Модуль #5
Примитивные типы и операторы

Модуль 5

- **Примитивные типы**
- Декларация переменных и их инициализация
- Приведение примитивных типов
- Массивы
- Garbage collector
- Параметры методов
- Операторы в Java
- Контроль потока выполнения

Типы данных

Условно, все типы данных можно разделить на 2 типа:

- Примитивные (базовые)
- Объектные (ссылочные)

Внимание! Массивы (в т.ч. массивы примитивов), перечисления и строки являются объектами

Примитивные типы данных

Примитивный тип данных – тип данных, предоставляемый языком программирования как базовая, встроенная единица языка.

- `boolean` (логический тип)
- `char` (символьный тип)
- `byte, short, int, long` (целый тип)
- `float, double` (вещественный тип)

Примитивные типы данных

- Особенности примитивных типов:
  - Простые типы
  - Зарезервированные ключевые слова языка
  - Значения типов передаются “по значению”

```
int i = 5;
int j = i;
j++;
// j => 6, i => 5
```

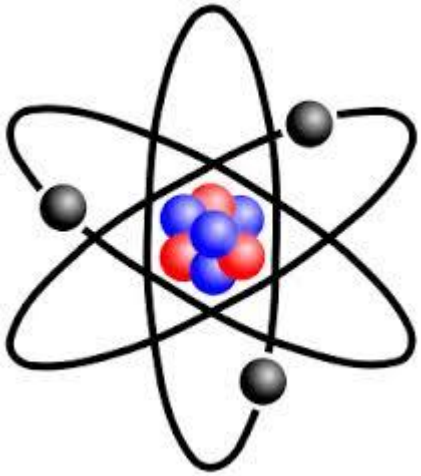
Примитивные типы данных

Разные примитивы имеют разное битовое представление, их размер строго определен спецификацией Java.

`byte`	8	bits
`short`	16	bits
`int`	32	bits
`long`	64	bits
`float`	32	bits
`double`	64	bits
`char`	16	bits

Внимание! В отличии от примитивных типов других языков, примитивные типы Java имеют фиксированный размер, что, отчасти, вызвано принципом **«Write once, run anywhere»**

Модель памяти



Атомарность типа говорит о том, что виртуальная машина выполняет операции над ним «в один заход».

Например:

Типы с плавающей запятой не атомарны, так как операции над мантиссой и экспонентой выполняются последовательно.

Внимание! Для обеспечения атомарности можно использовать классы из пакета `java.util.atomic.*`.

Логический примитивный тип

- Тип `boolean` может принимать только 2 значения: `true` или `false`
- Фактический размер может варьироваться в различных JVM.

```
boolean b1 = true;
boolean b2 = false;
```

Внимание! boolean не может быть приведен ни к какому типу. Обратное тоже верно.

Целые примитивные типы

Типы данных **byte**, **short**, **int**, **long** являются знаковыми – значения этих типов могут быть положительными, отрицательными и 0.

Тип	Размер	Минимум	Максимум
byte	**8 bits**		
short	**16 bits**		
int	**32 bits**		
long	**64 bits**		

?

Целые примитивные типы

Типы данных **byte**, **short**, **int**, **long** являются знаковыми – значения этих типов могут быть положительными, отрицательными и 0.

Тип	Размер	Минимум	Максимум
byte	8 bits	-2^7	2^7 - 1
short	16 bits	-2^15	2^15 - 1
int	32 bits	-2^31	2^31 - 1
long	64 bits	-2^63	2^63 - 1

Вещественный примитивный тип

Тип	Размер	Минимум
`float`	32 bits	+/-1.40239846^-45
`double`	64 bits	+/-4.94065645821^-324

	Максимум
`float`	+/-3.40282347^+38
`double`	+/-1.797693134862 3^+308

Вещественный примитивный тип

Тип	Знак	Мантиса	Экспонента
`float`	`1 bit`	`23 bits`	`8 bits`
`double`	`1 bit`	`52 bits`	`11 bits`

`+/-m * 2^e`, где m – мантисса, e - экспонента

- Стандарт IEEE-754

Символьный тип

Unicode — это объединение десятков кодировок символов, он включает в себя латинский, греческий, арабский алфавиты, кириллицу и многие другие наборы символов.

- Тип **`char`** представляет символом **`unicode`**, занимает 16 бит и является беззнаковым (unsigned)

Тип	Размер	Минимум	Максимум
`char`	`16 bits`	`0`	`2^16 - 1`

Символьный тип

- Код – 4 цифры в 16-ичном формате

- Так же можно указывать специальный “escape character”:

`\n`	новая строка
`\r`	возврат
`\t`	табуляция
`\b`	бэкспэйс
`\'`	одинарная кавычка
`\”`	двойная кавычка
`\\`	обратный слэш

Примитивные типы данных

Литерал – запись в исходном коде компьютерной программы, представляющая собой фиксированное значение.

- Литералы не могут быть изменены в тексте программы.
- Литералы представляют собой константы, непосредственно включаемые в текст программы.

Типы литералов:

- строковые
- логические
- числовые
- `null`-литерал

Примитивные типы данных

- Единственно допустимые литералы для типа `boolean` – значения `true` или `false`.
- Символьный литерал задается в одинарных кавычках.

```
char c = 'w'
```

- Строковый литерал задается в двойных кавычках

```
String s = “www”
```

- Символ Unicode можно указывать в формате

```
char c1 = '\u4567'
```

Использование примитивных типов

Использование любого типа данных включает в себя 2 этапа:

- Декларация переменной (объявление).
- Инициализация переменной некоторым значением.

```
int t; // Декларация
t = 7; // Инициализация
int var = 5; // Декларация и инициализация
```

```
int foo, bar, baz;

foo = bar = baz = 10;
```

Использование целых типов

- Целочисленным типам могут быть присвоены значения целочисленных литералов.

- Значение может быть задано в 10-ичном (по умолчанию), в 16-ричном (префикс 0x или 0X) и 8-ричном (префикс 0), в 2-ичном (префикс 0b).

```
28
034
0x1c
0X1C
0X1c
0b1101
```

Использование целых типов

- Начиниая с Java 7 можно использовать подчеркивание:

```
1_000_000_000_000

123_456_789

123_232_323_232
```

Примитивные типы данных

- По умолчанию целочисленный литерал типа **`int`**.

- Для того, что указать значение типа **`long`**, необходимо указать суффикс **`l`** или **`L`**.

```
long val = 10L;
```

Примитивные типы данных

Инициализация вещественной переменной:

- Десятичная точка `1.414`
- Буквы `e` и `E` `4.23E+21`
- Суффикс `f` и `F` `1.828f`
- Двоичная форма `0b111010`
- Суффикс `d` и `D` `1234d`

Строковый тип данных

- Строка – последовательность символов unicode, строка в литеральном представлении заключается в "".

- Класс – `java.lang.String`.

- Не то же, что массив символов.

- Никаких нулевых символов в конце, длина хранится отдельно.

