HIGHER SCHOOL OF ECONOMICS
NATIONAL RESEARCH UNIVERSITY

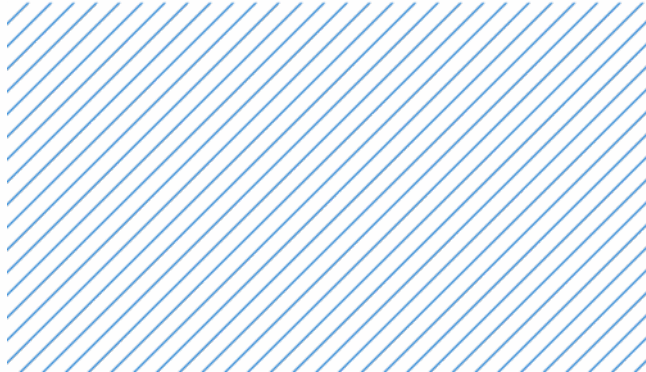
Лекция 4

Основы процедурного программирования на Java (часть 2)

Программирование на языке Java

Роман Гуров

ВШЭ БИ 2021



Сложение двоичных чисел

Как самостоятельно сложить два числа в двоичной записи?

Да так же, как в начальной школе – в столбик!

1	0	0	1	1
0	0	0	0	1
1	0	1	0	0

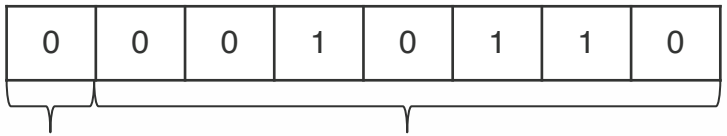
19 + 1 = 20

1	0	1	1	0
0	0	1	1	1
1	1	1	0	1

22 + 7 = 29

Представление целых чисел в памяти

Рассмотрим как хранится в памяти переменная типа `byte`:



= 22

Самый старший бит отвечает за знак:

`0: +`

`1: -`

Остальные 7 кодируют само значение

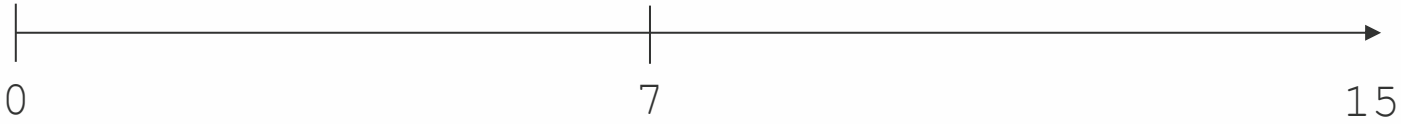
Для положительных чисел это просто двоичное представление

Отрицательные числа представляются иначе: *дополнительным кодом*

Дополнительный код

Для простоты будем использовать 4-битную переменную

Рассмотрим как бы она выглядела, если бы вообще не имела знака



Вся первая половина значений имеет старший бит 0

0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

А вся вторая половина – старший бит 1

Дополнительный код

А что вообще произойдёт при переполнении переменной?

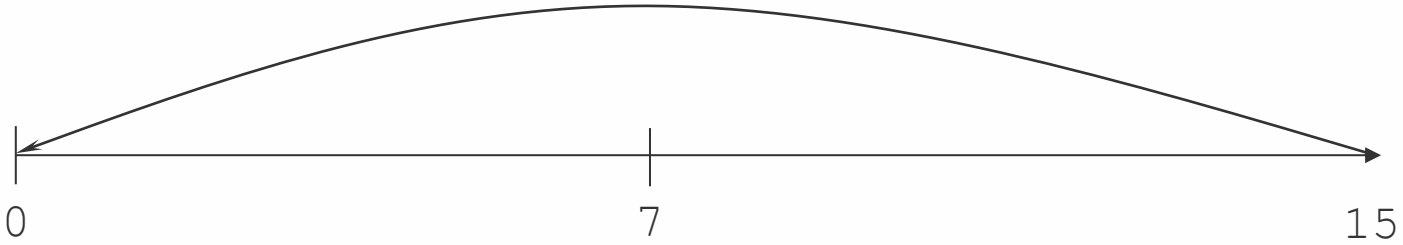
Вычислим в столбик такую сумму:

Т. к. пятый бит не влезает, он просто отбрасывается

	1	1	1	1
	0	0	0	1
1	0	0	0	0

+

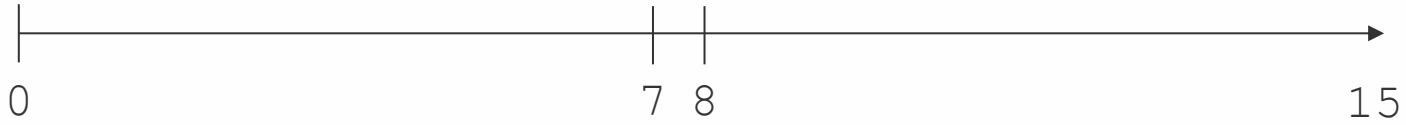
=



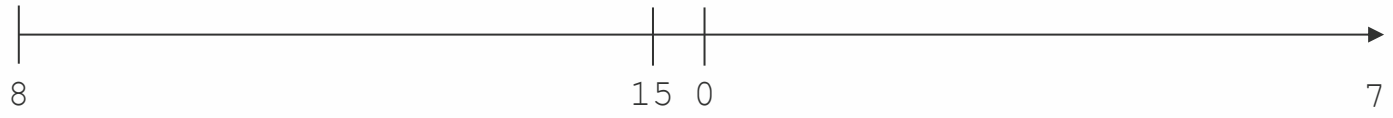
Получается, что мы “зациклились” – вернулись в самое начало к нулю

Дополнительный код

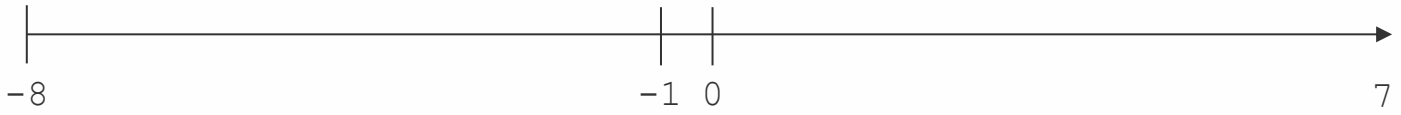
Теперь попробуем перейти к знаковым числам



Перенесём вторую половину значений в начало интервала – слева от нуля
Так как мы всё равно зацикливаемся, никакого смысла мы не теряем: после 7 всё так же идёт 8

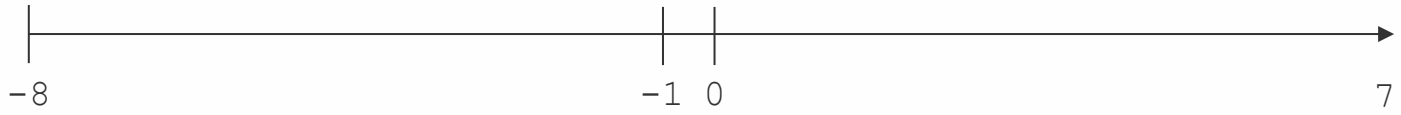


А теперь просто возьмём и переименуем левую часть в отрицательные числа:



Дополнительный код

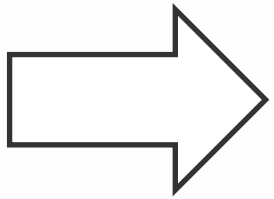
А теперь просто возьмём и переименуем левую часть в отрицательные числа:



Битовые представления при этом остались те же

0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111



0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

-8	1000
-7	1001
-6	1010
-5	1011
-4	1100
-3	1101
-2	1110
-1	1111

Самый смысл в том, что в такой записи мы всё ещё честно зацикливаемся при арифметических операциях

Такая запись отрицательных чисел и называется *дополнительный код*

Числа друг друга дополняют:
Если тупо сложить битовые представления -1 и 1, получим ровно 0

И вообще сложение не требует никаких знаний о знаке числа, достаточно просто сложить битовые представления

Побитовые операторы

Пусть A = 0011 1100
B = 0000 1101

Оператор	Описание	Пример
& (побитовый И)	Применяет логическое И к битам операндов	`(A & B) == 0000 1100`
| (побитовый ИЛИ)	Применяет логическое ИЛИ к битам операндов	`(A | B) == 0011 1101`
^ (побитовый XOR)	Применяет логическое ИСКЛЮЧАЮЩЕЕ ИЛИ к битам операндов	`(A ^ B) == 0011 0001`
~ (побитовый НЕ)	Переворачивает каждый бит числа на противоположный	`(~A) == 1100 0011`
<< (битовый сдвиг влево)	Биты левого операнда сдвигаются влево на число позиций, задаваемое правым операндом	`(A << 2) == 1111 0000`
>> (битовый сдвиг вправо)	Биты левого операнда сдвигаются вправо на число позиций, задаваемое правым операндом	`(A >> 2) == 0000 1111`
>>> (беззнаковый битовый сдвиг вправо)	Биты левого операнда сдвигаются вправо на число позиций, задаваемое правым операндом	`(A >>> 2) == 0000 1111`

