# Модуль 1-2

**Какие примитивные типы есть в Java**

• Простые типы

• Зарезервированные ключевые слова языка

• Передаются по значению

*void* – «пустой» тип

*boolean* (1 бит, 8 байт) – логический тип

*char* (2 байта, 16 бит) – символьный тип

*byte* (1 байт, 8 бит) – целочисленные типы

*short* (2 байта, 16 бит)

*int* (4 байта, 32 бита)

*long* (8 байт, 64 бита)

*float* (4 байта, 32 бита) – типы с плавающей точкой

*double* (8 байт, 64 бита)

Тип boolean прекрасно определяется множеством своих допустимых значений. Математически оно ограничивает минимальный размер значения одним битом.

Но использовать именно один бит (и ни битом более) эффективно далеко не всегда, поскольку на популярных архитектурах нельзя адресовать отдельные биты. А потому значения отдельных boolean'ов нельзя быстро сохранить в отдельные биты оперативной памяти — необходимо использовать комбинацию побитовых операций, что почти наверняка будет медленнее, чем запись целого отдельного регистра в оперативную память (размер которого может быть различным на разных платформах!).

|  |  |  |
| --- | --- | --- |
| **Тип** | **Размер в памяти (бит)** | **Диапазон значений** |
| byte | 8 бит | от -128 до 127 |
| short | 16 бит | от -32768 до 32767 |
| char | 16 бит | беззнаковое целое число, которое представляет собой символ UTF-16 (буквы и цифры) |
| int | 32 бита | от -2147483648 до 2147483647 |
| long | 64 бита | от -9223372036854775808 до 9223372036854775807 |
| float | 32 бита | от 2-149 до (2-2-23)\*2127 |
| double | 64 бита | от 2-1074 до (2-2-52)\*21023 |

**Что такое явные и неявные приведения, с чем связано их наличие?**

Неявное преобразование возможно в случаях:

• Преобразование целочисленных типов в более

емкие (byte → short → int → long)

• Преобразование char в int и long

• Преобразование целочисленных типов в типы с плавающей точкой (возможна потеря точности)

• Операторы приведения типа (typename): (int), (char)...

• При приведении более емкого целого типа к менее емкому старшие биты просто отбрасываются

• При приведении типа с плавающей точкой к целому типу дробная часть отбрасывается (никакого округления)

• Слишком большое дробное число при приведении к целому превращается в MAX\_VALUE или MIN\_VALUE

• Слишком большой double при приведении к float превращается в Float.POSITIVE\_INFINITY или Float.NEGATIVE\_INFINITY

Неявное преобразование типов выполняется в случае если выполняются условия:

Оба типа совместимы

Длина целевого типа больше или равна длине исходного типа

Существуют два типа преобразований:

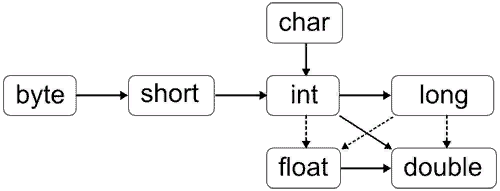
– Расширяющее преобразование (widening conversion)

– Сужающее преобразование (narrowing conversion)

Неявное преобразование всегда имеет расширяющий тип.

Сужающее преобразование это всегда явное преобразование типов.

**Какие данные мы рискуем потерять при явных приведениях?**



Например если преобразуется значение int в значение типа float, и у значения int в двоичном представлении больше чем 23 значащих бита, то возможна потеря точности, так как у типа float под целую часть отведено 23 бита. Все младшие биты значения int, которые не поместятся в 23 бита мантиссы float, будут отброшены, поэтому хотя порядок числа сохраниться, но точность будет утеряна. То же самое справедливо для преобразования типа long в тип double.

– byte, short, char в выражениях всегда повышаются до int

– если в выражении участвует тип long – то именно к этому типу будет приведён результат

– если в выражении участвует float – то результат приводится к float

– если один из операндов имеет тип double – то к этому типу будет приведён весь результат

При выборе между длиной и возможностью сохранить дробную часть – будет выбрана дробная часть.

**StringBuilder & StringBuffer**

Для работы с текстовыми данными в Java есть три класса: String, StringBuffer и StringBuilder.

В силу неизменности класса String, в результате каждой операции создаются новые экземпляры строк, а старые отбрасываются, порождая большое количество мусора.

Чтобы справиться с созданием временного мусора из-за модификаций объекта String, можно использовать класс StringBuffer.

