EPAM Systems, RD Dep.

Конспект и раздаточный материал

JAVA.SE.01 Java Fundamentals

REVISION HISTORY					
Ver.	**Description of Change**	**Author**	**Date**	**Approved**	
				Name	**Effective Date**
<1.0>	Первая версия	Игорь Блинов	<04.08.2011>		
<2.0>	Вторая версия. Конспект переделан под обновленное содержание материала модуля.	Ольга Смолякова	<09.12.2013>		

СОДЕРЖАНИЕ
JAVA.SE.01 JAVA FUNDAMENTALS

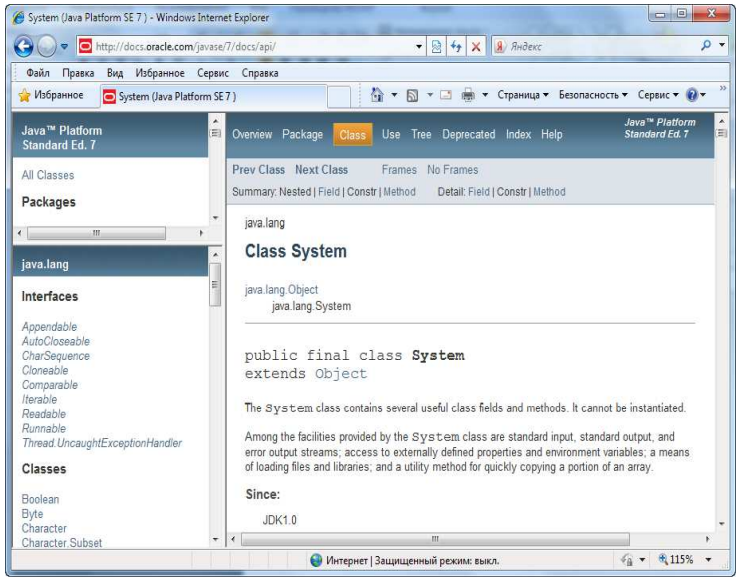
Введение в язык Java

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

Java language specification – техническое описание синтаксиса и семантики языка программирования Java.

Application program interface (API), или library (библиотека), содержит предопределенные классы и интерфейсы, используемые для разработки Java-программ.

Документация Java - http://docs.oracle.com/javase/7/docs/api/



Редакции Java

Java Standard Edition (Java SE) – разработка самостоятельных приложений на стороне клиента или апплетов.

Java Enterprise Edition (Java EE) – разработка приложений на стороне сервера, таких как сервлеты, JSP, JSF.

Java Micro Edition (Java ME) – разработка приложений для мобильных устройств.

JVM, JRE, JDK

JVM - это Java Virtual Machine, виртуальная машина Java, часть программного обеспечения Java, интерпретирующая байт-код, описываемый в класс-файлах.

JRE - это Java Runtime Environment, среда выполнения Java, предназначена только для запуска готовых Java-приложений, а потому содержит лишь реализацию виртуальной машины и набор стандартных библиотек.

JDK - это Java Development Kit, средство разработчика Java, включающее в себя набор утилит, стандартные библиотеки с их сходным кодом и набор демонстрационных

примеров. Утилиты включают в себя:

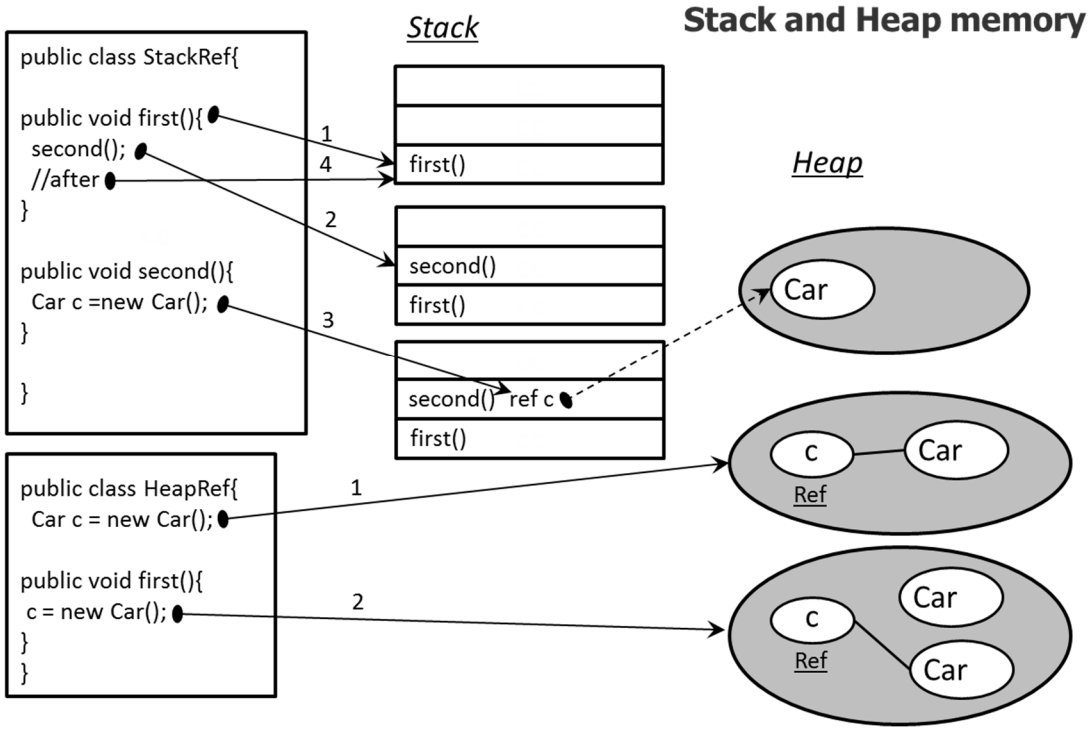
- *java* - реализация JVM;
- *javac* - компилятор Java;
- *appletviewer* - средство для запуска апплетов;
- *jar* - архиватор формата JAR;
- *javadoc* - утилита для автоматической генерации документации;
- и др.

Использование памяти

Управление памятью непосредственно обеспечивается JVM.

В Java все объекты программы расположены в **динамической памяти** (heap) и доступны по **объектным ссылкам**, которые в свою очередь хранятся в *стеке* (non-heap).

