isNaN());
 } }
```

Вывод:

Infinity: true, false

NaN: false, true

```
class StringConversions {
public static void main(String args[]) {
    int num = 19648;
    System.out.println(num + " в бинарной: " +
                                        Integer.toBinaryString(num));
    System.out.println(num + " в восьмеричной: " +
                                        Integer.toOctalString(num));
    System.out.println(num + " в шестнадцатеричной: " +
                                         Integer.toHexString(num));
 } }
```

Вывод:

19648 в бинарной: 100110011000000
19648 в восьмеричной: 46300
19648 в шестнадцатеричной: 4cc0

Character - это простая оболочка для char.

Конструктор: **Character(char ch)**

ch - символ, который будет помещен в оболочку создаваемого объекта Character. Чтобы получить значение char из объекта Character используется метод **char charValue()**.

static boolean isDigit(char ch) - true, если ch - цифра, иначе - false.

static boolean isIdentifierIgnorable(char ch) - true, если ch должен быть проигнорирован в идентификаторе, иначе - false.

static boolean isLetter (char ch) - true, если ch - буква, иначе - false.

static boolean isLetterOrDigit(char ch) - true, если ch - буква или цифра, иначе - false.

static boolean isLowerCase(char ch) - true, если ch - буква в нижнем регистре.

static boolean isSpaceChar (char ch) - true, если ch - пробельный символ Unicode.

static boolean isUpperCase(char ch) - true, если ch символ верхнего регистра.

static boolean isWhitespace(char ch) - true, если ch пробельный символ.

static char toLowerCase(char ch) - возвращает эквивалент ch в нижнем регистре.

static char toUpperCase(char ch) - возвращает эквивалент ch в верхнем регистре.

```
class  IsDemo {
  public static void main(String args[]) {
     char a [] =  { ‘a’, ‘b’, ‘5’, '?', ‘A‘ , ‘ ’};
     for(int i=0; i<a.length; i++) {
        if( Character.isDigit(a[i]) )
             System.out.println(a[i] + “десятичное число.“ );
        if( Character.isLetter(a[i]) )
             System.out.println(a[i] + “буква.“ );
        if( Character.isWhitespace(a[i]) )
           System.out.println(a[i] +“пробельный символ.“ );
        if( Character.isUpperCase(a[i]) )
           System.out.println(a[i] + “символ верхнего регистра.“ );
        if( Character.isLowerCase(a[i]) )
           System.out.println(a[i] + “символ нижнего регистра.“ );
} } }
```

Вывод :

a - буква. a - символ нижнего регистра.
b - буква. b - символ нижнего регистра. 5 - десятичное число.
A - буква. A - символ верхнего регистра. - пробельный символ.

Boolean - это оболочка значений типа boolean.
Константы: **TRUE** и **FALSE**.
Конструкторы:
Boolean(boolean boolValue), где **boolValue -** либо true, либо false.
Boolean(String boolString) если **boolString** содержит строку **“true”**(в верхнем или нижнем регистре), то новый объект Boolean будет true, иначе false.

Автоупаковка и Автораспаковка

Автоупаковка - это процесс, посредством которого примитивный тип автоматически инкапсулируется (упаковывается) в эквивалентную ему оболочку типа всякий раз, когда требуется объект этого типа. Нет необходимости явного конструирования объекта.

Автораспаковка - это процесс, с помощью которого значение упакованного объекта автоматически извлекается (распаковывается) из оболочки типа, когда нужно получить его значение.

Не нужно использовать методы вроде **intValue()** или **doubleValue().**

С использованием автоупаковки нет необходимости в ручном конструировании объектов для оболочки примитивных типов. Нужно только присвоить значение ссылке оболочки типов.

Создание объекта Integer, который содержит значение 100:

Integer iOb = 100; // автоупаковка int в Integer

Не нужно создавать объект операцией new.

Чтобы распаковать объект:
int i = iOb; // автораспаковка

```
class AutoBox {
  public static void main(String args{]) {
        Integer  iOb = 100;  // автоупаковка int
        int  i  = iOb;  // автораспаковка Integer
        System.out.println (i + " " + iOb); // отображает 100 100
   } }
```

Автоупаковка происходит автоматически, когда примитивный тип должен быть преобразован в объект.
Автораспаковка происходит автоматически, когда объект должен быть преобразован в примитивный тип.
Таким образом, автоупаковка / автораспаковка может произойти, когда аргумент передается методу, либо когда значение возвращается из метода.

```
class AutoBox2 {
  static int m(Integer v) {          // принимает тип Integer
                  return v;  }        // автораспаковка  в int

 public static void main(String args[]) {
      Integer iOb = m(100);     // автоупаковка int в Integer
      System.out.println(iOb);
        }
     }
```

Автоупаковка / авторaспаковка происходит в выражениях. Внутри выражения числовой объект автоматически распаковывается. Выходной результат выражения при необходимости упаковывается заново.

