HIGHER SCHOOL OF ECONOMICS
NATIONAL RESEARCH UNIVERSITY

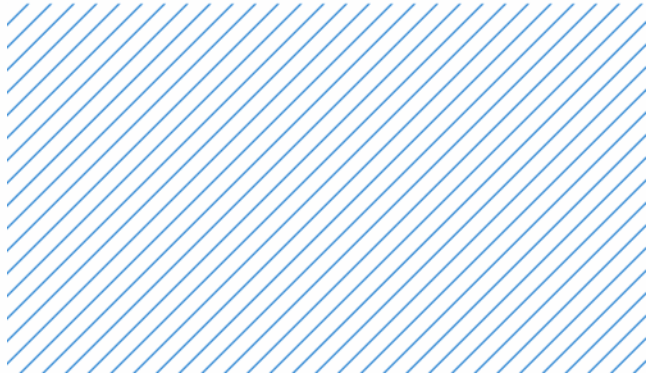
Лекция 5

Основы процедурного программирования на Java (часть 3)

Программирование на языке Java

Роман Гуров

ВШЭ БИ 2021



Большие числа

Диапазон значений, которые мы можем хранить в целочисленной переменной, довольно ограничен:

Тип	Размер (в байтах)	Мин	Макс
`long`	8	$-9223372036854775808 \approx -9.22 \cdot 10^{18}$	$9223372036854775807 \approx 9.22 \cdot 10^{18}$

Но как быть, если нужны очень большие целые числа, и без всяких погрешностей?

Как вообще хранить такое большое число?

Вспомним про массивы, которые позволяли хранить в себе целую последовательность значений и не имели таких явных ограничений на размер

Можно ведь просто хранить каждую цифру огромного числа в отдельной ячейке массива, а арифметические операции реализовать руками, аналогично счёту в столбик!

	1	2	3	4	5	6	7	8	9	1	2	3	4	5	6	7	8	9	1	2	3	4	5	6	7	8	
	8	7	6	5	4	3	2	1	9	8	7	6	5	4	3	2	1	9	8	7	6	5	4	3	2	2	+
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	=

BigInteger

Но рано пугаться, разработчики Java уже сделали всё за нас

Для длинной арифметики есть специальный тип – `BigInteger`, реализованный похожим образом

```java
BigInteger a = BigInteger.valueOf(100);

for (int i = 0; i < 6; ++i) {
    a = a.multiply(a);
}

System.out.println(a);
```

⇩

```
100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

BigInteger

Привычные нам операторы (+, -, *, / и др.) работают только с примитивными типами
Java вообще не позволяет переопределить операторы для каких-то новых типов

Поэтому для всех операций нужно использовать встроенные функции объектов `BigInteger`

Вот некоторые из них:

- `BigInteger add(BigInteger other)`
- `BigInteger subtract(BigInteger other)`
- `BigInteger multiply(BigInteger other)`
- `BigInteger divide(BigInteger other)`
- `BigInteger mod(BigInteger other)`
- `BigInteger sqrt()`
- `int compareTo(BigInteger other)`
- `int intValue()`
- `long longValue()`

Полный список тут:
https://docs.oracle.com/javase/7/docs/api/java/math/BigInteger.html

BigInteger

Функция `compareTo` используется для сравнения двух больших чисел и устроена по смыслу так:

```
a.compareTo(b);
```

=

```
if (a < b) {
    return -1;
} else if (a == b) {
    return 0;
} else {
    return 1;
}
```

```
BigInteger a = BigInteger.valueOf(2);
BigInteger b = BigInteger.valueOf(123);

while (a.compareTo(b) < 0) {
    a = a.multiply(BigInteger.valueOf(2));
}

System.out.println(a);
```

`BigInteger` неизменяем, для него нет аналогов составных операторов, таких как += у простых типов

Любые операции просто создают для результата новый объект,
который мы записываем обратно в переменную

BigInteger

Создание `BigInteger` из числа производится при помощи функции `BigInteger.valueOf()`

Внутри себя, данная функция создаст новый объект с помощью `new`, и вернёт его нам

Также, его можно создать из текстовой строки, в которой написано число
Для этого нам придётся уже самим использовать `new`

```
BigInteger a = BigInteger.valueOf(-12345678912345678911L);
BigInteger b = new BigInteger("-123456789123456789123456789123456789123456789");
```

Через текстовую строку можно передать значения произвольного размера,
а через число только то, что в это число влезает

BigDecimal

А что там с дробными числами? – спросите вы

А для них тоже есть специальный тип – `BigDecimal`,
который хранит дробное число как `BigInteger` и позицию точки в числе

Методы аналогичны BigInteger, но с делением есть нюанс

```
BigDecimal a = BigDecimal.valueOf(10.25);
BigDecimal b = new BigDecimal("0.8");