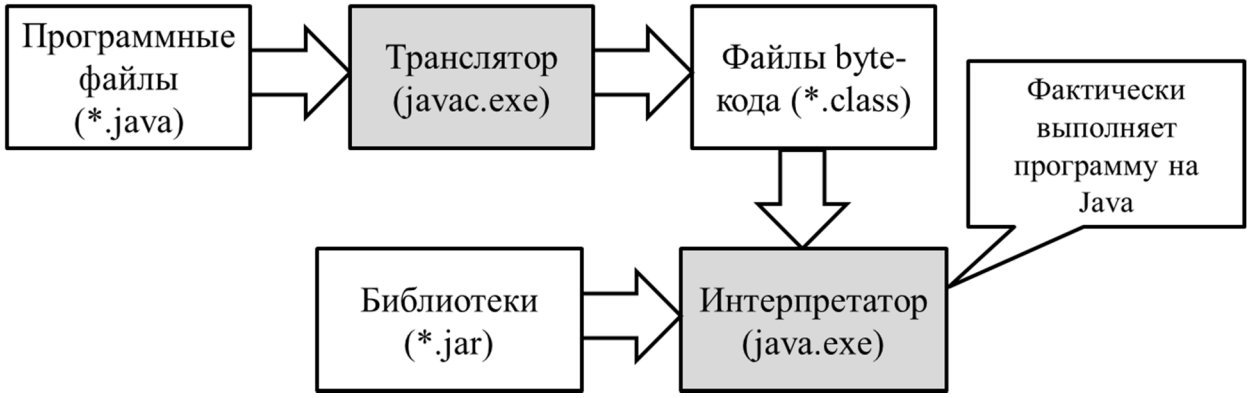
Необходимо отметить, что объектные ссылки языка Java *содержат информацию о классе* объектов, на которые они ссылаются, так что объектные ссылки - это не указатели, а дескрипторы объектов. Наличие дескрипторов позволяет JVM выполнять проверку совместимости типов на фазе интерпретации кода, генерируя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



Память, выделяемая *для ссылок*, **управляется автоматически**, как и память для примитивных типов. **Память** для каждого **объекта** при помощи этой операции *new*.

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

Жизненный цикл программ на Java



Простое линейное приложение

```
package _java._se._01._start;
public class First {
	public static void main(String[] args){
		System.out.print("Java ");
		System.out.println("уже здесь!");
	}
}
```

Простое объектно-ориентированное приложение

```
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();
	}
}
```

Компиляция и запуск приложения из командной строки

Разместите код в каталоге <workdir>/src

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

Скомпилируйте программу командой

javac -d bin src\Console.java

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

Запуск программы из консоли осуществляется командой

java -cp ./bin _java._se._01._start.Console

Работа с аргументами командной строки

```
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

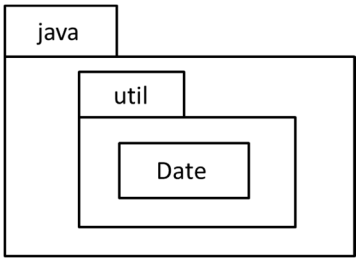
java.exe _java._se._01.start.CommandArg first second 23 56.2 23,9

Пакеты

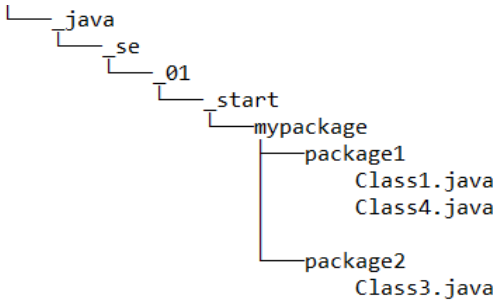
Пакеты – это контейнеры классов, которые используются для разделения пространства имен классов.

Пакет в Java создается включением в текст программы первым оператором ключевого слова package.

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



Для хранения пакетов используются *каталоги файловой системы*.



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

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

При компиляции поиск пакетов осуществляется в:

- рабочем каталоге;
- используется параметр переменной среды classpath;
- указывается местонахождение пакета параметром компилятора –classpath.

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

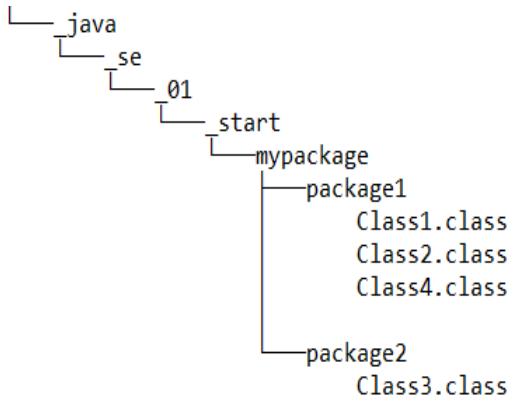
Import

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

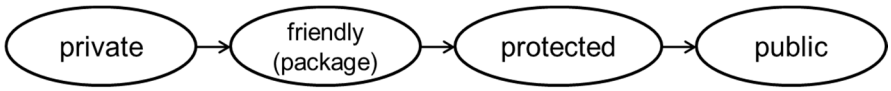
```
import имя_пакета.имя_подпакета.*;
import имя_пакета.имя_подпакета.имя_подпакета.имя_класса;
```

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

class Class2{
}
package _java._se._01._start.mypackage.package2;
import _java._se._01._start.mypackage.package1.Class1;
public class Class3 {
	public static void main(String[] args) {
		Class1 cl1 = new Class1();
	}
}
package _java._se._01._start.mypackage.package1;
public class Class4 {
	Class2 obj = new Class2();

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

Структура и содержимое пакетов при компиляции:



Модификаторы доступа




```
package _java._se._01._start;
public class Modificators {
	public static void main(String[] args) {
		AboutJava obj = new AboutJava();
		String s1 = obj.getAboutJava();
		String s2 = obj.info(); // error
	}
}
class AboutJava{
	public String getAboutJava(){
		return info();
	}
	private String info(){
		return "About Java.";
	}
}
```

Типы данных, переменные

Типы данных, переменные. Примитивные типы

Примитивные 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**.

Синтаксис Java позволяет создавать свои типы, получившие название ссылочных

Сравнение примитивных типов

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

Особенности примитивных типов

- Размер примитивных типов одинаков для всех платформ; за счет этого становится возможной переносимость кода
- Размер boolean неопределен. Указано, что он может принимать значения true или false.
- Преобразования между типом boolean и другими типами не существует.

Разрядность типов с плавающей точкой.

Тип	Бит	Знак	Мантисса	Порядок
float	32	1	23	8
double	64	1	52	11

Количество значащих цифр

Тип данных	Размер, бит	Количество значащих цифр
int	32	10
long	64	19
float	32	7
double	64	15

Значения по умолчанию

Неинициализированная явно переменная (!член класса или член экземпляра класса) примитивного типа принимает значение по умолчанию 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

Переменные

Характеристики переменных:

- 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.

Основная форма объявления:

```
тип идентификатор [ = значение];
```

При объявлении переменные могут быть проинициализированы

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.

Объявление переменных

```
package _java._se._01._types;
public class VariablesExample {
	boolean statusOn;
	double javaVar = 2.34;
	public static void main(String[] args) {
		int itemsSold = 04;
		double salary = 1.234e3;
		float itemCost = 11.0f;
		int i = 0xFd45, k$;
		double _interestRate;
	}
	public void javaMethod() {
		long simpleVar = 1_000_000_000_000L;
		byte byteVar2 = 123;
	}
}
```

Особенности работы с переменными

- Java не позволяет присваивать переменной значение более длинного типа.

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

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

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

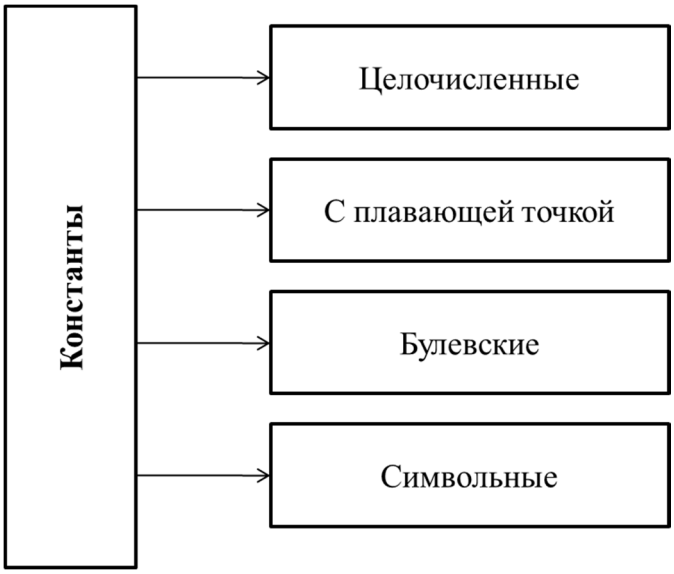
```
int intVar = 100;
long longVar = 10000000000000L;
intVar += longVar;