Строковый тип данных

- Строковые литералы:
  - `String zeros = "\u0000\u0000";`
  - `String hello = "Hello";`
  - `String specialChars = "\r\n\t\\\"";`
  - `String unicodeEscape = "\u0101u\2134"`
- Создание из массива символов:

```
char[] chars = {'a', 'b', 'c'};
String str = new String(chars);
```

Доступ к содержимому строки

- `int length()`
- `char charAt(int index)`
- `char[] toCharArray()`
- `String subString(int beginIdx, int endIdx)`

Сравнение строк

- Оператор == сравнивает ссылки, а не содержимое строк.

```
boolean equals(Object obj)
boolean equalsIgnoreCase(Object obj)

int compareTo(String str)
int compareToIgnoreCase(String str)
```

Операции

- Строки неизменяемы.

```
boolean startsWith(String prefix)
boolean endsWith(String suffix)

int indexOf(String str)
int lastIndexOf(String str)
```

Операции

- `String trim()`

  `String replace(char oldChar, char newChar)`

  `String toLowerCase()`

  `String toUpperCase()`

  `String[] split(String regexp)`

Конкатенация строк

- `String concat(String str)`
- Оператор `+`

```
String str1 = “Hello” + “, World!”

String str2 = 3 + “, World!” + 10;
```

Внимание! Для конкатенации строк обычно используется StringBuilder.

Модуль 5

- Примитивные типы
- **Декларация переменных и их инициализация**
- Приведение примитивных типов
- Массивы
- Garbage collector
- Параметры методов
- Операторы в Java
- Контроль потока выполнения

Типы инициализации полей объектов и классов

Название	Применимость	Описание
Инициализация в месте объявления поля	Поля класса, Поля объекта	Применяется, если инициализация может быть произведена коротким выражением и доступен контекст
Инициализационный блок	Поля класса, Поля объекта	Применяется, код неудобно записывать одним выражением.
Конструктор класса	Поля объекта	Применяется, если для инициализации нужны параметры конструктора

- Значение по умолчанию, а так же явное значение свойств объекта присваивается перед выполнением конструктора.
- Значение по умолчанию, а так же явное значение свойств класса (`static` свойств), присваивается при загрузке класса.

Инициализация по умолчанию

- Если при декларации переменной класса или объекта ей не было присвоено значение, JVM присваивает значение по умолчанию.
- Это справедливо и для примитивов и для объектов.

Тип	Начальное значение	Тип	Начальное значение
`byte`	`0`	`short`	`0`
`int`	`0`	`long`	`0L`
`float`	`0.0f`	`double`	`0.0d`
`char`	`'\u0000'`	`boolean`	`false`

Инициализация по умолчанию

```
public class Person {
    // JVM присвоит значение по умолчанию (null)
    String name;

    // Присвоили значение явно
    int age = 5;

    // Значение по умолчанию (0)
    int gender;

    public static void main(String[] args) {
        Person personInstance = new Person();
    }
}
```

Инициализация переменных

- JVM не присваивает значение по умолчанию для локальных переменных.

- **«Variable XXX may not have been initialized»**

```
public double fourthRoot(double d) {
    double result;
    if (d >= 0) {
        result = Math.sqrt(Math.sqrt(d));
    }
    return result; // Error: The local variable result
may not have been initialized
}
```

Внимание! Однако, если компилятор может определить, что блок выполняется всегда, он допускает подобную инициализацию

Инициализация в статическом блоке

```
static int a = 10;

static List<Character> alphabet;

static
{
    alphabet = new ArrayList<Character>();
    for (char c = 'a'; c <= 'z'; c++)
    {
        alphabet.add(c);
    }
}
```

- Инициализация статических полей и статические блоки выполняются в порядке их объявления.

Модуль 5

- Примитивные типы
- Декларация переменных и их инициализация
- **Приведение примитивных типов**
- Массивы
- Garbage collector
- Параметры методов
- Операторы в Java
- Контроль потока выполнения

Типы данных

Есть два типа преобразований данных:

- Конвертация типов
- Приведение типов

```
Type1 v1; // Type 1 - примитив
Type2 v2; // Type 2 - примитив
v1 = v2;  //???
```

Типы данных

- Типы `Type1` и `Type2` известны при компиляции.
- Все конвертации примитивов происходят во время компиляции программы.

Типы данных

Конвертация примитивных типов может возникать во время:

- Присваивания
- Вызова метода
- Арифметических переводов

Типы данных

Конвертация при присваивании
(**i** конвертируется в **double** и принимает)
значение **10.00000000**

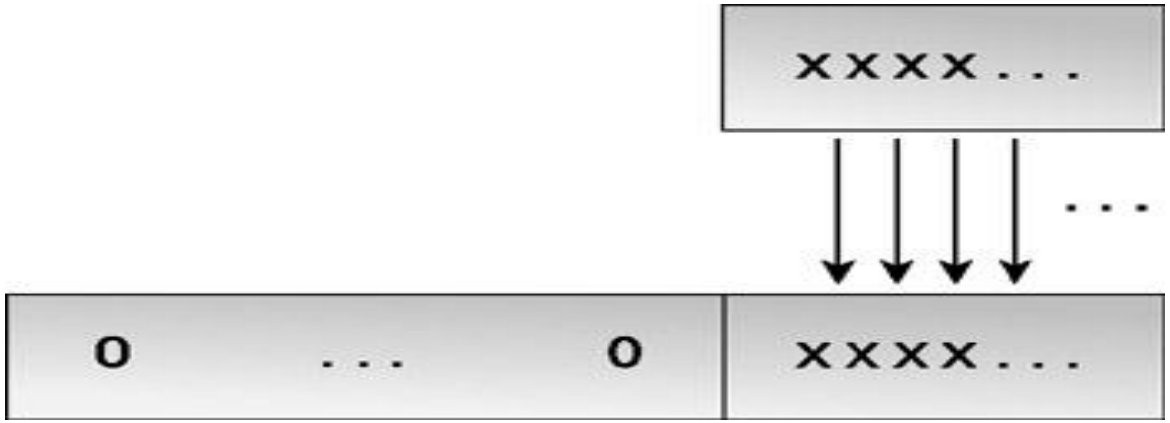
```
int i;
double d;
i = 10;
d = i; // Assign an int value to a double
variable
```

Возможна ситуация, когда значение данного типа не помещается в присваиваемом типе(Происходит ошибка компиляции):

```
double d;
short s;
d = 1.2345;
s = d; // Assign a double value to a short
```