System.out.println(a.divide(b));
```

```
12.8125
```

Так сработало, но, например, число 1/3 представляется только периодической дробью
и подобное деление уже вызовет ошибку

BigDecimal

Для деления придётся явно указывать количество знаков после запятой, которое нам необходимо

Ввиду того, что ограничение числа знаков может привести к потере информации,
нужно также указать способ округления числа

```
BigDecimal a = BigDecimal.valueOf(1).setScale(10, RoundingMode.HALF_UP);
BigDecimal b = new BigDecimal("3");

System.out.println(a.divide(b, RoundingMode.HALF_UP));
```

```
0.3333333333
```

Все методы класса BigDecimal можно увидеть тут:
https://docs.oracle.com/javase/7/docs/api/java/math/BigDecimal.html

Все виды округления перечислены тут:
https://docs.oracle.com/javase/8/docs/api/java/math/RoundingMode.html

Массивы

Чтобы создать переменную для массива, нужно приписать к имени типа пустые квадратные скобки:

```
int[] array;
```

Массивный тип не является примитивным!
Для создания самого объекта массива нужно использовать `new`

```
int[] array = new int[100];
```

Дальнейшее использование нам уже знакомо:

```
array[0] = 5;
array[99] = 9;

System.out.println(array[0] + array[50] + array[99]);
```

При создании, все элементы массива инициализируются нулями

Массивы

Можно явно перечислить начальные значения элементов массива:

```
int[] array = {1, 1, 2, 3, 5, 8, 13, 21};
System.out.println(array[6]);
```

```
13
```

Также, можно узнать размер массива, что очень пригодится в циклах:

```
int[] array = {1, 1, 2, 3, 5, 8, 13, 21};
System.out.println(array.length);
```

```
8
```

Многомерные массивы

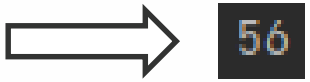
Помимо обычного одномерного массива, существуют и многомерные массивы

Одномерный массив задаёт последовательность элементов,
Двумерный массив – таблицу элементов,
И так далее

```
int[][] multiplication_table = new int[11][11];

for (int i = 0; i < multiplication_table.length; ++i) {
    for (int j = 0; j < multiplication_table[i].length; ++j) {
        multiplication_table[i][j] = i * j;
    }
}

System.out.println(multiplication_table[7][8]);
```



На самом деле, многомерный массив – это массив массивов

Особенность объектных типов

Как мы видели, объект массива изменяем

Наконец-то у нас есть хороший пример, чтобы продемонстрировать факт того, что переменная объектного типа не хранит в себе сам объект

```
int a = 5;
int b = a;

System.out.println(a);
System.out.println(b);

b = 10;

System.out.println(a);
System.out.println(b);
```

```
5
5
```

```
5
10
```

```
int[] a = {5};
int[] b = a;

System.out.println(a[0]);
System.out.println(b[0]);

b[0] = 10;

System.out.println(a[0]);
System.out.println(b[0]);
```

```
5
5
```

```
10
10
```

Особенность объектных типов

Поскольку объектные переменные не хранят сам объект, у них бывает пустое состояние

Такое состояние задается через специальное значение `null`

```
int[] a = null;
int[] b = {10};

System.out.println(a == null);
System.out.println(b == null);
```

```
true
false
```

И работает с любыми объектными типами:

```
float[] array = null;
BigInteger bigint = null;
Scanner = null;
```

Но не с примитивными:

```
int primitive = null;
```

Ключевое слово final

В Java существуют возможность объявить переменную, значение которой запрещено изменять

Для этого при объявлении используется слово final

```
final int a = 5;

a += 10;
a = 5;
```

Данное слово исключительно ограничивает наши возможности,
но ограничения бывают полезны, чтобы избежать случайных ошибок
и сделать свой код более понятным при прочтении

Важно отметить, что для объектных типов сам объект остаётся изменяемым,
запрещается только менять ссылку, которую хранит переменная:

```
final double[] a = {3.0, 3.14};

a[0] += 10;
a[1] = 5;

a = new double[10];
```

Строки

С самого начала изучения Java, мы говорим о текстовых строках,
но до сих пор ничего о них не знаем, даже какой у них тип

Ожидаемо, строки имеют тип String:

```
String abc = "abc";
```

Строки не ограничены только хранением, у них есть много интересных фишек

```
System.out.println("abc");
String cde = "cde";
System.out.println("abc" + cde);
String c = "abc".substring(2,3);
String d = cde.substring(1, 2);
```

```
double iamfloat = 2.78;
String result = "The number is " + 5 + "\nAnd the other one is " + iamfloat;
System.out.println(result);
```

```
The number is 5
And the other one is 2.78
```

Строки

Строки – неизменяемые объекты (также говорят иммутабельные/immutable)

У них метод charAt(), позволяющий *прочитать* символ по данному индексу, как в массиве