```
class AutoBox3 {
  public static void main(String args[]) {
      Integer iOb, iOb2;
      int i;
      iOb = 100;
      System.out.println(“Исходное значение iOb: “+ iOb);
      ++iOb;    //авторaспаковка, увеличение,  автоупаковка
      System.out.println (“После  ++iOb:“  + iOb);
      iOb2 = iOb + (iOb / 3);
      i = iOb + (iOb / 3);
      System.out.println ("i после выражения: " + i);
}}
```

Исходное значение iOb: 100
После ++iOb: 101
iOb2 после выражения: 134
i после выражения: 134

Автоупаковка / авторaспаковка значений Boolean и Character

```
class AutoBox5 {
  public static void main(String args[]) {
     Boolean b = true;
              if(b)
              System.out.println(“b равна true“);
     Character ch = ‘x‘;
           char ch2 = ch;
            System.out.println(“ch2 равна “ + ch2);
  } }
```

Важный момент: авторaспаковка b внутри условного выражения if. Условное выражение, которое управляет if, должно при вычислении возвращать boolean. Значение boolean, содержащееся в b, автоматически распаковывается при вычислении условного выражения (аналогично while, for , …, do/while).

Класс **Void** имеет одно поле TYPE, которое содержит ссылку на объект Class для типа void. Экземпляры этого класса не создаются.

Абстрактный класс **Process** инкапсулирует процесс, то есть выполняющуюся программу.

Класс **Runtime** инкапсулирует среду времени выполнения. Создать объект Runtime невозможно. Можно получить ссылку на текущий объект Runtime, вызвав статический метод **Runtime.getRuntime()**. По ссылке можно вызвать несколько методов, контролирующих состояние и поведение виртуальной машины Java JVM).

Класс **Class** представляет характеристики класса. Когда JVM загружает файл .class, который описывает некоторый тип, в памяти создается объект класса Class, который будет хранить эти характеристики. Он хранит информацию о том, является ли объект интерфейсом, массивом или примитивным типом, каков суперкласс объекта, каково имя класса, какие в нем конструкторы, поля, методы и вложенные классы.

Управление памятью

Несмотря на то что Java предлагает автоматическую сборку мусора, иногда требуется узнать, насколько велика объектная куча и сколько свободной памяти осталось. Для получения этих значений используются методы :

long totalMemory()
long freeMemory()

Сборщик мусора java запускается периодически для утилизации неиспользуемых объектов. Однако иногда может потребоваться собрать отброшенные объекты до того, как сборщик мусора будет запущен. Запуск по требованию - метод **void gc( )** .

```
class MemoryDemo {
 public static void main(String args[]) {
   Runtime r = Runtime.getRuntime();
   long mem1, mem2;
   Integer someints[] = new Integer[1000];
   System.out.println(“ Всего памяти: “  + r.totalMemory());
   mem1 = r.freeMemory();
   System.out.println(“Свободной памяти вначале: “ + mem1);
   r.gc () ;
   mem1 = r.freeMemory();
   System.out.println("Свободной памяти после сборки мусора: " + mem1);
   for(int i=0; i<1000; i++)
        someints[i] = new Integer(i);
   mem2 = r.freeMemory();
   System.out.println("Свободной памяти после распределения: " +mem2);
     for(int i=0; i<1000; i++)
           someints[i] = null;
   r.gc();
  mem2= r.freeMemory();
  System.out.println("Свободной памяти после сборки" + " отброшенных
                                                    Integer: " + mem2); } }
```

Всего памяти: 16318464
Свободной памяти вначале: 16030712
Свободной памяти после сборки мусора: 16327456
Свободной памяти после распределения: 16234336
Свободной памяти после сборки отброшенных Integer: 16327456

Цикл FOR

1. for (i=0; i < n; i++)
2. for (i=1, j=4;   i < j; i++, j--)
3. for(i=1; !done ; i++) {.. done=true;..}
4. for(; !done;) => while
5. for(;;)
6. **“for-each”** (для каждого)

for (итерац_ переменная: коллекция или массив)

Итерационная переменная будет последовательно принимать каждое значение из коллекции или массива. Тип итерационной переменной должен совпадать с типом элементов в коллекции (массиве).