Это mutable класс, т.е. изменяемый. Объект класса StringBuffer может содержать в себе определенный набор символов, длину и значение которого можно изменить через вызов определенных методов.

Для создания нового объекта используется один из его конструкторов, например:

StringBuffer() — создаст пустой (не содержащий символов) объект

StringBuffer(String str) — создаст объект на основе переменной str (содержащий все символы str в той же последовательности)

Конкатенация строк через StringBuffer в Java выполняется с помощью метода append.

Для работы со строками у класса StringBuffer есть ряд методов. Перечислим основные:

* delete(int start, int end) — удаляет подстроку символов начиная с позиции start, заканчивая end
* deleteCharAt(int index) — удаляет символ в позиции index
* insert(int offset, String str) — вставляет строку str в позицию offset. Метод insert также перегружен и может принимать различные аргументы
* replace(int start, int end, String str) — заменит все символы начиная с позиции start до позиции end на str
* reverse() — меняет порядок всех символов на противоположный
* substring(int start) — вернет подстроку, начиная с позиции startsubstring(int start, int end) — вернет подстроку, начиная с позиции start до позиции end

**StringBuilder** в Java — класс, представляющий последовательность символов. Он очень похож на StringBuffer во всем, кроме потокобезопасности.

StringBuilder предоставляет API, аналогичный API StringBuffer’a.

Разница лишь в том, что StringBuffer потокобезопасен, и все его методы синхронизированы, а StringBuilder — нет. Это единственная особенность.

StringBuilder в Java работает быстрее StringBuffer’а благодаря несинхронизированности методов.

### 

### 

### **String vs StringBuffer vs StringBuilder**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **String** | **StringBuffer** | **StringBuilder** |
| Изменяемость | Immutable (нет) | mutable (да) | mutable (да) |
| Расширяемость | final (нет) | final (нет) | final (нет) |
| Потокобезопасность | Да, за счет неизменяемости | Да, за счет синхронизации | Нет |
| Когда использовать | При работе со строками, которые редко будут модифицироваться | При работе со строками, которые часто будут модифицироваться в многопоточной среде | При работе со строками, которые часто будут модифицироваться, в однопоточной среде |

Логические операторы

•and && & &=

•or || | |=

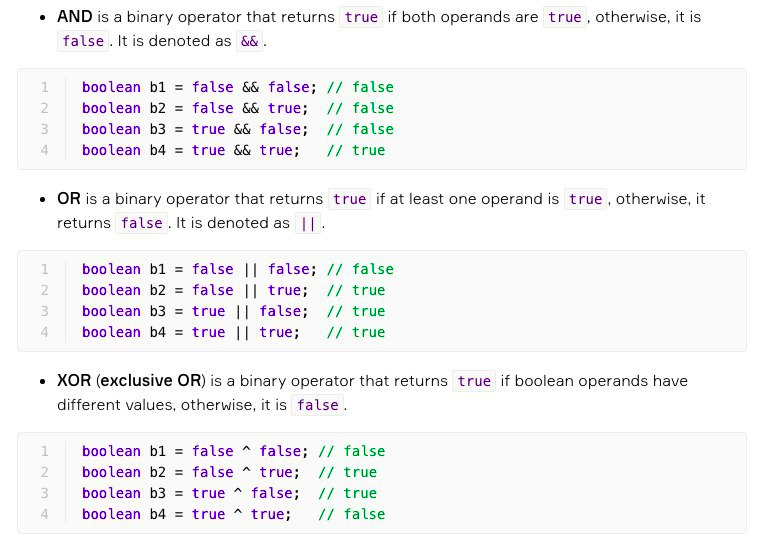
• xor ˆ ˆ=

• not !

• && и || - вычисление по сокращенной схеме

• & и | - вычисление по полной схеме

AND Operation will need to check fully if and only if the left hand operand is True , and for OR operation to evaluate fully check the left hand operand is false.



# **Сhar**

• char — 16 бит, беззнаковый (0 .. 2 в 16 ст − 1)

• Представляет номер символа в кодировке Unicode

• Литералы:

• символ в одинарных кавычках: ’a’

• шестнадцатеричный код символа: ’\u78bc’

• спецпоследовательности: ’\t’, ’\n’, ’\r’, ’\"’, ’\”, ’\\’

• Свободно конвертируется в числовые типы и обратно

**Класс-обертка для char**

• boolean isLowerCase(char)

• boolean isUpperCase(char)

• boolean isDigit(char)

• boolean isWhiteSpace(char)

• boolean isLetter(char)

• char toLowerCase(char)

• char toUpperCase(char)

• int getNumericValue(char)

• char c = Character.toLowerCase(‘a’)

# **Неизменяемые типы**

String и все классы-обертки над примитивными типами — неизменяемые.