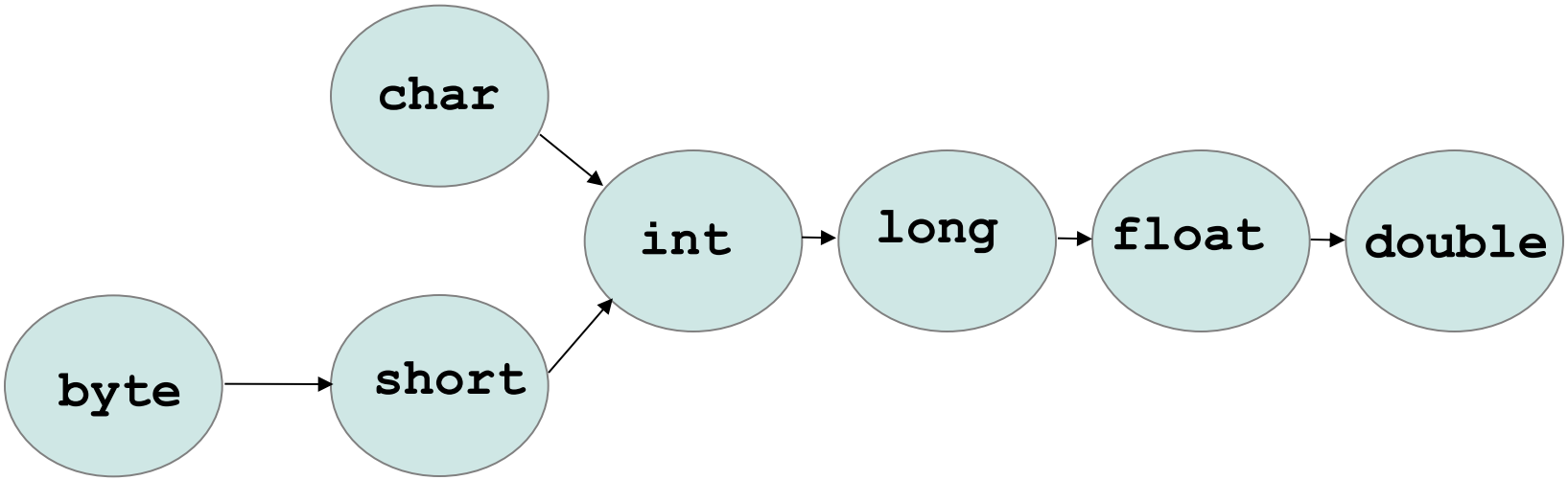
Типы данных

- **boolean** НЕ может быть конвертирован к другому типу данных.
- Не-**boolean** может быть конвертирован к другому не-**boolean**, если конвертация расширяющая.
- Не-**boolean** НЕ может быть конвертирован к другому не-**boolean** типу, если конвертация сужающая.



Примитивные типы данных

- Схема расширяющего преобразования:



- Все прочие преобразования являются сужающими.

Примитивные типы данных

- По умолчанию числовой литерал — это значение типа `int` или `double`.

```
float a = 1.234; // Ошибка
```

- Следующие возможно:

```
byte  b = 1;
short s = 2;
char  c = 3;
```

Внимание! Компилятор выполняет проверку литерала и, если он попадает в диапазон допустимых значений типа, разрешает присваивание

Примитивные типы данных

- Конвертация примитивов также может происходить при вызове метода:

```
public void cos(double d) {…}


{
    float a = 2.34f;
    Math.cos(a);
}
```

Внимание! Правила конвертации при вызове метода те же, что и при присваивании

Примитивные типы данных

- Арифметический перевод возникает при вычислении выражения, в котором участвуют значения различных типов данных:

```
short s = 9;
int i = 10;
float f = 11.1f;
double d = 12.2;
if (-s * i >= f / d)
   System.out.println(">=");
else
   System.out.println("<");
```

Примитивные типы данных

Для унарных операторов:

- Если операнд типа **byte**, **short** или **char**, он конвертируется к **int** перед выполнением операции (если операция не **++** или **--**, иначе конвертация не происходит)

```
byte b = 5;
byte b1 = -b; // Type mismatch: cannot convert
from int to byte
```

Внимание! Во всех остальных случаях конвертация не происходит

Примитивные типы данных

Для бинарных операторов:

- Если один операнд – **`double`**, второй конвертируется в **`double`**
- Если один операнд – **`float`**, второй конвертируется в **`float`**
- Если один операнд – **`long`**, второй конвертируется в **`long`**
- Иначе — оба конвертируются в **`int`**

Примитивные типы данных

- Преобразование (cast) – явное указание компилятору производить преобразование типов.
- Тип, к которому производится преобразование заключается в ():

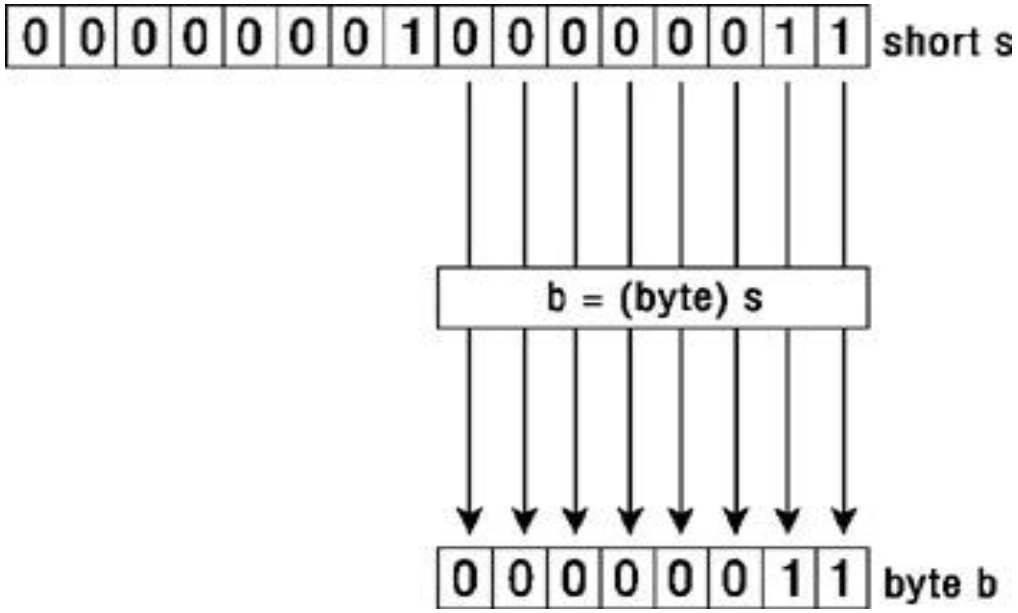
```
double d = 5;
int i = (int) d;
```

  - Cast необходим, когда выполняются сужающие преобразования.
  - Cast – указание компилятору «Я инженер, я знаю, что я делаю»

Примитивные типы данных

- Преобразование (cast) примитивов.

```
short s = 259;
byte b = (byte) s; // Explicit cast
System.out.println("b = " + b);
```



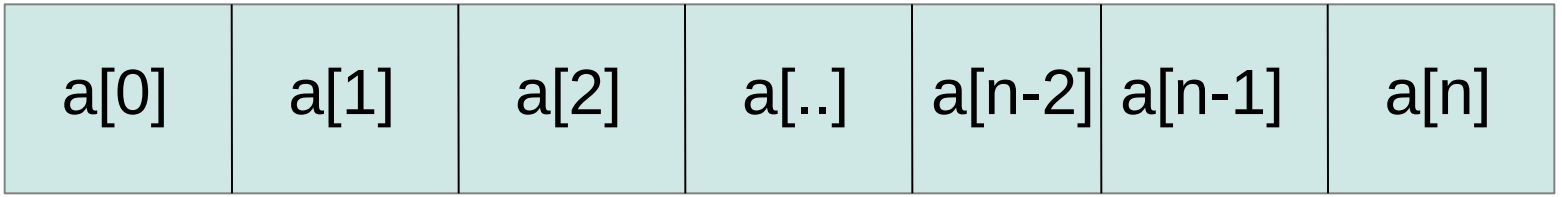
Внимание! Нельзя преобразовывать `boolean` к любому другому типу