Побитовые операторы

Оператор	Описание	Пример
<< (битовый сдвиг влево)	Биты левого операнда сдвигаются влево на число позиций, задаваемое правым операндом	`(A << 2) == 1111 0000`
>> (битовый сдвиг вправо)	Биты левого операнда сдвигаются вправо на число позиций, задаваемое правым операндом	`(A >> 2) == 0000 1111`
>>> (беззнаковый битовый сдвиг вправо)	Биты левого операнда сдвигаются вправо на число позиций, задаваемое правым операндом	`(A >>> 2) == 0000 1111`

Сдвиг влево вытягивает из-за границы числа нули:

```
(0000 0001 << 1) == 0000 0010
```

Беззнаковый сдвиг вправо тоже:

```
(1100 0001 >>> 3) == 0001 1000
```

Знаковый сдвиг вправо ведёт себя иначе, он копирует самый левый бит:

```
(1100 0001 >> 3) == 1111 1000
(0100 0001 >> 3) == 0000 1000
```

Представление вещественных чисел в памяти

Рассмотрим представление `float` в памяти компьютера:

Вспомним такую запись чисел: $2{,}77 \cdot 10^{-25}$

мантисса

экспонента

естественно, основание будет уже двойкой

`float` работает примерно так же

Точность вещественных чисел

Особенности представления дробных чисел приводят к неточностям:

```
double a = 0.1;
a += 0.1;
a += 0.1;
System.out.println(a);
```

```
0.30000000000000004
```

Но самое страшное:

```
double a = 0.1;
a += 0.1;
a += 0.1;
System.out.println(a == 0.3);
```

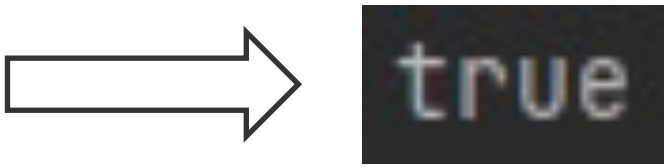
```
false
```

Точность вещественных чисел

Для корректного сравнения чисел нужно допускать некоторую погрешность ε.

Например, числа a и b равны, если:
$|a - b| < \varepsilon$

```java
double eps = 1e-6; // 1 * 10^-6
double a = 0.1;
a += 0.1;
a += 0.1;
System.out.println(Math.abs(a - 0.3) < eps);
```



Точность вещественных чисел

Теперь, решим задачу поиска корней квадратного уравнения “правильно”

```
public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    float a = scanner.nextFloat();
    float b = scanner.nextFloat();
    float c = scanner.nextFloat();

    float eps = 1e-6f;

    if (Math.abs(a) < eps) {
        // Решаем линейное уравнение :)
        return;
    }

    float D = b * b - 4 * a * c;

    if (Math.abs(D) < eps) {
        System.out.println(-b / (2 * a));
    } else if (D >= eps) {
        float x_1 = (-b - (float)Math.sqrt(D)) / (2 * a);
        float x_2 = (-b + (float)Math.sqrt(D)) / (2 * a);
        System.out.println(x_1);
        System.out.println(x_2);
    } else {
        System.out.println("No roots");
    }
}
```

Условный оператор

Условный (тернарный) оператор позволяет кратко записывать подобные выражения:

```
int a;

if (x > 20) {
    a = x + 1;
} else {
    a = x * 15;
}
```

```
int a = x > 20 ? x + 1 : x * 15;
```

Общий вид таков:

`<условие> ? <выражение когда истина> : <выражение когда ложь>`

В Java с `void`-функциями тернарный оператор не работает!

```
int n = 20;
n > 10 ? System.out.println("n is big") : System.out.println("n is small");
```

Not a statement

Используйте `if`

switch / case

Бывает, приходится писать подобные конструкции:

```
if (n == 1) {
    System.out.println("Нечто");
}
else if (n == 5) {
    System.out.println("Что-то");
}
else {
    System.out.println("Не пойми что");
}
```

Её же можно выразить через `switch / case`

```
switch (n) {
    case 1:
        System.out.println("Нечто");
        break;
    case 5:
        System.out.println("Что-то");
        break;
    default:
        System.out.println("Не пойми что");
        break;
}
```

Проверяется, подходит ли n под какой-то из перечисленных кейсов. Если есть default, туда попадают все случаи, для которых нет кейса

switch / case

break можно не писать, тогда исполнение пойдёт в следующий кейс

Перечислим, какие месяцы остались до конца года, начиная с переданного:

```
int month = 8;

switch (month) {
    case 1:  System.out.println("January");
    case 2:  System.out.println("February");
    case 3:  System.out.println("March");
    case 4:  System.out.println("April");
    case 5:  System.out.println("May");
    case 6:  System.out.println("June");
    case 7:  System.out.println("July");
    case 8:  System.out.println("August");
    case 9:  System.out.println("September");
    case 10: System.out.println("October");
    case 11: System.out.println("November");
    case 12: System.out.println("December");
        break;
    default:
        System.out.println("Incorrect month code");
        break;
}
```

```
August
September
October
November
December
```