*Integer, Byte, Character, Short, Boolean, Long, Double, Float* — все эти классы создают **Immutable** объекты. Сюда же относятся и классы, используемые для создания больших чисел — *BigInteger и BigDecimal*.

* объекты класса *java.lang.StackTraceElement*

Ссылочные типы – все классы, интерфейсы и массивы.

**Причины неизменяемости строк:**

Во-первых, экономия памяти. Неизменяемые строки можно помещать в String Pool и использовать каждый раз одну и ту же вместо создания новых.

Во-вторых, безопасность. Например, большинство логинов и паролей в любой программе — строки. Возможность их изменения могла бы повлечь проблемы с авторизацией.

Всё, что имеет модификатор **final** – тоже неизменяемо.

# **Вещественные числа**

Вещественные числа в Java представлены типами данных float и double. float является 32 битным значением с плавающей точкой, с обычной точностью, а double представляет 64 битное значение с плавающей точкой, с двойной точностью.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Тип** | **Содержит** | **По умолчанию** | **Размер** | **Диапазон** | **Обертки** |
| float | вещественное знаковое | 0.0 | 32 bits | от 1.4E−45 до 3.4028235E+38 | Float |
| double | вещественное знаковое | 0.0 | 64 bits | от 4.9E−324 до 1.7976931348623157E+308 | Double |



Литералы с плавающей точкой по умолчанию являются значениями типа **double**. При включении значения типа **float** в программу за числом следует поставить символ **f** или **F**

Поскольку типы с плавающей точкой в Java могут сводить переполнение к бесконечности, антипереполнение – к нулю, а также имеют особое NaN значение, то арифметические операции с плавающей точкой никогда не генерируют исключений, даже при выполнении недопустимых операций, например при делении нуля на нуль либо при вычислении корня отрицательного числа.

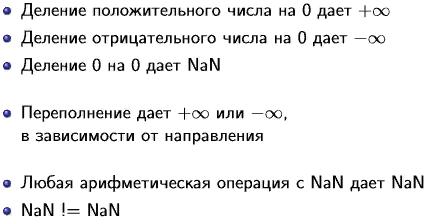
Побитовые операции с вещественными типами не поддерживаются.

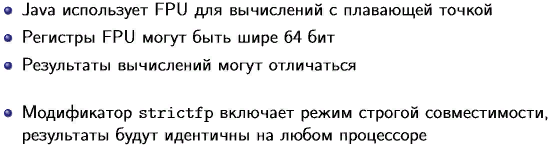
Операция деление по модулю (или остаток) определяется так же как и для целочисленных типов:

a % b = a — (a / b) \* b

Так же для операции деления по модулю справедливо следующее выражение:

a = ((long)(a/b))\*b+(a%b)





При сравнении вещественных чисел: никогда не используй == при сравнении чисел с плавающей точкой.

Нужно брать некое пороговое значение и делать Math.abs()

public class Main {

public static void main(String[] args) {

final double threshold = 0.0001;

//прибавляем к нулю 0.1 одиннадцать раз подряд

double f1 = .0;

for (int i = 1; i <= 11; i++) {

f1 += .1;

}

//Умножаем 0.1 на 11

double f2 = .1 \* 11;

System.out.println("f1 = " + f1);

System.out.println("f2 = " + f2);

if (Math.abs(f1 - f2) < threshold)

System.out.println("f1 и f2 равны");

else

System.out.println("f1 и f2 не равны");

}

}

# Еще один хороший способ сравнения вещественных чисел — использовать специальный класс BigDecimal. Этот класс специально был создан для хранения очень больших чисел с дробной частью.

# **Большие числа BigInteger и BigDecimal**

The BigInteger class is immutable which means methods of the class return new instances instead of changing existing ones.

BigInteger operations are substantially slower than operations on built-in integer types.

The class BigInteger belongs to the *java.math package*. We import it by writing the following statement:

import java.math.BigInteger;

**BigInteger number = new BigInteger("62957291795228763406253098");**

It is also possible to create an instance by passing a long value to the method valueOf:

**BigInteger number = BigInteger.valueOf(1\_000\_000\_000);**

In addition, the class has several useful constants:

**BigInteger zero = BigInteger.ZERO; // 0**

**BigInteger one = BigInteger.ONE; // 1**

**BigInteger ten = BigInteger.TEN; // 10**

## **Methods of BigInteger**

BigInteger eleven = ten.add(one);

BigInteger nine = ten.subtract(BigInteger.ONE); // 10 - 1 = 9

BigInteger oneHundredTen = ten.multiply(eleven); // 10 \* 11 = 110

BigInteger twelve = oneHundredTen.divide(nine); // integer division: 12

The method negate returns a new BigInteger with the changed sign, like this:

nine.negate(); // -9

The method divideAndRemainder returns an array consisting of two numbers: the result of integer division and the remainder.