Модуль 5

- Примитивные типы
- Декларация переменных и их инициализация
- Приведение примитивных типов
- **Массивы**
- Garbage collector
- Параметры методов
- Операторы в Java
- Контроль потока выполнения

Массивы

- Массив – упорядоченная коллекция примитивов, объектных ссылок или других массивов
- Java массивы гомогенны – массивы должны содержать элементы одинакового типа



Массивы

Для того, чтобы создать и использовать массив необходимо выполнить 3 стадии:

- Декларация
- Конструирование
- Инициализация

```
int[] ints;
Dimension dims[];
float[][] twoDee;
```

Массивы

- Размер массива указывается в runtime

```
int[] ints;                 // declaration
ints = new int[25];         // runtime construction
```

- Размер можно специфицировать переменной

```
int size = 1152 * 900;
int[] raster;
raster = new int[size];
```

- Декларация и конструирование могут быть объеденены

```
int[] raster = new int[size];
```

Внимание! После того как массив сконструирован, его элементам автоматически присваивается значение по умолчанию

Массивы

- Массив можно инициализировать литералами

```
float[] diameters = {1.1f, 2.2f, 3.3f};
```

- Массив можно инициализировать явно

```
long[] diameters = new long[6000];
long[10] = 1;
long[30] = 42;
```

- Массив java — объект!

Массивы

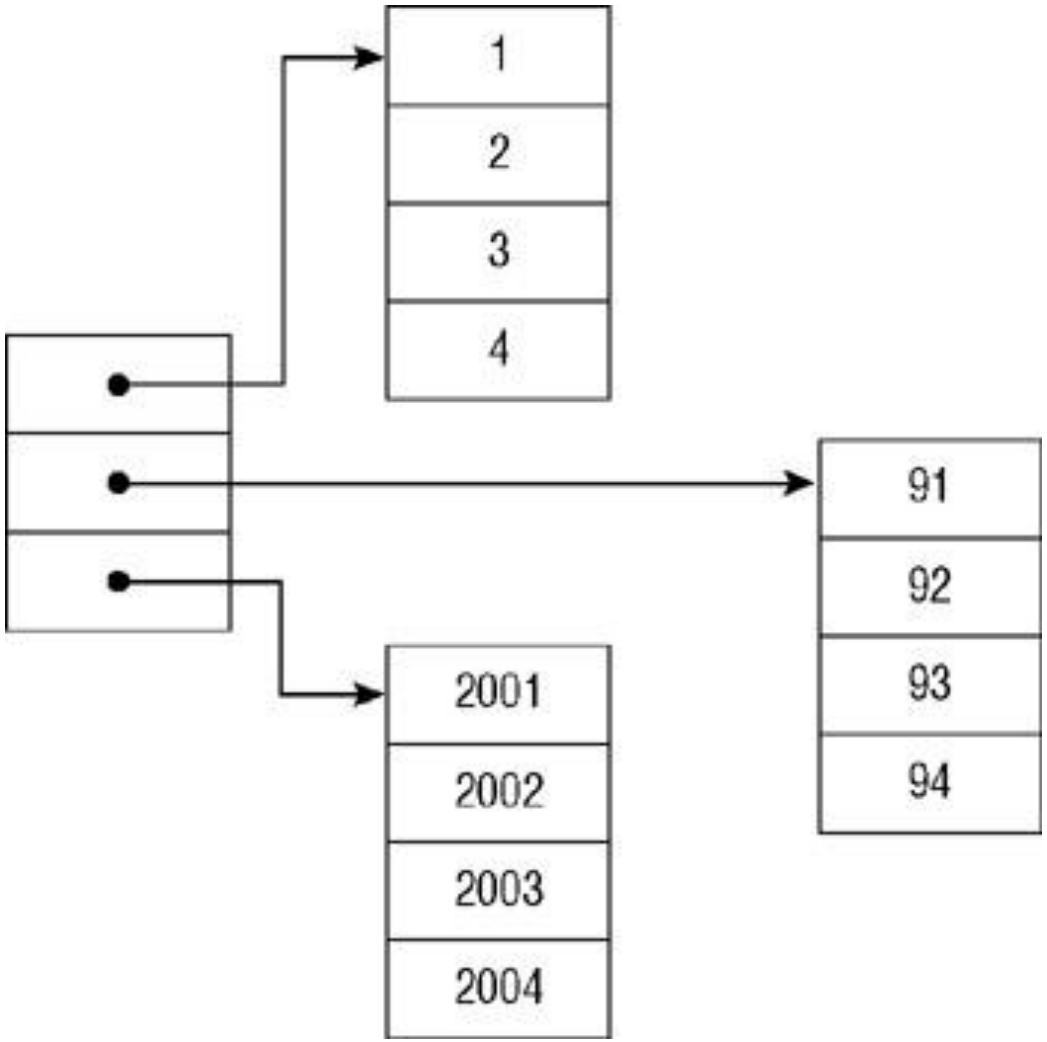
- Размер массива можно получить с помощью свойства `length`.

```
long[] squares = new long[6000];
for (int i = 0; i < squares.length; i++)
{
    squares[i] = i * i;
}
```

- Массивы могут быть многомерные.

```
int[][] myInts = new int[3][4];
```

Размерность массива



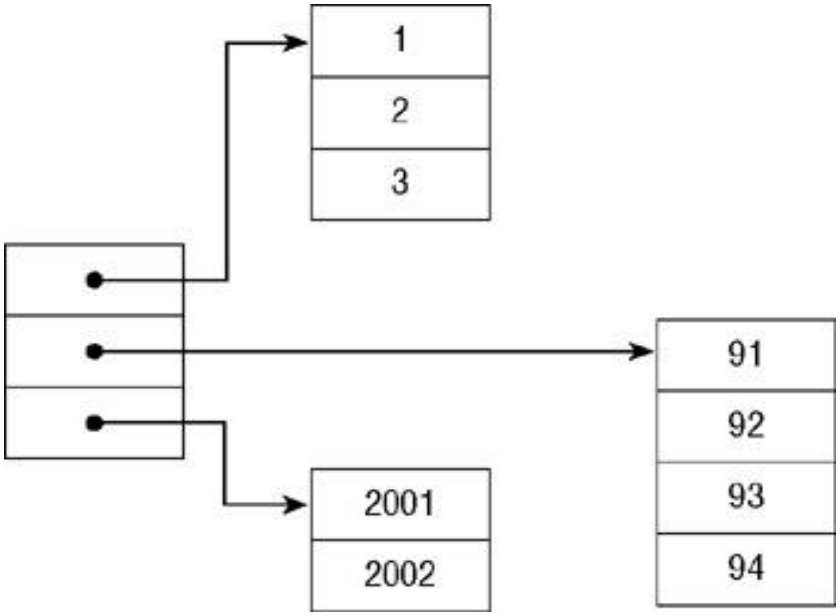
Внимание! Подчиненные массивы необязательно должны иметь одинаковую длину