// Error
intVar = intVar + longVar;
```

Ключевые и зарезервированные слова языка Java

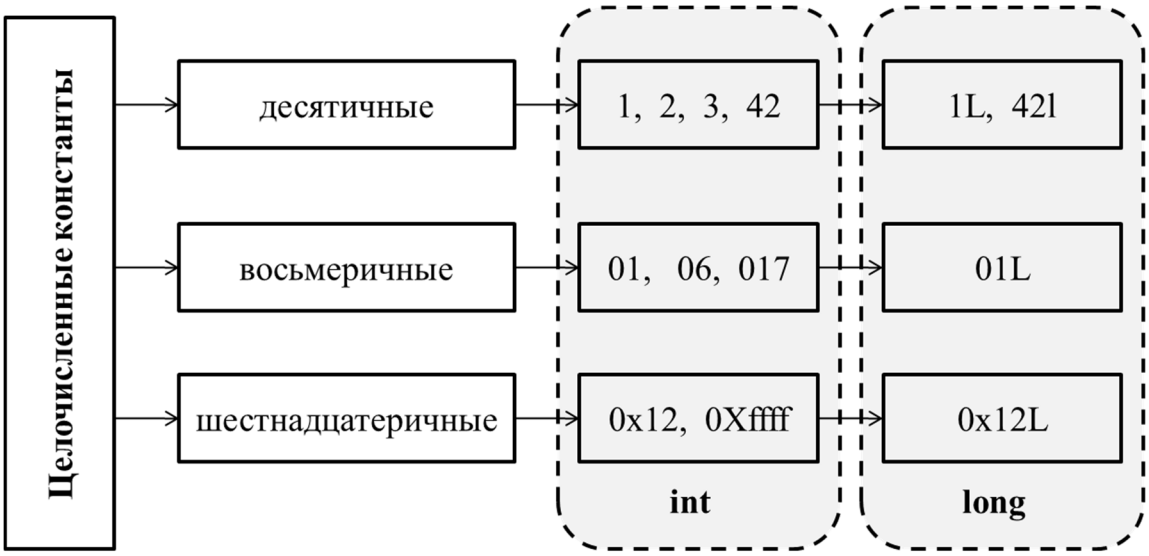
abstract	**continue**	**for**	**new**	**switch**
assert	**default**	**goto**	**package**	**synchronized**
boolean	**do**	**if**	**private**	**this**
break	**double**	**implements**	**protected**	**throw**
byte	**else**	**import**	**public**	**throws**
case	**enum**	**instanceof**	**return**	**transient**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

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

Литералы

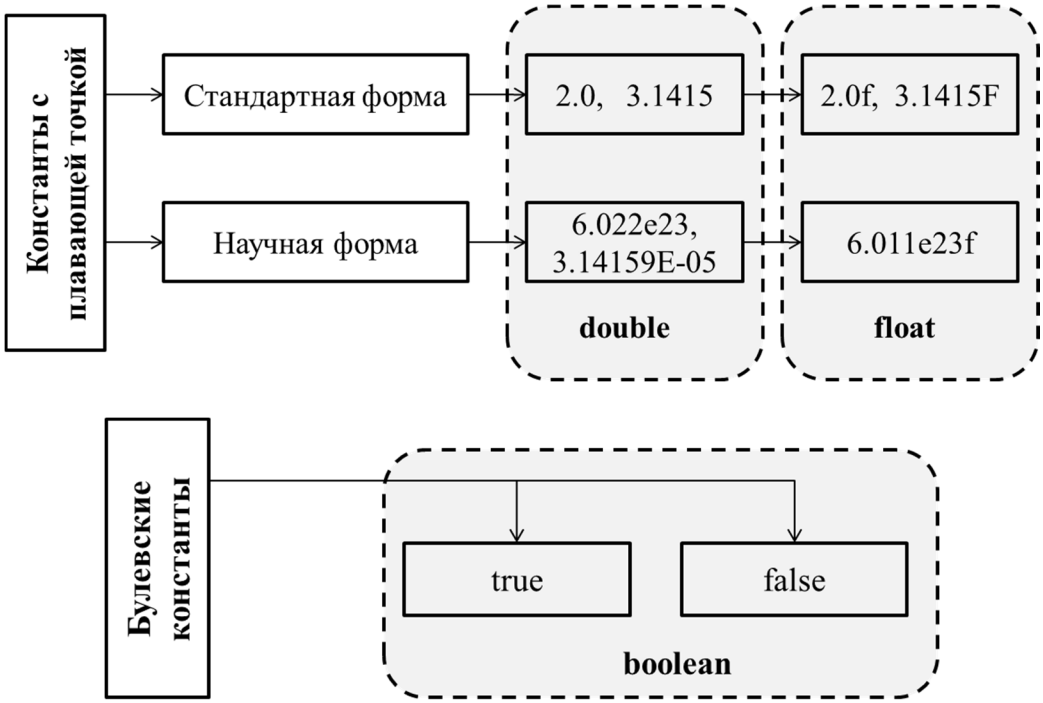


Целочисленные литералы

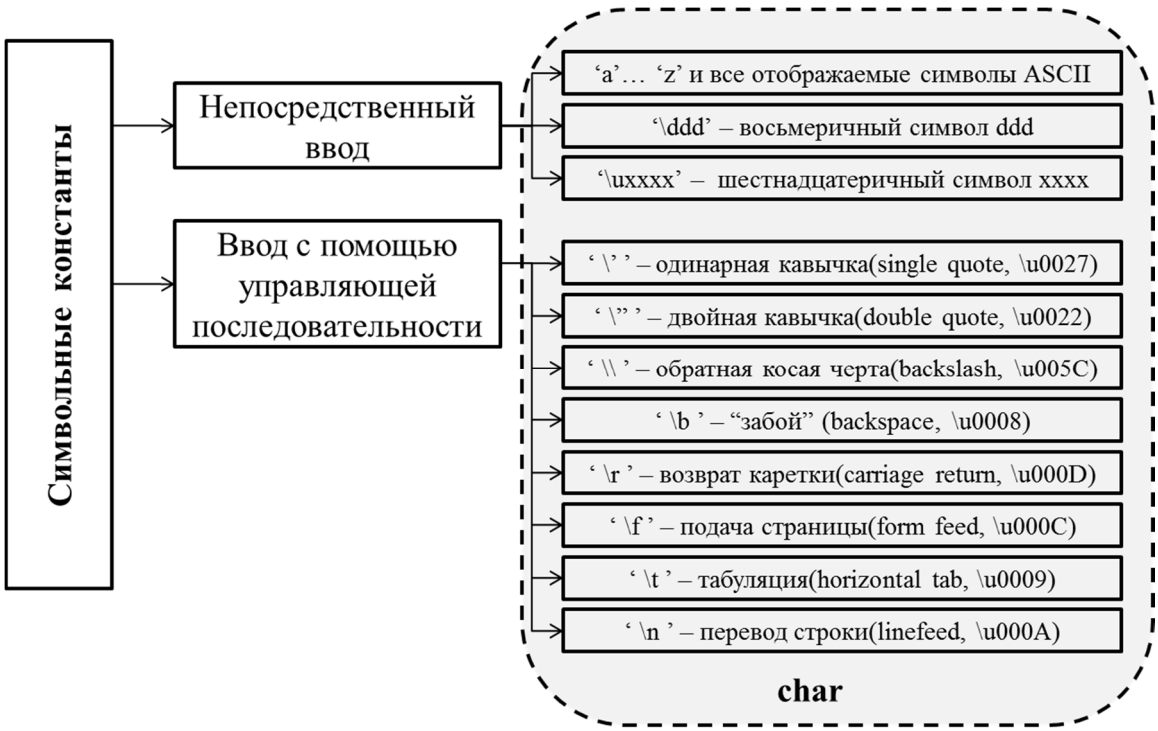


Целочисленное значение можно присваивать типу char, если оно лежит в пределах допустимого диапазона этого типа

Литералы с плавающей точкой и булевские литералы



Символьные литералы



Преобразование примитивных типов

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

Различают:

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

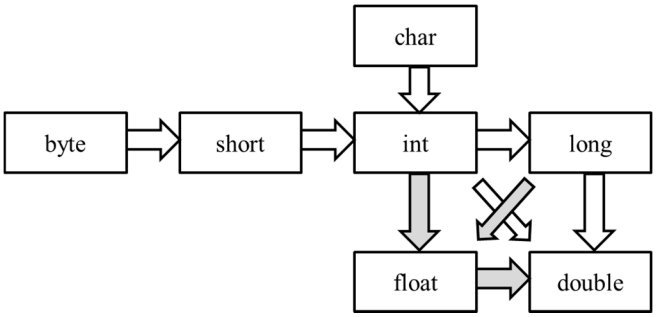
Расширяющее (повышающее) преобразование. Результирующий тип имеет больший диапазон значений, чем исходный тип.

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

Сужающее (понижающее) 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 ☺
```

Неявное (повышающее) преобразование. Повышающее преобразование осуществляется автоматически, даже в случае потери данных.



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

```
package _java._se._01._types;
public class LoseAccuracy01 {

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

```
package _java._se._01._types;
public class LoseAccuracy02 {

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

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

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

```
package _java._se._01._types;
public class LoseAccuracy03 {

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

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

Приведение типов в выражении

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

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

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

```
package _java._se._01._types;
public class LoseAccuracy04 {

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

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

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

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

Особенности приведения вещественных чисел

- Слишком *большое дробное число* при приведении *к целому* превращается в **Integer.MAX_VALUE** или **Integer.MIN_VALUE**
- Слишком *большой double* при приведении *к float* превращается в **Float.POSITIVE_INFINITY** или **Float.NEGATIVE_INFINITY**

```
package _java._se._01._types;
public class MaxDecimal {
	public static void main(String[] args) {
		double d = 1000000e100;
		int x = (int) d;
		int y = (int) (-d);
		System.out.println("x = " + x);
		System.out.println("Integer.MAX_VALUE = " + Integer.MAX_VALUE);
		System.out.println("y = " + y);
		System.out.println("Integer.MIN_VALUE = " + Integer.MIN_VALUE);
		float z = (float) d;
		float k = (float) (-d);
		System.out.println("z = " + z);
		System.out.println("k = " + k);
	}
}
```

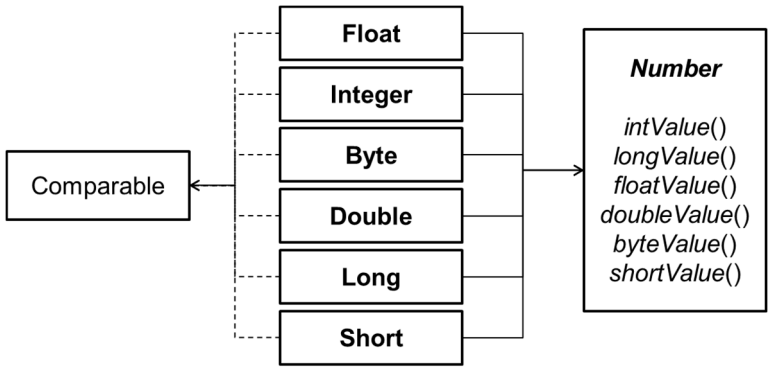
Классы-оболочки

Кроме базовых типов данных широко используются соответствующие классы (wrapper классы):

- **Boolean, Character, Integer, Byte, Short, Long, Float, Double.**
- Объекты этих классов могут хранить те же значения, что и соответствующие им базовые типы.
- Объекты этих классов представляют ссылки на участки динамической памяти, в которой хранятся их значения, и являются классами оболочками для значений базовых типов.