BigInteger[] pair = oneHundredTen.divideAndRemainder(nine); // 12 and 2

The class also provides methods for performing more complex math operations. The method abs returns a new BigInteger whose value is the absolute value of this BigInteger.

BigInteger number = new BigInteger("-8");

System.out.println(number.abs()); // 8

The method gcd returns the greatest common divisor of two numbers.

BigInteger three = BigInteger.valueOf(3);

BigInteger six = BigInteger.valueOf(6);

System.out.println(three.gcd(six)); // 3

Также присутствуют методы для более сложных (специфических) операций:

* операции с вычислением [mod](https://habr.com/ru/post/421071/):

BigInteger firstValue = new BigInteger("-34");

BigInteger secondValue = new BigInteger("5");

* BigInteger resultValue = firstValue.mod(secondValue); //1

Существуют несколько разных вариаций данной функции:

* получение рандомного числа с заданием количества битов, которое будет использовать полученное значение:

BigInteger firstValue = BigInteger.probablePrime(8, new Random());//211

* BigInteger secondValue = BigInteger.probablePrime(16, new Random());//42571
* операции побитовых сдвигов(this >> n)  
  Сдвиг влево:

BigInteger firstValue = new BigInteger("5");

* BigInteger firstResultValue = firstValue.shiftLeft(3);//40
* Сдвиг вправо:

BigInteger secondValue = new BigInteger("34");

* BigInteger secondResultValue = secondValue.shiftRight(2); //8

## 

## **BigDecimal в Java**

Когда нам нужно вещественное число произвольной длины, используется класс Java — BigDecimal. Как правило его применяют для работы с финансами вместо double, так как он дает больше возможности настройки.

Как и BigInteger, BigDecimal является потомком класса Number и имеет методы, возвращающие значение объекта в виде определенного примитивного типа:

BigDecimal value = new BigDecimal(35563.3);

long longValue = value.longValue();//35563

double doubleValue = value.doubleValue();//35563.3

Как мы можем видеть при приведении к long, остаётся только целая часть, а знаки после запятой отбрасываются.

### **Конструкторы BigDecimal**

С конструкторами BigDecimal мы ознакомимся подробнее, так как у класса есть их гораздо более широкий выбор. Есть конструкторы, позволяющие задать значение объекта разными способами (передачей int, long, double, String и даже BigInteger), а есть такие, которые позволяют задавать настройки создаваемого объекта (способы округления, количество знаков после запятой):

BigDecimal firstValue = new BigDecimal("455656.545");//455656.545

Тут всё понятно, мы непосредственно задали значение и количество знаков после запятой, которые хотим видеть.

BigDecimal secondValue = new BigDecimal(3445.54);//3445.5399999999999636202119290828704833984375

Результаты этого конструктора могут быть весьма непредсказуемыми, ведь мы задаем double, который по своей природе весьма неоднозначный тип. Поэтому обычно рекомендуется использовать в конструкторе String.

BigDecimal thirdValue = new BigDecimal(3445.554645675444, MathContext.DECIMAL32);//3445.555

Мы задаем double, но при этом и задаём параметр, описывающий правило округления (который содержит количество знаков после запятой и алгоритм для округления).

char[] arr = new String("455656.545").toCharArray();

BigDecimal fourthValue = new BigDecimal(arr, 2, 6);//5656.5

Задаем массив знаков, с которого элемента берём значения для объекта и сколько этих элементов берём.

BigDecimal fifthValue = new BigDecimal(new BigInteger("44554"), 3);//44.554

Берем уже существующий объект BigInteger, задаём какое количество знаков после запятой.

### **Методы BigDecimal**

Класс BigDecimal также содержит в себе методы для разнообразных арифметических операций, но методов работы с битами, как у BigInteger, у него нет.

Но тем не менее, главная фишка BigDecimal — гибкость в работе с числами с плавающей запятой.