```
int nums[]=new int[10];
…
for (int x : nums)
        x=nums[i];
```

Varargs – аргументы переменной цены

variable-length arguments – аргументы переменной длины.

Метод, который принимает переменное число аргументов, называется ***методом переменной арности***.

static void VarTest (int…r) { }

Такая синтаксическая конструкция указывает компилятору, что метод **VarTest** может вызываться с нулем или более аргументов. Внутри метода **VarTest** доступ к аргументам осуществляется с использованием синтаксиса обычного массива.

Аргументы будут храниться в массиве, на который ссылается переменная **r**. Аргументы автоматически помещаются в массив и передаются переменной **r**. В случае отсутствия аргументов длина массива равна нулю.

```
class VarArgs {

static void VarTest (int ... r) {
        System.out.println("Количество аргументов"+
                                r.lenght  +  "Содержимое:");
        for (int x : r)
                System.out.println(x+"   ");   }

public static void main (String args[]){
        VarTest(10);        // 1 аргумент
        VarTest (1,2,3);    // 3 аргумента
        VarTest();          // без аргументов
    }
}
```

Вместе с параметром переменной длины массив может содержать обычные параметры.

НО! параметр переменной длины должен быть последним параметром метода. Например:

int Method(int a, int b, double c, int ... vars) { }

В данном случае первые три аргумента в вызове метода соответствуют первым трем параметрам, все остальные аргументы принадлежат параметру **vars**.

Еще одно ограничение: метод должен содержать только один параметр типа varargs.

Varargs – аргументы переменной цены

Такие методы можно перегружать:
static void VarTest (int…r)
static void VarTest (boolean…r)
static void VarTest (String msg, int…r)

При перегрузке таких методов следует избегать неопределенности при вызове. В частности, если в программе будет вызов метода без VarTest() параметров, т.к. компилятору не понятно к какому типу преобразовывать массив int или boolean.

Или:
static void VarTest (int…r)
static void VarTest (int n, int…r)
Компилятор не может разрешить вызов: VarTest(l),
т.к. не знает преобразовывать к VarTest(int…) с одним аргументом переменной длины или к VarTest(int, int...) без аргумента переменной длины.

Перечисления

В Java перечисления являются типом класса. Перечисления могут иметь конструкторы, методы и переменные экземпляры. Перечисления создаются использованием ключевого слова **enum.**

enum Apple { Jonathan, Antonovka, GoldenDel, Anisovka }

Идентификаторы **Jonathan, GoldenDel** и др. называются константами перечисления. Каждая из них явно объявлена как **public static final** член класса Apple. Тип констант - это тип перечисления, в котором они объявлены. В данном случае - это Apple. В языке Java эти константы называются ***самотипизированными.***

Объявив перечисление, можно создавать переменные этого типа, нельзя создавать объекты этого типа с помощью операции new. Нужно просто объявить переменную перечисления.

Например: **Apple ap;**

Поскольку ap имеет тип Apple, присвоить ей можно только те значения, которые определены в перечислении: **ap = Apple.Antonovka**;

Две перечислимых константы можно проверять на равенство

if (ap == Apple.GoldenDel) ...

Перечислимые значения также могут быть использованы в операторе switch.

```
class EnumDemo {
public static void main(String args[]) {
  Apple ap;
  ap = Apple.Antonovka;
  System.out.println("Значение ap: " + ap);
  System.out.println();
  ap = Apple.GoldenDel;
  if(ap == Apple.GoldenDel)
  System.out.println(“ap содержит GoldenDel“);
switch (ap) {
  case  Jonathan: System.out.println(“Jonathan красный“ );  break;
  case GoldenDel: System.out.println(“Golden Delicious желтый“ ); break;
  case Antonovka: System.out.println(“ Antonovka  желтый“ ); break;
  case Anisovka: System.out.println(“Anisovka  желтый“); break;
} } }
```

Вывод:
Значение ap: Antonovka
ap содержит GoldenDel.
Golden Delicious желтый.

Перечисления автоматически включают два предопределенных метода:
public static enum_type[] values() - возвращает массив, содержащий список констант перечисления.
public static enum_type valueOf(String str) - возвращает константу перечисления, чье значение соответствует **str**.
enum_type - это тип перечисления.

```
enum  Apple { Jonathan,  Antonovka,  GoldenDel,  Anisovka }
class EnumDemo2 {
  public static void main(String args[]) {
      Apple ap;
      System.out.println(“Константы Apple:“);
      Appel allapples[] = Apple.values();
      for(Apple a : allapples)
            System.out.print(“  ” + a);
      ap = Apple.valueOf (“Anisovka“ ) ;
      System.out.println ("ap содержит" + ap);
} }
```

Вывод :
Константы Apple: Jonathan, Antonovka, GoldenDel, Anisovka
ap содержит Anisovka

Перечисление в Java - это тип класса.

Хотя нельзя создать экземпляр enum с помощью операции new, в остальном перечисление обладает всеми возможностями, которые имеются у других классов.

Можно предоставлять перечислениям конструкторы, добавлять переменные экземпляров и методы, реализовывать интерфейсы.

Важно понимать, что каждая константа перечисления является объектом его типа перечисления. Т.е., когда определяется конструктор для enum, он вызывается при каждом создании константы перечисления.

Каждая константа перечисления имеет свою собственную копию переменных перечисления.