  Объекты этих классов являются константными.

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




```
package _java._se._01._types;
public class IntegerType {
	public static void main(String[] args) {
		Integer i = new Integer(10);
		System.out.println("i1=" + i);
		changeInteger(i);
		System.out.println("i2=" + i);
	}
4
	public static void changeInteger(Integer x) {
		System.out.println("x1=" + x);
		x = new Integer(20);
		System.out.println("x2=" + x);
	}
}
```

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**.

```
package _java._se._01._types;
public class CharacterType {

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

Big-классы

Java включает два класса для работы с высокоточной арифметикой:

BigInteger и **BigDecimal**,

которые поддерживают целые числа и числа с фиксированной точкой произвольной точности.

```
package _java._se._01._types;
import java.math.BigInteger;
public class BigNumbers {
public static void main(String[] args) {
	BigInteger numI1, numI2, bigNumI;
	numI1 = BigInteger.valueOf(1_000_000_000_000L);
	numI2 = numI1.multiply(numI1);
	System.out.println(numI2);
	numI2 = numI1.multiply(numI1).multiply(numI1);
	System.out.println(numI2);
	numI2 = numI1.multiply(numI1).multiply(numI1).multiply(numI1);
	System.out.println(numI2);
	numI2 = numI1.multiply(numI1).multiply(numI1)
					.multiply(numI1).multiply(numI1);
	System.out.println(numI2);
	numI2 = numI1.multiply(numI1).multiply(numI1).multiply(numI1)
					.multiply(numI1).multiply(numI1);
	System.out.println(numI2);
}
}
```

Автоупаковка/авторараспаковка

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

```
Integer  iob = 71;
```

Автораспаковка – процесс извлечения из объекта-оболочки значения базового типа. Вызовы таких методов, как **intValue()**, **doubleValue()** становятся излишними.

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

```
package _java._se._01._types;
public class AutoPack {
	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;
		System.out.println(k);
	}
}
```

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

```
package _java._se._01._types;
public class IntegerCache {
	public static void main(String[] args) {
		Integer i1 = 10;
		Integer i2 = 10;
		System.out.println(i1 == i2);
		i1 = 128;
		i2 = 128;
		System.out.println(i1 == i2);
	}
}
```

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

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

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

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

Операторы

Арифметические операторы

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

Операторы отношения

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

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

Битовые операторы

|	Или	>>	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		

>> - арифметической сдвиг
>>> - логический сдвиг

```
package _java._se._01. _operators;
public class URShift {
	public static void main(String[] args) {
		int i = -1; //11111111111111111111111111111111
		i >>>= 10;  //00000000001111111111111111111111
		System.out.println(i);
		long l = -1;
		l >>>= 10;    System.out.println(l);
		short s = -1;
		s >>>= 10;          System.out.println(s);
		byte b = -1;
		b >>>= 10;          System.out.println(b);
		b = -1;             System.out.println(b >>> 10);
	}
}
```

Логические операторы

||, |, |=	Или	&&, &, &=	И
!	Унарное отрицание	^, ^=	исключающее ИЛИ

&& и || - вычисление по сокращенной схеме
& и | - вычисление по полной схеме

```
package _java._se._01. _operators;
public class LogicOP {

	public static void main(String[] args){
		if(bFalse()&&bTrue()){}
		System.out.println();
		if(bFalse()&bTrue()) {}
		System.out.println();

	}
	private static boolean bTrue(){
		System.out.print("true ");
		return true;
	}

	private static boolean bFalse(){
		System.out.print("false ");
		return false;
	}
}
```

Дополнительные операторы Java

К операторам относится также оператор определения принадлежности типу **instanceof,** оператор **[ ]** и тернарный оператор **?:** (if-then-else).

Логические операции выполняются над значениями типа **boolean** (**true** или **false**).

Оператор **instanceof** возвращает значение **true**, если объект является экземпляром

данного класса.

Приоритет операций

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

```
package _java._se._01. _operators;
public class PriorityOp {
	public static void main(String[] args) {
		int i=3;
		i = -i++ + i++ + -i;
		System.out.println(i);
	}
}
```

Характеристики операторов

Операции над целыми числами: **+, –, *, %, /, ++,--** и битовые операции **&, |, ^, ~** аналогичны операциям большинства языков программирования. Деление на ноль целочисленного типа вызывает исключительную ситуацию, переполнение не контролируется (исключение не выбрасывается).

IEEE 754

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

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

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

- **Double.MAX_VALUE;**
- **Double.MIN_VALUE;**
- **Double.POSITIVE_INFINITY;**
- **Double.NEGATIVE_INFINITY;**
- **Double.NaN;**
- **Double.isNaN().**

- **Float.MAX_VALUE;**
- **Float.MIN_VALUE;**
- **Float.POSITIVE_INFINITY;**
- **Float.NEGATIVE_INFINITY;**
- **Float.NaN;**
- **Float.isNaN().**

```
package _java._se._01. _operators;
public class IEEE754 {
	public static void main(String[] args) {
		double i = 7.0;
		double k;
		System.out.println(i / 0);
		System.out.println(-i / 0);
		System.out.println(k=Math.sqrt(-i));
		System.out.println(Double.isNaN(k));
	}
}
```

strictfp

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

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

Math&StrictMath

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

- java.lang.StrictMath – класс-утилита, содержащий основные математические функции. Гарантирует точную повторяемость числовых результатов вплоть до бита на разных аппаратных платформах.
- java.lang.Math – класс-утилита, работающая быстрее чем StrictMath (на старых

версиях машины), но не гарантирующая точное воспроизводство числовых результатов.

- В версии 1.6 java.lang.Math делегирует вызовы StrictMath.

Статический импорт

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

```
package _java._se._01. _operators;
import static java.lang.Math.pow;
import static java.lang.Math.PI;
public class StaticImport {
	private int i = 20;

	public void staticImport() {
		double x;
		x = pow(i, 2)*PI;
		System.out.println("x=" + x);
	}

}
```

Блоки кода

Блоки кода обрамляются в фигурные скобки “{“ “}”
Охватывают

- определение класса;
- определения методов;
- логически связанные разделы кода.