Давайте рассмотрим некоторые методы, которые дают нам возможность властвовать над вещественными числами:

* получаем точность (количество чисел):

BigDecimal value = new BigDecimal("454334.34334");

* int result = value.precision();//11
* задаем количество знаков после запятой и правило округления:

BigDecimal firstValue = new BigDecimal(3445.544445);

* BigDecimal secondValue = firstValue.setScale(3,BigDecimal.ROUND\_CEILING);//3445.545
* Немного ниже мы рассмотрим подробнее константы для задания правил округления.
* делим BigDecimal на другой BigDecimal, при этом указывая необходимое количество знаков после запятой и правило округления:

BigDecimal firstValue = new BigDecimal("455656.545");

BigDecimal secondValue = new BigDecimal(3445.544445);

* BigDecimal result = firstValue.divide(secondValue, 2,RoundingMode.DOWN);//132.24
* перемещение запятой вправо/влево на определенное количество знаков:

BigDecimal value = new BigDecimal("455656.545");

BigDecimal firstResult = value.movePointRight (2);//45565654.5

* BigDecimal secondResult = value.movePointLeft (2);//4556.56545
* обрезаем конечные нули:

BigDecimal value = new BigDecimal("45056.5000");

* BigDecimal result = value.stripTrailingZeros();//45056.5
* Если же у нас в вещественной части все нули и в целой тоже есть (или у нас и вовсе нет знаков после запятой), тогда:

BigDecimal value = new BigDecimal("450000.000");

* BigDecimal result = value.stripTrailingZeros();//4.5E+5

### **Правила округления BigDecimal**

Для задания правил округления, внутри BigDecimal мы можем увидеть специальные константы, описывающие алгоритмы округления:

ROUND\_UP — округления от нуля, округление в сторону вещественной части:

BigDecimal firstValue = new BigDecimal("2.578");

BigDecimal firstResult = firstValue.setScale(1, BigDecimal.ROUND\_UP );//2.6

BigDecimal secondValue = new BigDecimal("-2.578");

BigDecimal secondResult = secondValue.setScale(1, BigDecimal.ROUND\_UP );//-2.5

ROUND\_DOWN — округления до нуля, то есть усечение вещественной части:

BigDecimal firstValue = new BigDecimal("2.578");

BigDecimal firstResult = firstValue.setScale(1, BigDecimal.ROUND\_DOWN );//2.5

BigDecimal secondValue = new BigDecimal("-2.578");

BigDecimal secondResult = secondValue.setScale(1, BigDecimal.ROUND\_DOWN );//-2.6

ROUND\_CEILING — округления до положительной бесконечности. То есть, если число у нас положительное, то -> ROUND\_UP, если отрицательное, то -> ROUND\_DOWN

BigDecimal firstValue = new BigDecimal("2.578");

BigDecimal firstResult = firstValue.setScale(1, BigDecimal.ROUND\_CEILING);//2.6

BigDecimal secondValue = new BigDecimal("-2.578");

BigDecimal secondResult = secondValue.setScale(1, BigDecimal.ROUND\_CEILING);//-2.5

ROUND\_FLOOR — округления до отрицательной бесконечности, то есть если число у нас положительное, то -> ROUND\_DOWN, если отрицательное, то -> ROUND\_UP

BigDecimal firstValue = new BigDecimal("2.578");

BigDecimal firstResult = firstValue.setScale(1, BigDecimal.ROUND\_FLOOR);//2.5

BigDecimal secondValue = new BigDecimal("-2.578");

BigDecimal secondResult = secondValue.setScale(1, BigDecimal.ROUND\_FLOOR);//-2.6

Для рассматриваемого значения это ближайшее число с урезанным знаком после запятой будем рассматривать как ближайшего соседа рассматриваемого числа.

Например, 2.43 будет ближе к 2.4, чем к 2.5, но 2.48 будет уже ближе к 2.5.

ROUND\_HALF\_DOWN — округления до «ближайшего соседа». Если оба соседа равноудалены от конкретного значения, тогда производится округление к нулю. Равноудалены — это, например, когда округляемое число — 5, и оно находится на одинаковом расстоянии от 0 и 10):

BigDecimal firstValue = new BigDecimal("2.58");

BigDecimal firstResult = firstValue.setScale(1, BigDecimal.ROUND\_HALF\_DOWN );//2.6

BigDecimal secondValue = new BigDecimal("2.55");

BigDecimal secondResult = secondValue.setScale(1, BigDecimal.ROUND\_HALF\_DOWN );//2.5

ROUND\_HALF\_UP — режим для округления в сторону «ближайшего соседа». Если оба соседа равноудалены, округление выполняется к большему числу (это то самое округление, которому нас обучали в школе):