Размерность массива

```
int[][] myInts = {
   {1,2,3}, {91,92,93,94}, {2001, 2002}}

int[] replacement =
   {1,2,3,4,5,6,7,8,9,10,11,12};

myInts[1] = replacement;
```



Упражнение 7

Использование базовых типов и их приведение

Модуль 5

- Примитивные типы
- Декларация переменных и их инициализация
- Приведение примитивных типов
- Массивы
- **Garbage collector**
- Параметры методов
- Операторы в Java
- Контроль потока выполнения

Garbage collection

- Объекты создаются в куче с помощью оператора `new` и могут занимать значительный объем памяти

```
MyClass mc = new MyClass();
```

- Во многих языках программирования память под выделенный объект освобождается явно:
  - Освободив память слишком рано, можно привести к порче данных
  - Можно забыть освободить память
  - Можно дважды очистить память

Внимание! В Java память никогда не освобождается явно, этим процессом занимается garbage collector

Garbage collection

- Сборка мусора – это освобождение памяти, путем удаления объектов, которые уже не будут востребованы приложением.
- Сборщик мусора – процесс-демон.

Внимание! В Java память никогда не освобождается явно, этим процессом занимается garbage collector

Алгоритмы сборки мусора

- Подсчет ссылок
- Трассирующие сборщики мусора
- Маркирующе-зачищающие сборщики
- Копирующие сборщики мусора
- Маркирующе-сжимающие сборщики мусора
- Сборка на основе поколений

Внимание! В Java одновременно работает 5 сборщиков мусора

Garbage collection

- Метод **`finalize()`** класса **`Object`** вызывается у объекта перед его освобождением.
- Вызов этого метода не гарантируется.
- Обезателен вызов **`finalize()`** базового класса при переопределении.

```
@Override
public void finalize()
{
    super.finalize();
    ...
}
```

Garbage collection

- Спецификация Java не отвечает на вопрос «когда вызывается Garbage Collector».

- Программный вызов:

```
System.gc(); Runtime.getRuntime().gc();
```

Внимание! Нет гарантии что вызов этих методов приведет к сборке мусора, это лишь подсказка для JVM.

Garbage collection

```
public class Person {
    byte[] array = new byte[1024];

    protected void finalize() throws Throwable {
        System.out.println("GC was called");
        super.finalize();
    }

    public static void main(String[] args) {
        if (true) {
            Person p = new Person();
        }
        // в данной строке память, объект Person созданный
в блоке является gc eligible, т.к. управление вышло за
пределы блока. Вызовется ли метод finalize()?
    }
}
```

Типы ссылок

- Java поддерживает следующие виды ссылок:
  - **Жесткие ссылки** (Strong Reference) – стандартные, известные нам ссылки. Если на объект есть хотябы одна жесткая ссылка, то данный объект не будет утилизирован при сборке мусора.
  - **Мягкие ссылки** (Soft Reference) – если на объект есть только мягкая ссылка, то будет выполнена попытка утилизации данного объекта при сборке мусора, если приложению не хватает памяти.
  - **Слабые ссылки** (Weak Reference) – если на объект есть только слабая ссылка, то будет выполнена попытка утилизации этого объекта при сборке мусора.
  - **Фантомные ссылки** (Phantom Reference) – если на объект есть только фантомная ссылка, то будет выполнена попытка утилизации этого объекта при сборке мусора. Сам объект не будет удален из памяти, до тех пор пока на него существует фантомная ссылка или ссылке не отчищена вызовом `clear()`.

Модуль 5

- Примитивные типы
- Декларация переменных и их инициализация
- Приведение примитивных типов
- Массивы
- Garbage collector
- **Параметры методов**
- Операторы в Java
- Контроль потока выполнения

Передача параметров

- Существует два способа передачи параметров в метод:
  - Передача параметра по значению
  - Передача параметра по ссылке
- При передаче аргументов в Java создается копия аргумента (верно и для примитивов и для объектов).

```
public void bumper(int bumpMe) {
        bumpMe += 15;
}
    ….
int var = 5;
obj.bumper(var);
System.out.print(var);
```

Передача объектной ссылки

- Работа с объектом в Java всегда ведется через ссылку

```
Button b = new Button("Ok");
```

- Передача `b` создает копию ссылки.
- Возможно изменить состояние объекта `b` через ссылку.
- При передаче в метотод изменить значение оригинальной ссылки невозможно.

Передача объектной ссылки

```
public static void main(String[] args) {
    Button btn;
    btn = new Button("Ok");
    System.out.println(btn.getLabel()); // Ok
    changeLabel(btn);
    System.out.println(btn.getLabel()); // ?
    changeRef(btn);
    System.out.println(btn.getLabel()); // ?
}
private static void changeLabel(Button btn) {
    btn.setLabel("Cancel");
}
private static void changeRef(Button btn) {
    btn = new Button("Apply");
}
```

Внимание! Скопировать и передать ссылку – незатратная операция.

Передача объектной ссылки

- Необходимо помнить это при передаче массива, т.к. массив – это объект!

```
public static void main(String[] args) {
    int m [] = {1};
    changeArray(m);
    System.out.println(m[0]);
}
private static void changeArray(int[] array) {
    array[0] = 2;
}
```

Параметер final

- Модификатор **`final`** указывает, что значение переменной не может быть модифицировано после присваивания.
- Разумно всегда декларировать аргумент метода как **`final`**.

```
public static void main(String[] args) {
    int m[] = {1};
    changeArray(m);
    System.out.println(m[0]);
}
// Note that parameter array is declared final
private static void changeArray(final int[] array)
    array[0] = 2;
    array = new int[1]; // Compiler error
}
```

Модуль 5

- Примитивные типы
- Декларация переменных и их инициализация
- Приведение примитивных типов
- Массивы
- Garbage collector
- Параметры методов
- **Операторы в Java**
- Контроль потока выполнения

Операторы в Java

Категория	Операторы	Ассоциативность
Unary	++ -- + - ! ~ *(type)*	R to L
Arithmetic	* / % plus; -	L to R
Shift	<< >> >>>	L to R
Comparison	< <= > >= instanceof = = !=	L to R
Bitwise	& ^ |	L to R
Short-circuit	&& ||	L to R
Conditional	?:	R to L
Assignment	= *op*=	R to L

Внимание! Перед выполнением операндов, JVM должна вычислить значения операндов. Вычисление происходит слева направо.

Порядок вычисления операндов

```
int[] a = { 4, 4 };
int b = 1;
a[b] = b = 0;
```

- Вначале вычисляется **a[b].**
- Затем выполняется присваивание.