```
public class CodeBlock {
	public Date getToday() {
		//...
	}

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

	}
}
```

Оператор if

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*/ }
```

Операторы безусловного перехода

break – применяется для выхода из цикла, оператора **switch**
continue - применяется для перехода к следующей итерации цикла

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

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

Использование циклов

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

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

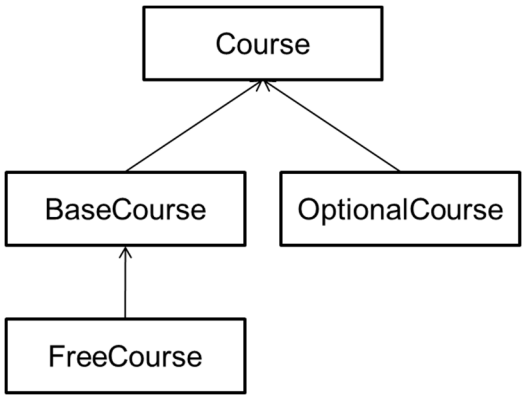
```
case exp2:/* операторы, если exp==exp2 */
	break;
default: /* операторы Java */
}
```

```
package _java._se._01. _operators;
public class SwitchWithBreak {

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

```
package _java._se._01. _operators;
public class SwitchWithoutBreak {

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

Instanceof

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




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

Объект подкласса может быть использован всюду, где используется объект суперкласса

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

```
package _java._se._01._operators;
public class InstanceofOp {
	public static void main(String[] args) {
		doLogic(new BaseCourse());
		doLogic(new OptionalCourse());
		doLogic(new FreeCourse());
	}

	public static void doLogic(Course c) {
		if (c instanceof BaseCourse) {
			System.out.println("BaseCourse");
		} else if (c instanceof OptionalCourse) {
			System.out.println("OptionalCourse");
		} else {
			System.out.println("Что-то другое.");
		}
	}
}
```

Перевод чисел в строки и обратно

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 (**“”**).

```
package _java._se._01._operators;
public class StrToNum {
	public static void main(String[] args) {
		String strInt = "123";
		String strDouble= "123.24";
```

```
                int x;
                double y;
                double z;

                x = Integer.parseInt(strInt);
                y = Double.parseDouble(strDouble);
                System.out.println(x+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);
        }
}
```

Строковое преобразование чисел

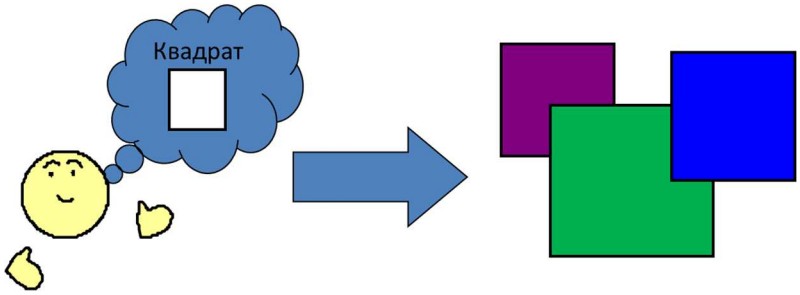
Для преобразования целого числа в десятичную, двоичную, шестнадцатеричную и восьмеричную строки используются методы **toString()**, **toBinaryString()**, **toHexString()** и **toOctalString()**.

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

Простейшие классы и объекты

Определения

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



Класс

Классом называется *описание совокупности объектов с общими атрибутами, методами, отношениями и семантикой.*

Классы **определяют структуру и поведение** некоторого набора элементов предметной области, для которой разрабатывается программная модель.

Объявление класса имеет вид:

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

Спецификаторы класса

Спецификатор класса может быть:

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

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

Свойства и методы класса

Определение класса включает:

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

Объект состоит из следующих трех частей:

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

Свойства классов:

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

Методы классов:

- метод отражает ПОВЕДЕНИЕ объектов

- выполнение методов, как правило, меняет значение свойств
- поведение объекта может меняться в зависимости от состояния

Методы

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

Объявление метода имеет вид:

[спецификаторы] [static|abstract] возвращаемый_тип имя_метода([аргументы]) {

/*тело метода*/

} | ;

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

***Спецификаторы доступа методов*:**

static	public	friendly	synchronized
final	private	native	
protected	abstract	strictfp	

Поля

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

спецификатор тип имя;

***Спецификаторы доступа полей класса*:**

static	public	final	private
protected	friendly	transient	volatile

Конструкторы

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

- 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.

Создание объекта имеет вид:

имя_класса имя_объекта= new конструктор_класса([аргументы]);

```
package _java._se._01._easyclass;
public class BookUse {
	public static void main(String[] args) {
		Book book = new Book("Java");
		System.out.println(book.getTitle());
	}
}
package _java._se._01._easyclass;
public class Book {
	private String title;
	public Book() {
		setTitle("without a title");
	}
	public Book(String title) {
		setTitle(title);
	}
	public void setTitle(String title) {
		if (null == title) {
			this.title = "no title";
		} else {
			this.title = title;
		}
	}
	public String getTitle() {
		return title;
	}
}
```

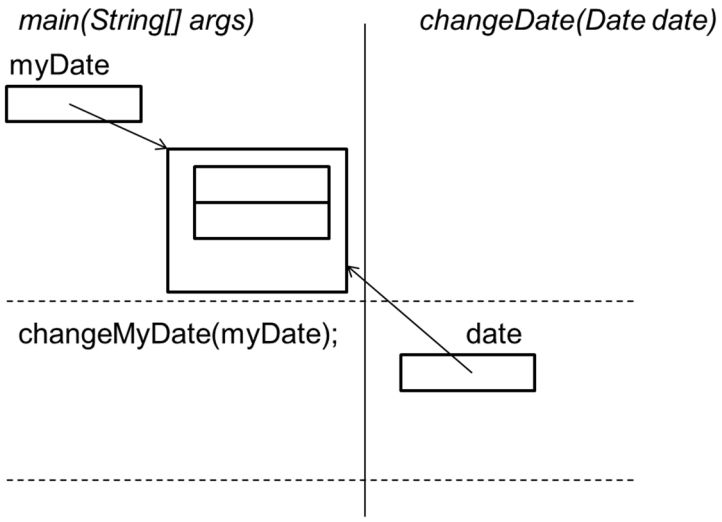
Передача параметров в методы

Методы классов, передача параметров в методы

Ссылки в методы передаются по значению.