BigDecimal firstValue = new BigDecimal("2.53");

BigDecimal firstResult = firstValue.setScale(1, BigDecimal.ROUND\_HALF\_UP );//2.5

BigDecimal secondValue = new BigDecimal("2.55");

BigDecimal secondResult = secondValue.setScale(1, BigDecimal.ROUND\_HALF\_UP );//2.6

ROUND\_HALF\_EVEN — округления до «ближайшего соседа», если оба соседа не равноудалены. В этом случае если перед округляемым числом стоит нечетное, выполняется округление в большую сторону, если чётное — в меньшую:

BigDecimal firstValue = new BigDecimal("2222.2225");

BigDecimal secondValue = firstValue.setScale(3,BigDecimal.ROUND\_HALF\_EVEN );//2222.222

Такой результат мы получаем, потому что при округлении 5 смотрит на предыдущее число 2, и видя, что оно чётное, округление идет в меньшую сторону.

Но если:

BigDecimal firstValue = new BigDecimal("2222.22255");

BigDecimal secondValue = firstValue.setScale(3,BigDecimal.ROUND\_HALF\_EVEN );//2222.223

То округление идёт в большую сторону, так как последняя 5 смотрит на предыдущее значение и видит нечётное число. Как следствие, число округляется в большую сторону до 6, после чего следующая 6 тоже идёт на округление. Но шестерка уже не смотрит на число слева, так как число явно ближе в большую сторону, и в итоге последняя 2 увеличивается на 1.

ROUND\_UNNECESSARY — используется для проверки, что число в округлении не нуждается. То есть, мы проверяем, что число имеет нужное нам количество знаков после запятой:

BigDecimal firstValue = new BigDecimal("2.55");

BigDecimal firstResult = firstValue.setScale(2, BigDecimal.ROUND\_UNNECESSARY);//2.55

Тут всё хорошо, значение имеет два знака и мы проверяем, что после запятой только два знака.

Но если:

BigDecimal secondValue = new BigDecimal("2.55");

BigDecimal secondResult = secondValue.setScale(1, BigDecimal.ROUND\_UNNECESSARY);

То мы получим — ArithmeticException, так как проверяемое значение превышает заданное количество знаков после запятой.

Но если мы будем проверять два знака после запятой, а по факту там есть один, то исключение не выпадет, а недостающие знаки просто дополняются нулями:

BigDecimal thirdValue = new BigDecimal("2.5");

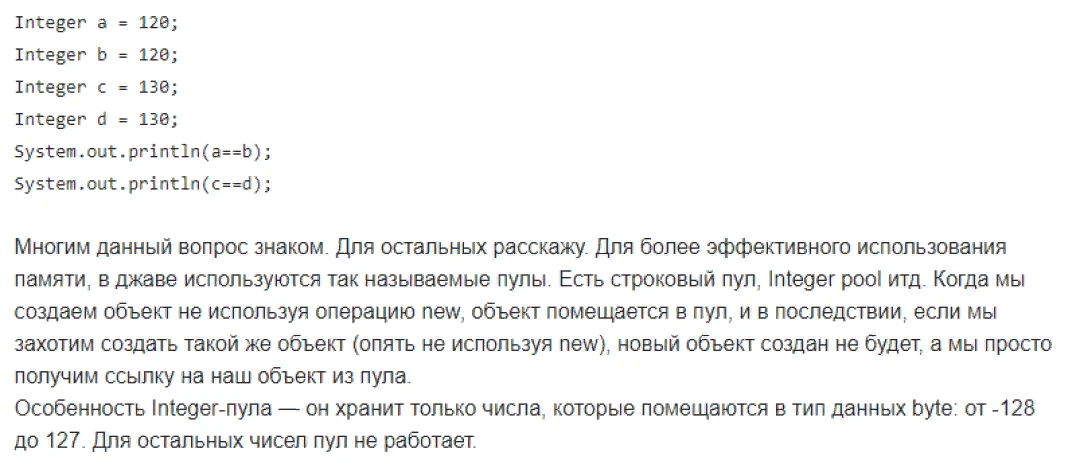
BigDecimal thirdResult = thirdValue.setScale(3, BigDecimal.ROUND\_UNNECESSARY );//2.500

Еще хотелось бы отметить что у BigDecimal есть константы, аналогичные константам BigInteger ZERO, ONE и TEN. Вот ссылка на [документацию](https://docs.oracle.com/javase/8/docs/api/java/math/BigDecimal.html#ROUND_CEILING).