Порядок вычисления операндов

- Операторы Java могут иметь один, два и три операнда, соответственно выделяют:
  - Унарные операторы
  - Бинарные операторы
  - Тернарные операторы

Унарные операторы

- Унарный оператор выполняет работу только над одним операндом
  - Инкремент и декремент (++ и --)
  - Унарный плюс и минус (+ и -)
  - Побитовая инверсия (~)
  - Логическое отрицание (!)
  - Приведение типа ('()')

Initial Value of *x*	Expression	Final Value of *y*	Final Value of *x*
5	`y = x++`	5	6
5	`y = ++x`	6	6
5	`y = x--`	5	4
5	`y = --x`	4	4

Унарные + и –

```
x = -3;
y = +3;
z = -(y + 6);
```

- Необходимо помнить, что при выполнении операции производится arithmetic promotion.

```
byte b = 5;
byte b1 = +b; // Ошибка: Type mismatch: cannot
convert from int to byte
```

Оператор + со строками

- Оператор кроме числовых типов применим к типу `String`
- Если операнд – объект, у него вызывается метод `toString()`, который возвращает строковое представление объекта.
- Если значение операнда – `null`, то строка `"null"`

Оператор + со строками

```
int i = 5;
int j = 5;

System.out.println(i + j); // 10

System.out.println("7" + j);// 75

System.out.println(new Person() + " 7"); // Person@ad3ba4

Person p = null;

System.out.println(p + "7"); // null7


System.out.println(new Person() + 7); // Compilation erro
```

Битовые инверсии

- Унарный оператор ~ производит инвертирование битового представления операнда.

```
~00001111 = 11110000
```

  - Применяется для целочисленных типов.
  - Используется совместно с операторами сдвига.

Логическая инверсия

- Унарный оператор `!` Инвертирует значение `boolean` выражения.

```
!true = false
```

```
!false = true
```

- Количество инверсий неограничено.

```
!!!!!!!!!!false = false
```

Оператор cast

- Оператор cast используется для явного преобразования типов.
- Существуют ограничения на преобразуемые типы.
- Нельзя приобразовать ссылку в примитив.
- Также не всегда можно преобразовать один объектный тип в другой.

```
int c = (int)(Math.PI * diameter)
```

Оператор умножения в java

- Операторы `*` и `/` производят умножение и деление.
- Целочисленное деление на 0 возбуждает **`ArithmethicException`**.
- Ситуация, когда результат `*` превышает максимальную размерность результирующего типа называется overflow.
- Ситуация, когда целочисленное деление приводит к потере дробной части результата называется underflow.

Оператор в java

```
int a = 12345, b = 234567, c, d;
long e, f;

c = a * b / b; // this should equal a, that is,
12345
d = a / b * b; // this should also equal a
System.out.println("a is " + a + "\nb is " + b +
"\nc is " + c + "\nd is " + d);

e = (long) a * b / b;
f = (long) a / b * b;
System.out.println("\ne is " + e + "\nf is " + f);
```

Оператор % в java

- Оператор `%` возвращает остаток от деления первого операнда на второй.
- Оператор `%` может быть применен к целочисленным и и вещественным типам.
- Знак результата совпадает со знаком левого операнда.
- Оператор возбуждает **`ArithmethicException`** при делении на 0.

Оператор % в java

17 % 5

17 – 5 → 12

12 – 5 → 7

7 – 5 → 2

2 < 5, поэтому 17 % 5 = 2

21 % 7

21 – 7 → 14

14 – 7 → 7

7 – 7 → 0

0 < 7, поэтому 21 % 7 = 0

7.6 % 2.9

7.6 – 2.9 → 4.7

4.7 – 2.9 → 1.8

1.8 < 2.9, поэтому 7.6 %2.9 = 1.8

-5 % 2

уменьшая абсолютное значение на 2, мы должны добавлять

-5 + 2 → -3

-3 + 2 → -1

|– 1 |= 1 и 1 < 2,

поэтому -5 % 2 = -1

-5 % -2

уменьшая абсолютное значение -5 абсолютным значением -2, которое есть 2

-5 – (-2) → -3

-3 – (-2) → -1

и опять 1 < 2,

поэтому -5 % -2 = -1

Операторы сравнения

- Операторы сравнения применимы ко всем числовым типам данных.

```
int p = 9;
int q = 65;
int r = -12;
float f = 9.0F;
char c = 'A';
//the following tests all return true:
p < q
f < q
f <= c
c > r
c >= q
```

Операторы instanceof

- Оператор `instanceof` тестирует класс объекта в runtime.
- Левый операнд – ссылка на произвольный объект.
- Правый операнд – класс, интерфейс или массив.

```
Object o = new String("aaa");
if (o instanceof String)
{
    System.out.println("It's a String");
}
```

Внимание! Более подробно данный оператор рассмотрен в разделе “Классы и объекты”

Операторы сравнения == и !=

- Операторы **==** и **!=** проверяют операнды на равенство
- Для примитивов – сравнение по значению

```
float f = 10.0f;
byte b = 10;
if (b == f) { // b is promoted to 10.0f
   System.out.println("True");
}
```

- Для объектов происходит сравнение ссылок
- Для сравнения объектов необходимо переопределить метод **equals()**.

Внимание! Метод **equals()** рассматривается в модуле “ООП в Java”

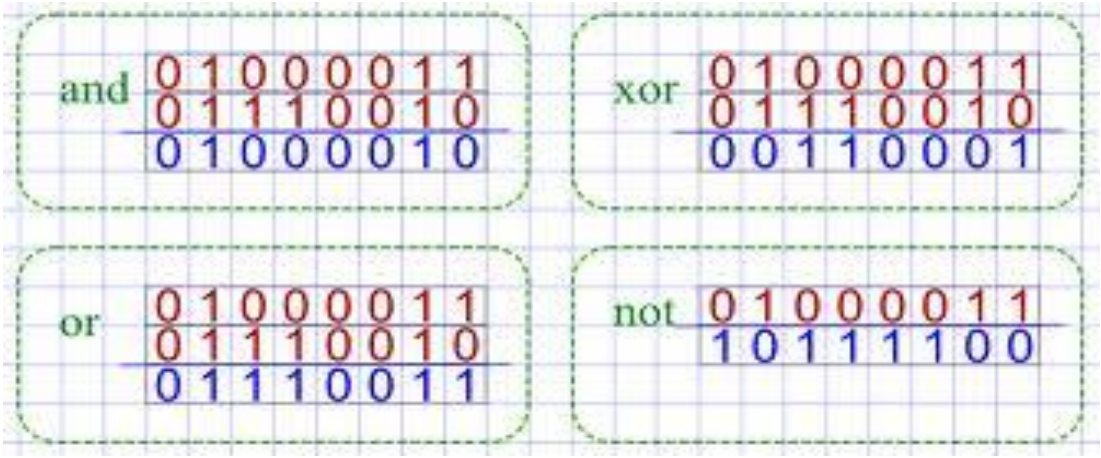
Операторы сравнения == и !=

```
Person p1 = new Person();
Person p2 = new Person();
// p1 и p2 ссылаются на разные обекты, адреса у
них разные
if (p1 != p2) {
    System.out.println("True");
}
Person p3 = p2;
// ссылка p3 создается в стеке, ее значение -
адрес объекта равный адресу p2
if (p3 == p2) {
    System.out.println("True");
}
```

Битовые операторы

- Битовые операторы **&**(битовое и), **^**(или) и **|**(исключающее или) применимы к целочисленным типам данных.