Литералы

Литерал – значение, написанное в коде явно

```
int a = 5;
int b = 07774;       // восьмеричная запись
int c = 0xFF00BB;    // шестнадцатеричная запись
int d = 0b10101;     // двоичная запись
long e = 132534L;
long f = 0xCAEF00L;
int g = 123_456_789;
long h = 123_456_789_000_000L;
float i = 0.f;
float j = 5f;
float k = 3.14e20f; // 3.14 * 10^20
double l = 5.;
double m = 3.14;
double n = 10d;
double o = 22e-13;
double p = 2.7e0D;
char q = 'q';
System.out.println("I'm a string literal.");
```

Литералы

В строчных и символьных литералах можно передавать некоторые символы особым образом, через обратный слеш:

```
System.out.println("I can skip line\nLike this");
System.out.println("If I want double quotes, I escape them with \\ \"like this\"");
char quote = '\'';
```

```
I can skip line
Like this
If I want double quotes, I escape them with \ "like this"
```

Область видимости переменных

Переменные определяются только на некоторую конкретную область, в которой они живут и их можно “увидеть”

```
public static void main(String[] args) {
    int x = 5;
    if (true) {
        int y = x; // видно
    }
    int z = y;  // не видно
}
```

Если не хочется, чтобы набор переменных растёкся на всё тело функции, можно явно задать область видимости:

```
int x = 5;
{
    int y = 10;
    int z = x + y; // видно
    // ...
}
int f = y + z;  // не видно
```

Ошибки компиляции

Я что-то нажал, и всё сломалось

```
float x = 10.5;
int i = x * 2;
if (i)
    System.out.println(x, i);
```

```
java: incompatible types: possible lossy conversion from double to float
java: incompatible types: possible lossy conversion from float to int
java: incompatible types: int cannot be converted to boolean
java: no suitable method found for println(float,int)
    method java.io.PrintStream.println() is not applicable
      (actual and formal argument lists differ in length)
    method java.io.PrintStream.println(boolean) is not applicable
      (actual and formal argument lists differ in length)
    ...
```

Ошибки компиляции

Я что-то нажал, и всё сломалось

Можно разделить ошибки компиляции на три вида:

Лексические – появляются на ранней стадии компиляции, когда проверяется, что код состоит из правильных слов

```
public class Main {
    public static void main(String[] args) {
        int q = 10000000000000000000000000;

        int 123a = 1;
    }
}
```

`int` литерал не может быть таким большим

имя переменной не может начинаться с цифры

Ошибки компиляции

Я что-то нажал, и всё сломалось

Синтаксические – появляются на стадии компиляции, когда проверяется общая корректность структуры кода

```
public class Main {
    public static void main(String[] args) {
        int q = 5
        int y = (3 + 5;
    }
}
```

забыли точку с запятой

не закрыли скобку

Ошибки компиляции

Я что-то нажал, и всё сломалось

Семантические – ошибки уже в какой-то более умной логике кода

```
public class Main {
    public static void main(String[] args) {
        int q = 5.5;
        boolean b = true - false;
    }
}
```

попытка присвоить `float` к `int`

оператор “–” не определён для `boolean`

Класс Math

Для математических функций в Java есть специальный класс Math

```
float x = 27;
double square_root = Math.sqrt(x);
float cubic_root = (float)Math.cbrt(x);
float fourth_root = (float)Math.pow(x, 1. / 4);
```

Явно импортировать его, как `Scanner`, не нужно; `Math` доступен всегда

Многие функции из Math работают только с double, их результат придется явно приводить

Но есть и функции с версиями для:
`int`, `long`, `float` и `double`

Полный список можно посмотреть в документации Java:
https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html

Бонус: инкремент

В Java есть оператор “инкремент”, увеличивающий значение в переменной на 1:

```
int i = 3;
++i;
System.out.println(i);
```

```
4
```

Инкрементов два вида: префиксный и постфиксный

Отличие в том, что префиксный возвращает, что стало после; а постфиксный, что было до его вызова

```
int i = 3;
int j = 3;

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

System.out.println(i);
System.out.println(j);
```

```
4
3
4
4
```

Теперь мы можем посмотреть приоритет всех операторов в Java!
https://docs.oracle.com/javase/tutorial/java/nutsandbolts/operators.html