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

```
package _java._se._01._easyclass;
import java.util.Date;
pulic class TransferParameter {
	public static void main(String[] args) {
		Date myDate = new Date();
		System.out.println("Before:" + myDate.getDate());
		changeDate(myDate);
		System.out.println("After:" + myDate.getDate());
	}
	public static void changeDate(Date date) {
		System.out.println("	- before change: " + date.getDate());
		date.setDate(12);
		System.out.println("	- after change: " + date.getDate());
	}
}
```



Передача константных объектов в методы

- При передаче в метод аргумента-ссылки можно изменить состояние объекта и оно сохранится после возвращения из метода, так как в этом случае нового объекта не создается, а создается лишь новая ссылка, указывающая на старый объект.

- Из этого правила существует одно исключение – когда передается ссылка, указывающая на константный объект.

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

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

```
package _java._se._01._easyclass;
public class StringForChange {
	public static void main(String[] args) {
		String str = "I like ";
		System.out.println("Before: " + str);
		changeString(str);
		System.out.println("After: " + str);
	}

	public static void changeString(String s) {
		System.out.println("	- before change: " + s);
		s = s + " Java.";
		System.out.println("	- after change: " + s);
	}

}
```

Основы работы со строками

Создание строкового объекта:

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

Можно использовать упрощенный синтаксис

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

Знак + применяется для объединения двух строк.

Если в строковом выражении применяется нестроковый аргумент, то он преобразуется к строке автоматически.

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

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

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

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

Пул литералов – это коллекция ссылок на строковые объекты.

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

```
package _java._se._01._easyclass;
public class ComparingStrings {
	public static void main(String[] args) {
		String s1, s2;
		s1 = "Java";
		s2 = "Java";
		System.out.println("сравнение ссылок " + (s1 == s2));
		s1 += '2';
		// s1-="a"; //ошибка, вычитать строки нельзя
		s2 = new String(s1);
		System.out.println("сравнение ссылок " + (s1 == s2));
		System.out.println("сравнение значений " + s1.equals(s2));
	}
}
```

Java beans (основы)

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.

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)*

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**.

```
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;
	}
}
```

Свойства Bean, массивы

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

```
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);
	}
}
```

Свойства Bean, boolean

Атрибуту типа **boolean** *в классе Bean* должны соответствовать 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;
	}
}
```

События

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.

Синхронизация

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

Массивы

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.

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

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.

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

Массивы объектов в действительности представляют собой массивы ссылок, проинициализированных по умолчанию значением **null**.

Все массивы хранятся в куче (**heap**), одной из подобластей памяти, выделенной системой для работы виртуальной машины.

Определить общий объем памяти и объем свободной памяти, можно с помощью методов **totalMemory()** и **freeMemory()** класса **Runtime.**

Одномерные массивы

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

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

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

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

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

Работа с массивами

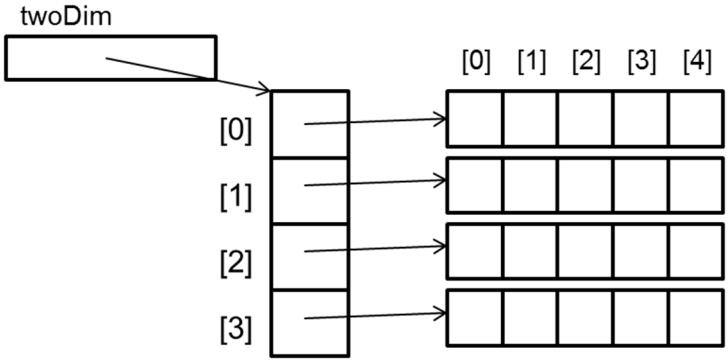
```
package _java._se._01._array;

public class FindMax {
	public static void main(String[] args) {
		int a[] = { 5, 10, 0, -5, 16, -2 };
		int max = a[0];
		for (int i = 0; i < a.length; i++){
```

```
                if (max < a[i])
                        max = a[i];
                }

                System.out.println("Max="+max);
                for(int x : a){
                        System.out.print(x+" ");
                }
                System.out.println();
        }
}
```

Массив массивов

Двумерных массивов в Java нет. Есть только массивы массивов.

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



Каждый из массивов может иметь отличную от других длину.

```
int twoDim [][] = new int[4][];
twoDim[0] = new int [10];
twoDim[1] = new int [20];
twoDim[2] = new int [30];
twoDim[3] = new int [100];
```

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

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

Работа с массивами массивов

Члены объектов-массивов:

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

```
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]);
        }
}
```

Приведение типов в массивах

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

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

Ошибки времени выполнения

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]);
        }
}
```

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);
        }
}
```

Code conventions

Code conventions for Java Programming. Содержание и причины возникновения.

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

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.

При объявлении разделяйте public и private члены класса

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

Используйте javadoc

- Javadoc – это мощный инструмент, который необходимо использовать.

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

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

Проверяйте аргументы методов

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

Дополнительные пробелы в списке аргументов

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

Применяйте Testing Framework

- Используйте testing framework чтобы убедиться, что класс выполняет контракт

Используйте массивы нулевой длины вместо null

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

Избегайте пустых блоков catch

- В этом случае когда происходит исключение, 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.

Правильно выбирайте используемые коллекции

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

Работайте с коллекциями без использование индексов

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

Структура source-файла

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

Declarations. Длина строк кода

- Не используйте строки длиной более 80 символов.

Объявление переменных

- Не присваивайте одинаковые значения нескольким 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.

Statements. Каждая строка кода должна содержать только один оператор.

- Example:

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

Имена файлов

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

Имена свойств класса

- firstName
- id

Имена методов

- getName
- isAlive

Имена констант

- SQUARE_SIZE

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

Документирование кода (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

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 ячейка 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;
	}
}
```