- Битовая операция вычисляет соответствующий бит базируясь но соответствующих битах операндов



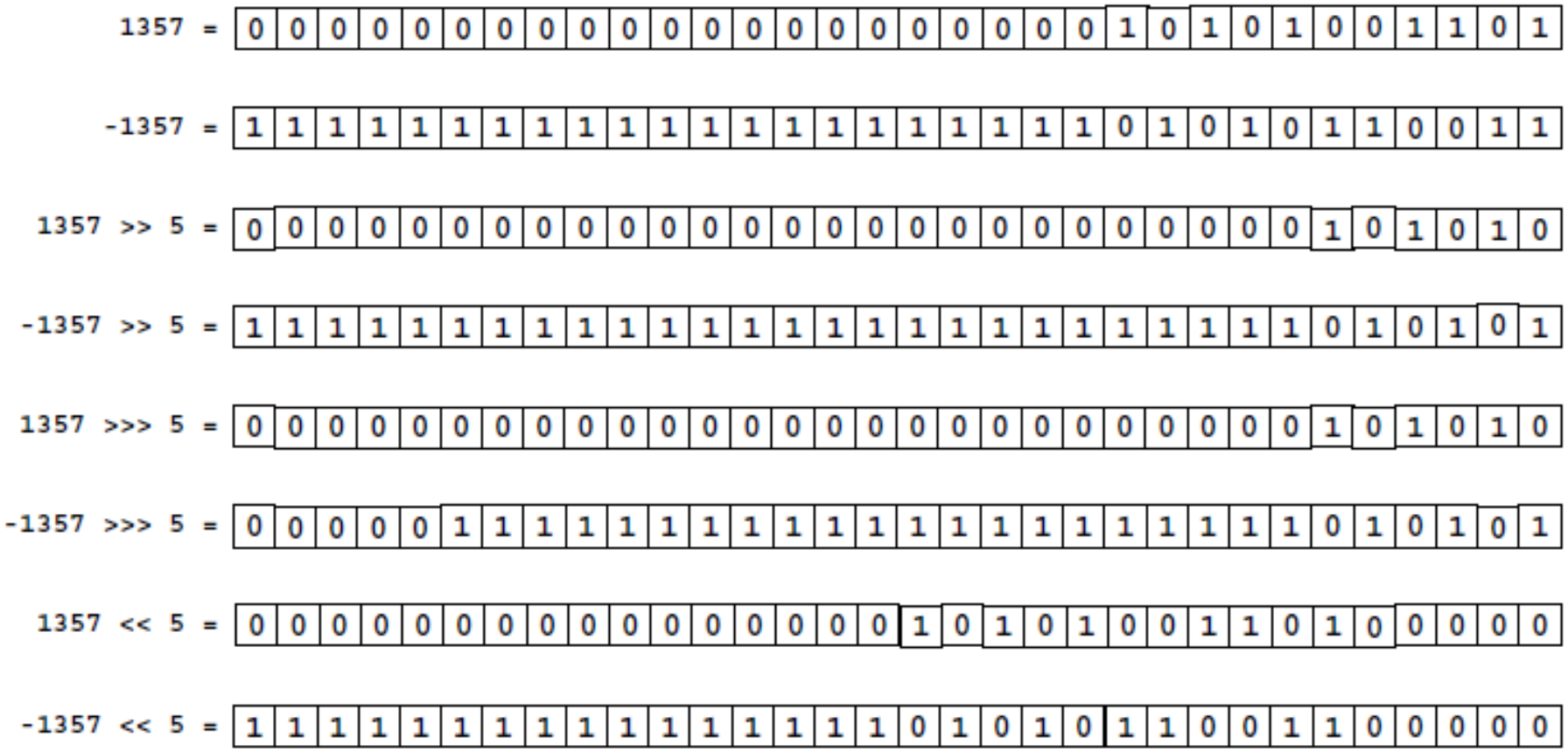
Операторы сдвига

- Операторы сдвига (<<, >>, <<<, >>>) применяются к целочисленным типам данных
- Второй операнд показывает количество разрядов, на которое нужно осуществить сдвиг в двоичном представлении.
- Сдвиг влево на разрядов эквивалентен умножению на, а сдвиг вправо - делению на то же число

```
128 << 1 // 128 * 2^1 = 256
16  << 2 // 16  * 2^2 = 64
```

Операторы сдвига

Логический и арифметический сдвиги:



Булевы операторы

- Операторы сравнения и битовые операторы (`&`, `^`, `|`) применимы и к операндам типа boolean.

- `true` трактуется как `1`, `false` как `0`.

The AND Operation on *boolean* Values

Op1	Op2	Op1 AND Op2
false	false	false
false	true	false
true	false	false
true	true	true

The XOR Operation on *boolean* Values

Op1	Op2	Op1 XOR Op2
false	false	false
false	true	true
true	false	true
true	true	false

Булевы операторы

The OR Operation on *boolean* Values

Op1	Op2	Op1 OR Op2
`false`	`false`	`false`
`false`	`true`	`true`
`true`	`false`	`true`
`true`	`true`	`true`

- Операторы `&&` и `||` применимы только к операндам типа `boolean`.
- Операторы `&&` и `||` возвращают результат “досрочно” на основе значения первого операнда, не вычисляя значение второго операнда.

Булевы операторы

- Необходимо проверить, что значение ссылки не **`null`**.

```
if (s != null) {
   if (s.length() > 20) {
      System.out.println(s);
   }
}
```

- Можно переписать в короткой форме.

```
if (s != null && s.length() > 20) {
   System.out.println(s);
}
```

Булевы операторы

- Необходимо помнить о побочных эффектах: значение правого операнда вычисляется не всегда:

```
int val = (int) (2 * Math.random());

boolean test = (val == 0) || (++val == 2);

System.out.println("test = " + test + "\nval = " +
val);
```

Тернарный оператор

- Оператор `?:` есть более короткая форма условия **`if-else`**.

- Вначале вычисляется значение слево от `?`, если оно **`true`**, то результат всего выражения – значение слева от `:`, если **`false`** – значение выражения справа.

```
a = x ? b : c;
// Эквивалентно
if (x) {
    a = b;
} else {
    a = c;
}
```

Внимание! Применять тернарный оператор нужно для повышения читаемости кода.