И напоследок: как вы, наверное, заметили, выполняя операции с объектами BigInteger и BigDecimal, мы не изменяем старые, а всегда получаем новые. Это нам говорит о том, что они — immutable, то есть неизменные после их создания, как и String.

Иными словами, все их методы не могут изменить внутреннее состояние объекта, максимум — вернуть новый объект с параметрами, заданными используемой нами функцией.

# **Integer pool**



# **String pool**

Пул строк (**String Pool**) — это множество строк в куче (**[Java Heap Memory](https://javadevblog.com/chto-takoe-heap-i-stack-pamyat-v-java.html)**).

Однако использование оператора new заставляет класс String создать новый объект String. После этого можем использовать метод intern(), чтобы поместить этот объект в пул строк или обратиться к другому объекту из пула строк, который имеет такое же значение.

Ниже приведена программа, которая демонстрирует работу с пулом строк:

StringPool.java

public class StringPool {

public static void main(String[] args) {

String s1 = "Cat";

String s2 = "Cat";

String s3 = new String("Cat");

// сравниваем наши строки

System.out.println("s1 == s2 : " + ( s1 == s2 ) );

System.out.println("s1 == s3 : " + ( s1 == s3 ) );

}

}

Результат выполнения программы: s1 == s2 : true | s1 == s3 : false

# 

# **String**

• Последовательность символов произвольной длины

• Класс java.lang.String

• Не то же, что массив символов

• Никаких нулевых символов в конце, длина хранится отдельно

• Строки неизменяемы!

• String hello = " Hello";

• String specialChars = "\r\n\t\"\\ \u0101\u2134\u03ff ";

• char[] charArray = {’a’, ’b’, ’c’};

• String string = new String (charArray);

ДОСТУП К СОДЕРЖИМОМУ

• int length()

• char charAt(int index)

• char[] toCharArray()

• String substring(int beginIndex)

• String substring(int beginIndex, int endIndex)

СРАВНЕНИЕ СТРОК

• Оператор == сравнивает ссылки, а не содержимое строки

• boolean equals(Object anObject)

• boolean equalsIgnoreCase(String anotherString)

• int compareTo(String anotherString)

• int compareToIgnoreCase(String anotherString)

ОПЕРАЦИИ

• boolean startsWith(String prefix)

• boolean endsWith(String suffix)

• int indexOf(String str)

• int lastIndexOf(String str)

• String trim()

• String replace(char oldChar, char newChar)

• String toLowerCase()

• String toUpperCase()

КОНКАТЕНАЦИЯ СТРОК

• String concat(String str)

• Оператор + (работает, как StringBuilder)

String helloWorld = " Hello " + " World!";

• java.lang.StringBuilder

StringBuilder buf = new StringBuilder(); buf.append ("Hello");

buf.append ("World");

buf.append (’!’);

String result = buf.toString()

# **Многомерные массивы**

Массивы, элементами которых являются другие массивы. Такие массивы называются многомерными.

Объявление многомерных массивов:

int[][] twoDimArray; //двумерный массив

String[][][] threeDimArray; //трёхмерный массив

double[][][][][] fiveDimArray; // пятимерный массив

Процедура объявления и создания двумерного массива практически такая же, как и в случае одномерного:

int[][] twoDimArray = new int[3][4];

Этот массив имеет 3 строки и 4 столбца.

Можно провести процедуру инициализации быстрее:

int [][] twoDimArray = {{5,7,3,17}, {7,0,1,12}, {8,1,2,3}};

И в том, и в другом случае мы получим двумерный массив с тремя строками и четырьмя столбцами, заполненный целыми числами.



## **Вывод двумерного массива на экран**

Эту операцию логичнее всего делать так: сначала выводим нулевую строку поэлементно, затем — вторую и так далее. Чаще всего в Java вывод двумерного массива реализуют с помощью двух вложенных циклов.

int [][] twoDimArray = {{5,7,3,17}, {7,0,1,12}, {8,1,2,3}};//объявили массив и заполнили его элементами

for (int i = 0; i < 3; i++) { //идём по строкам

for (int j = 0; j < 4; j++) {//идём по столбцам

System.out.print(" " + twoDimArray[i][j] + " "); //вывод элемента

}

System.out.println();//перенос строки ради визуального сохранения табличной формы

}

## **Быстрый вывод двумерного массива**

Самый короткий способ вывести список элементов двумерного массива на экран — применение метода deepToString класса Arrays. Пример:

int[][] myArray = {{18,28,18},{28,45,90},{45,3,14}};

System.out.printLn(Arrays.deepToString(myArray));