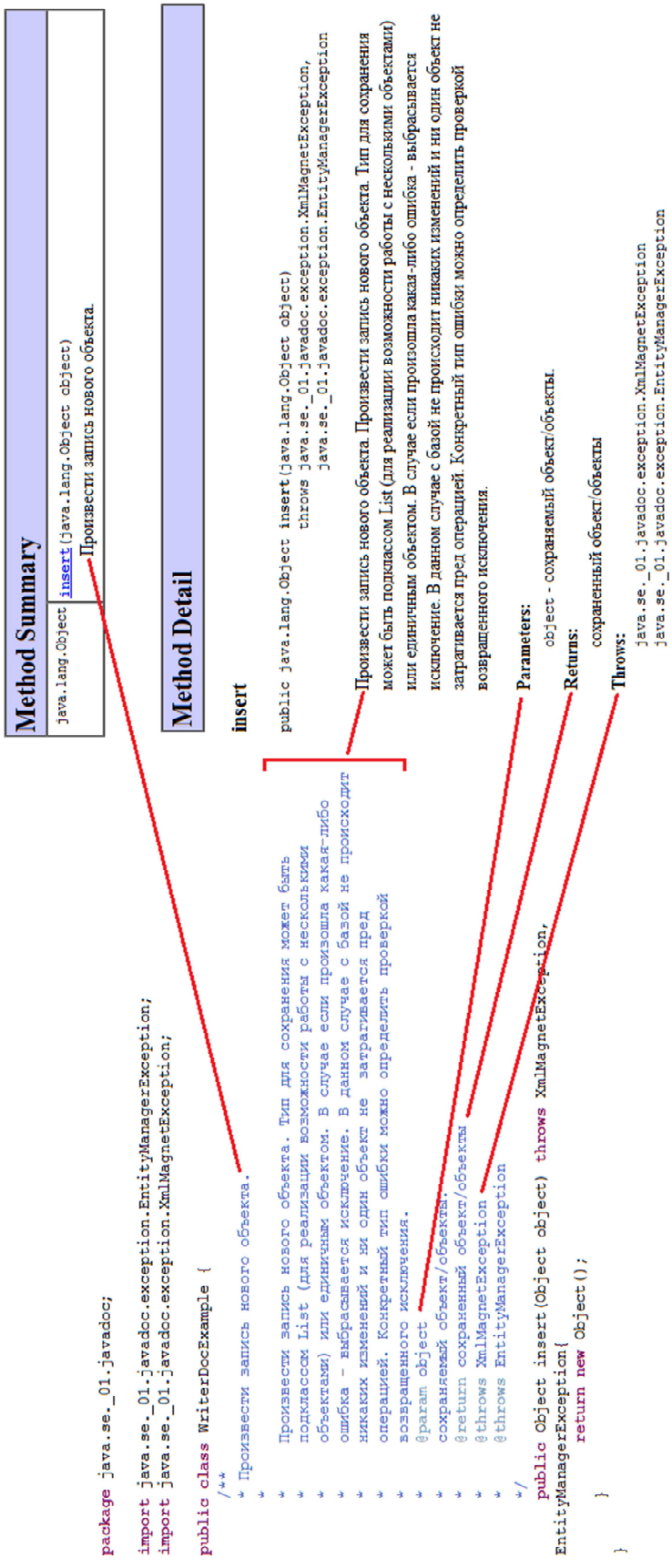
- Поддерживать документацию в актуальном состоянии

```
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-embedded.xml");
		 * String configXml = readStringFromStream(is);
		 * XmlConfigurationParserImpl parser = new
		 * XmlConfigurationParserlmpl(configXml); IEmConfiguration res =
		 * parser.parse(); assertNotNull(res);
		 * assertFalse(res.getOperationHistoryStorageConfiguration
		 * ().isEmbedded(); assertEquals("HSQLDB",
		 * res.getOperationHistoryStorageConfiguration().getStorageDBType());
		 */
	}
}
```

- 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 newlnstance(String contextName,
		IXmlObjectRelationManager objectRelationManager,
		IXmlObjectPersister xmlObjectPersister,
		OperationHistoryManager ohm,
		ISearchNotificationManager snm,
		CountDownLatch initializationLatch) {
	...
	}
}
```

-



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) {
...
```

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.

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.
```

Тег @param

Описывает параметров методов и конструкторов.
Синтаксис

```
@param <имя параметра> <описание>
```

Пример

```
@param x a value
```

Тег @return

Описывает возвращаемое значение метода.
Синтаксис:

```
@return <описание>
```
Пример:
```
@return the factorial of <code>x</code>
```

Тег @throws

Описывает исключения, генерируемые методом/конструктором.

Синтаксис:
```
@throws <класс исключения> <описание>
```
Пример:
```
@throws IllegalArgumentException if <code>x</code> is less than zero
```

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"
```

Тэг @version

Текущая версия класса/пакета.

Синтаксис:
```
@version <описание версии>
```
Пример:
```
@version 5.0
```

Тег @since

Версия в которой была добавлена описываемая сущность.

Синтаксис:
```
@since <описание версии>
```
Пример:
```
@since 5.0
```

Тэг @deprecated

Помечает возможности, которые не следует использовать.

Синтаксис:
```
@deprecated <комментарий>
```
Пример:
```
@deprecated replaced by {@link #setVisible}
```

Тэг @author

Описывает автора класса/пакета.

Синтаксис:
```
@author <имя автора>
```
Пример:
```
@author  Josh Bloch
@author  Neal Gafter
```

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}
```

Тэг {@docRoot}

Заменяется на ссылку на корень документации.

Синтаксис:

```
{@docRoot}
```

Пример:

```
<a href="{@docRoot}/copyright.html">Copyright</a>
```

Тэг {@value}

Заменяется на значение поля.

Синтаксис:

```
{@value <имя класса>#<имя поля>}
```

Пример:

```
Default value is {@value #DEFAULT_TIME}
```

Тэг {@code}

Предназначен для вставки фрагментов кода.
Внутри тэга HTML не распознается.

Синтаксис:

```
{@code <код>}
```

Пример:

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

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.

Наследование Javadoc

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

Копируемая информация:

- описание
- @param
- @return
- @throws

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>

Наследование Javadoc

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

Копируемая информация:

- описание
- @param
- @return
- @throws

Компиляция Javadoc

- Инструмент

javadoc

- Применение

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

- Пример

javadoc JavadocExample1.java

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

Пример

```
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 Веап'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 {
        /**
        * 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(); }
        …
}
```

All Classes

EntityManagerInvoker

Package Class **Use** **Tree** **Deprecated** **Index** **Help**

PREV PACKAGE NEXT PACKAGE FRAMES NO FRAMES

Package java.se._01.javadoc

Class Summary	
EntityManagerInvoker	Представитель модуля EntityManger на клиентской стороне.

Package Class **Use** **Tree** **Deprecated** **Index** **Help**

PREV PACKAGE NEXT PACKAGE FRAMES NO FRAMES

Author:
Ivanov

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