```
enum  Apple { Jonathan(10),  Antonovka(15),  GoldenDel(20),  Anisovka(25) ;
   private int price;
        Apple (int p) {
              price = p;   }
       int getPrice() {
              return price;  }      }

class EnumDemo {
 public static void main(String args[]) {
  Apple ap;        //констуктор вызывается для каждой константы
  System.out.println(“ Jonathan стоит“ + Apple. Jonathan.getPrice() + “ руб.”);
 System. out.println (“Все цены яблок:“);
 for(Apple a : Apple.values())
      System.out.println(a + “ стоит“ + a.getPrice() + “ руб.“ ); } }
```

Вывод:
Jonathan стоит 10 центов.
Все цены яблок:
Jonathan стоит 10 руб.
Antonovka стоит 15 руб.
GoldenDel стоит 20 руб.
Anisovka стоит 25 руб.

Перечисления наследуются от класса **Enum** (java.lang.Enum). Этот класс определяет несколько методов, доступных к использованию перечислениями.

final int ordinal() - возвращает порядковое значение вызывающей константы. Порядковые значения начинаются с нуля. (в перечислении Apple.Johnatan имеет порядковое значение 0).

final int compareTo(type_enum e) - сравнить порядковые значения двух констант перечисления (**type_enum** - тип перечисления, а **e** - константа, которую нужно сравнить с вызывающей константой). Вызывающая константа и **e** должны относиться к одному перечислению. Если вызывающая константа имеет порядковое значение меньше чем **e**, то возвращается отрицательное значение. Если два порядковых значения одинаковы, возвращается 0. Если вызывающая константа имеет порядковое значение больше чем **e**, то возвращается положительное значение.

Можно сравнить на эквивалентность перечислимую константу с любым другим объектом, используя **equals()**. Два объекта будут эквивалентны только в случае, если оба они являются ссылкой на одну и ту же константу из одного и того же перечисления.

Java включает два класса для работы с «длинной арифметикой»: **BigInteger** и **BigDecimal**. Пакет **java.math**

BigInteger поддерживают целые числа произвольной точности. (можно точно представить значение целого числа любого размера без потерь любой информации во время операций).

BigDecimal для дробных чисел произвольной точности.

```
BigInteger big1 = new BigInteger("11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111");

BigInteger big2 = new BigInteger("22222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222");

BigInteger result = big1.add(big2);
System.out.println(big1 + " + " + big2 + " = " + result);
```

Класс **BigDecimal**

Конструкторы:

BigDecimal(double val) – объект из числа типа double;

BigDecimal(String str) – из числа, представленного строкой **str;**

BigDecimal(BigInteger val) – на вход объект BigInteger;

BigDecimal(BigInteger val, int scale) - число по формуле

val × 10^{-scale}, scale – кол-во цифр после запятой.

Методы:

BigDecimal **abs()** – модуль;

BigDecimal **add(BigDecimal val)** – сложение this с val;

int **compareTo(BigDecimal val)** – сравнение this с val;

BigDecimal **divide(BigDecimal val)** – деление this на val;

boolean **equals(Object x) -** проверка на равенство this и x;

BigDecimal **multiply(BigDecimal val)** – умножение this на val;

BigDecimal **negate()** – отрицательное значение this

BigDecimal **pow(int n)** – степень **this^{n}**.

Класс **BigInteger**

Конструкторы:

BigInteger(byte[] val) – из массива типа byte (каждая цифре числа соответствует val[i]);

BigInteger(String str) – из числа, представленного строкой **str;**

Методы:

BigInteger	**abs()** – модуль;
BigInteger	**add(BigInteger val)** – сложение с val;
BigInteger	**and(BigInteger val)** – побитовая операция **this & val**;
int	**compareTo(BigInteger val**) – сравнение с val:
BigInteger	**divide(BigInteger val)** – деление на val;
boolean	**equals(Object x)** – проверка на равенство;
BigInteger	**gcd(BigInteger val)** – НОД this и val;
BigInteger	**mod(BigInteger m)** - остаток от деления на m;
BigInteger	**negate()** – отрицательное значение;
BigInteger	**not()** – побитовое НЕ (отрицание **~this**);
String	**toString()** – строковый эквивалент;
BigInteger	**multiply(BigInteger val)** – умножение на val;
BigInteger	**pow(int n)** – степень **this^n.**