Результат работы программы — следующий вывод:

*[[18, 28, 18], [28, 45, 90], [45, 3, 14]]*

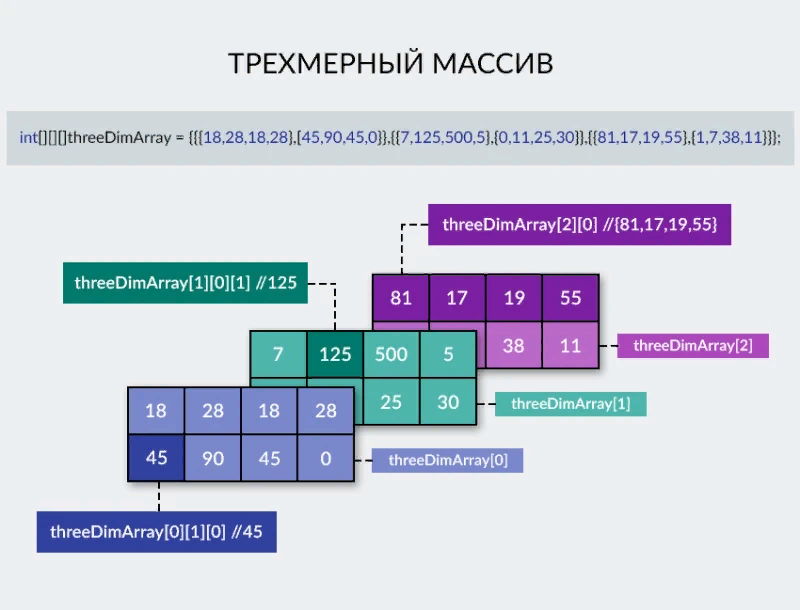
## **“Длины” двумерного массива**

int [][] twoDimArray = {{5,7,3,17}, {7,0,1,12}, {8,1,2,3}};

System.out.println(twoDimArray.length);

Вывод: *3*

Таким образом, эта операция выводит количество строк в массиве. А как получить количество столбцов? Если мы имеем дело с прямоугольными двумерными массивами (то есть такими, у которых все строки одинаковой длины), то можно применить операцию twoDimArray[0].length или вместо нулевого элемента (по сути — нулевой строки) — любой другой существующий. Мы можем так поступить, потому что в Java двумерный массив — это массив массивов, и нулевой элемент twoDimArray[0] — это массив длины 4.



# 

# **Побитовые операции**

Бит — минимальная единица измерения информации в компьютере. Его название происходит от английского “*binary digit*” — “двоичное число”. Бит может быть выражен одним из двух чисел: 1 или 0.

Существует специальная система счисления, основанная на единицах и нулях — двоичная. Любое число в Java можно сконвертировать в его двоичную форму. Для этого нужно использовать классы-обертки.

Вот как можно сделать это для числа int:

public class Main {

public static void main(String[] args) {

int x = 342;

System.out.println(Integer.toBinaryString(x));

}

}

Вывод в консоль: 101010110

## 

## 

## 

## 

## **Приоритет операций в Java**

Operator Precedence

|  |  |
| --- | --- |
| **Operators** | **Precedence** |
|  |  |
| postfix | expr++ expr-- |
| unary | ++expr --expr +expr ~ ! |
| Multiplicative | \* / % |
| additive | + - |
| shift | << >> >>> |
| relational | < > <= >= instanceof |
| equality | == != |
| bitwise AND | & |
| bitwise exclusive OR | ^ |
| bitwise inclusive OR | | |
| logical AND | && |
| logical OR | || |
| ternary | ? : |
| assignment | = += -= \*= /= %= &= ^= |= <<= >>= >>>= |

Все операции выполняются слева направо, однако с учетом своего приоритета.

Существует несколько побитовых операторов, применимых к целочисленными типам **long**, **int**, **short**, **char**, **byte**.

|  |  |
| --- | --- |
| ~ | Побитовый унарный оператор NOT |
| & | Побитовый AND |
| &= | Побитовый AND с присваиванием |
| | | Побитовый OR |
| |= | Побитовый OR с присваиванием |
| ^ | Побитовый исключающее OR |
| ^= | Побитовый исключающее OR с присваиванием |
| >> | Сдвиг вправо |
| >>= | Сдвиг вправо с присваиванием |
| >>> | Сдвиг вправо с заполнением нулями |
| << | Сдвиг влево |
| <<= | Сдвиг влево с присваиванием |
| >>>= | Сдвиг вправо с заполнением нулями с присваиванием |