Оператор присваивания

- Является оператором самого низкого приоритета.
- Оператор `=` присваивает значение выражения, стоящего справа переменной, стоящей слева от знака `=`.
- Группа операторов присваивания вида `operation=` `*=`, `+=`, `-=` и т.д.
- Неявное `x operation= y` производит неявное преобразовани результата `operation` к типу переменной.

```
byte x = 2;
x += 3; // Допустимо
```

```
byte x1 = 2;
x1 = x1 + 3; // Error: Type mismatch
```

```
x1 = (byte)(x1 + 3); // Необходим явный cast
```

Примеры

```
int i =  2_000_000_000;

long j = 10_000_000_000L;

long k = i++ + ++j;

k /= i;
```

Примеры

```
int mask =  0xFF000000;

int i = mask >> 16;

int j = mask >> 24;

int k = mask >> 32;
```

Примеры

```
int m = ~0;

m >>>= 1;

int n = ~m;
```

Модуль 5

- Примитивные типы
- Декларация переменных и их инициализация
- Приведение примитивных типов
- Массивы
- Garbage collector
- Параметры методов
- Операторы в Java
- **Контроль потока выполнения**

Структурное программирование

- Программа представляет собой структуру, построенную из трех типов базовых конструкций:
  - Последовательное исполнение

    Однократное выполнение операции в том порядке, в котором они записаны в тексте программы.
  - Ветвление

    Однократное выполнение одной из двух или более операций,
    В зависимости от выполнения некоторого заданного условия.
  - Цикл

    Многократное исполнение одной и той же операции до тех пор, пока выполняется некоторое условие (условие продолжения цикла)

Внимание! Контроль потока выполнения также осуществляется с помощью исключений.

Контроль потока выполнения

- Конструкция **if/else** принимает **boolean** выражение. Если выражение вычисляется как **true**, выполняется блок **if**, если как **false**, то выполняется блок **else**.

```
if (x > 5) {
    System.out.println("x is more than 5");
} else {
    System.out.println("x is not more than 5");
}
```

Контроль потока выполнения

- Конструкция **if/else** принимает **boolean** выражение. Если выражение вычисляется как **true**, выполняется блок **if**, если как **false**, то выполняется блок **else**.

```
if (x > 5) {
    System.out.println("x is more than 5");
} else {
    System.out.println("x is not more than 5");
}
```

- Существует уточняющая конструкция (**else if**).

```
if (hours > 1700) {
    System.out.println("good evening");
} else if (hours > 1200) {
    System.out.println("good afternoon");
} else {
    System.out.println("good morning");
}
```

Контроль потока выполнения

- В случае, если необходимо осуществить более, чем 1 выбор используется конструкция **switch/case**.

```
switch (x) {
case 1:
   System.out.println("Got a 1");
   break;
case 2:
case 3:
   System.out.println("Got 2 or 3");
   break;
default:
   System.out.println("Not a 1, 2, or 3");
   break;
}
```

Контроль потока выполнения

- Выражение выбора должно быть одно из типов: **`byte`**, **`short`**, **`char`**, **`int`** или **`String`** (Java 7).

- Выражение **`default`** может быть помещено в любом месте **`switch`**, однако логичнее размещать в конце.

- Выражения **`case`** должны быть константами или константными выражениями т.е. могут быть вычислены в процессе компиляции.

- Компилятор строит специальную хэш-таблицу, позволяющую осуществить быстрый поиск.

Внимание! После выполнения **`case`** блока происходит выплнение последующих блоков. Для того чтобы это продотвратить нужно использовать **`break`**.

Контроль потока выполнения

```
switch (carModel) {
case DELUXE:
    addAirConditioning();
    addRadio();
    addWheels();
    addEngine();
    break;
case STANDARD:
    addRadio();
    addWheels();
    addEngine();
    break;
default:
    addWheels();
    addEngine();
}
```

```
// Эквивалентно
switch (carModel) {
case DELUXE:
    addAirConditioning();
case STANDARD:
    addRadio();
default:
    addWheels();
    addEngine();
}
```

Цикл while

```
while (<boolean_condition>){
<statement_or_block>
}
```

- В отличие от C, конструкция **while** в Java принимает только **boolean** выражение в качестве условия.

- **statement_or_block** будет выполняться, пока **boolean_condition** равно **true**.

```
int i = 0;
while (i < 10) {
    System.out.println(i + " squared is " + (i * i));
    i++;
}
```

Внимание! Условие вычисляется на каждой итерации.

Цикл do-while

```
do {
   <statement_or_block>
} while (<boolean_condition>);
```

- Выполнение цикла прерывается, когда **boolean_condition** вычисляется в **false**.

```
int i = 0;
do {
   System.out.println(i + " squared is " + (i * i));
   i++;
} while (i < 10);
```

Внимание! Первая итерация выполняется всегда

Цикл for

```
for (<init_expr>; <test_expr>; <alter_expr>)
    <statement_or_block>
```

- Выражение `init_expr` выполняется один раз сразу после начала работы **`for`**.
- `text_expr` должно быть **`boolean`** и вычисляется на каждой итерации.
- `alter_expr` выполняется сразу после первой итерации и всех последующих, до того как проверяется `test_expr`.

```
for (int x = 0; x < 10; x++){
    System.out.println("Value is " + x);
}
```

Цикл for

- Любая часть цикла **for** може быть опущена. Отсутствие `test_expr` эквивалентно константной величине **`true`**.

- Цикл который не имеет условия выхода

```
for(;;) {
}
```

- `init_expr` и `alter_expr` может содержать несколько выражений, разделенных запятой.

```
int j, k;
for (j = 3, k = 6; j + k < 20; j++, k += 2) {
   System.out.println("j is " + j + " k is " + k);
}
```

Цикл for

- Нельзя смешивать выражение и декларацию, а также иметь декларации разного типа.

```
int i = 7;
for (i++, int j = 0; i < 10; j++) { } // illegal!

for (int i = 7, long j = 0; i < 10; j++) { } // illegal!

for (int i = 7, j = 0; i < 10; j++) { } // legal
```

Цикл foreach

- В Java 1.5 был представлен расширенный цикл **for**, позволяющий итерировать массивы и коллекции.

```
//итерация массива:
float sumOfSquares(float[] floats) {
    float sum = 0;
    for (int i = 0; i < floats.length; i++)
        sum += floats[i];
    return sum;
}

//может быть переписана:
float sumOfSquares(float[] floats) {
    float sum = 0;
    for (float f : floats)
        sum += f;
    return sum;
}
```

Контроль потока выполнения

- Иногда необходимо прервать выполнение тела цикла, или нескольких вложенных циклов.
- Выражение **`break`** прерывает выполнение текущего блока кода (например цикла).
- Выражение **continue** прерывает выполнение текущей итерации цикла, после чего возможно выполнение последующих итераций данного цикла.

Контроль потока выполнения

```
for (int i = 0; i < array.length; i++) {
   if (array[i].secondString == null) {
      continue;
   }
}
// continue к метке:
mainLoop: for (int i = 0; i < array.length; i++)
{
   for (int j = 0; j < array[i].length; j++) {
      if (array[i][j] == '\u0000') {
         continue mainLoop;
      }
   }
}        for (int j = 0; j < array.length; j++)
         {
            if (array[j] == null) {
               break; // break out of inner loop
            }
            // process array[j]
         }
```

Модуль 5

- Примитивные типы
- Декларация переменных и их инициализация
- Приведение примитивных типов
- Массивы
- Garbage collector
- Параметры методов
- Операторы в Java
- Контроль потока выполнения

Упражнение 8

Создание серии Java-приложений