```
String test = "Testing charAt()";
System.out.println(test.charAt(12));
```

```
A
```

Документация строки тут:
https://docs.oracle.com/javase/7/docs/api/java/lang/String.html

StringBuilder

Но если хочется собрать строку по кусочкам, как изменяемый объект, то можно воспользоваться `StringBuilder` – по сути, изменяемой строкой

```
StringBuilder builder = new StringBuilder("Testing set+harAt()");
builder.setCharAt(11, 'C');
System.out.println(builder);
```

```
Testing setCharAt()
```

Самый главный метод билдера – append()
Он добавляет текстовое представление объекта в конец строки

```
int number = -1;
String sign = number >= 0 ? " non-negative" : " negative";

StringBuilder builder = new StringBuilder("The");
if (Math.abs(number) == 1) {
    builder.append(" greatest");
}
builder.append(sign);
builder.append(" number is: ");
builder.append(number);

String final_string = builder.toString();
System.out.println(final_string);
```

```
The greatest negative number is: -1
```

StringBuilder

```java
StringBuilder builder = new StringBuilder();
builder.append("word");
builder.insert(0, 's');
System.out.println(builder);

builder.deleteCharAt(1);
System.out.println(builder);

builder.insert(4, "er");
builder.insert(0, "di");
System.out.println(builder);

builder.delete(0, 3);
System.out.println(builder);

builder.replace(1, 3, "bserv");
System.out.println(builder);

builder.reverse();
System.out.println(builder);
```

```
sword
sord
disorder
order
observer
revresbo
```

Документация: https://docs.oracle.com/javase/7/docs/api/java/lang/StringBuilder.html

Задание методов в Main

До сих пор мы не умели создавать функции

Пора научиться, тут всё просто:

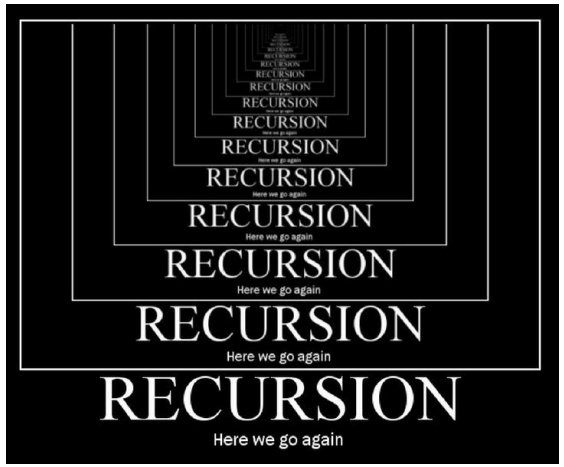
```
public class Main {
    public static void main(String[] args) {
        System.out.println(cube(10));
    }

    static int cube(int x) {
        return x * x * x;
    }
}
```

Научились.

Рекурсия

Рекурсия – вызов функции из неё же самой, непосредственно или через другие функции



Напишем рекурсивную функцию, выводящую на экран числа от N до 1

```java
public static void main(String[] args) {
    printFromNTo1(6);
}

static void printFromNTo1(int n) {
    if (n < 1)
        return;
    System.out.println(n);
    printFromNTo1(n - 1);
}
```

```
6
5
4
3
2
1
```

Рекурсия

Напишем рекурсивную функцию, выводящую на экран числа от N до 1

```
public static void main(String[] args) {
    printFromNTo1(6);
}

static void printFromNTo1(int n) {
    if (n < 1)
        return;
    System.out.println(n);
    printFromNTo1(n - 1);
}
```

```
6
5
4
3
2
1
```

А как вывести от 1 до N?

```
public static void main(String[] args) {
    printFrom1ToN(6);
}

static void printFrom1ToN(int n) {
    if (n < 1)
        return;
    printFrom1ToN(n - 1);
    System.out.println(n);
}
```

```
1
2
3
4
5
6
```

Стек вызовов

Пусть, мы исполняем код:

```java
public static void main(String[] args) {
    printFrom1ToN(6);
}

static void printFrom1ToN(int n) {
    if (n < 1)
        return;
    printFrom1ToN(n - 1);
    System.out.println(n);
}
```

И в данный момент исполняется строчка `System.out.println(n)`, и `n == 1`

Следовательно в данный момент стек вызовов выглядит так:

```
main()
printFrom1ToN(6)
printFrom1ToN(5)
printFrom1ToN(4)
printFrom1ToN(3)
printFrom1ToN(2)
printFrom1ToN(1)
System.out.println(1)
```

Переполнение стека вызовов

Следовательно в данный момент стек вызовов выглядит так:

```
main()
printFrom1ToN(6)
printFrom1ToN(5)
printFrom1ToN(4)
printFrom1ToN(3)
printFrom1ToN(2)
printFrom1ToN(1)
System.out.println(1)
```

Получается, что при вызове функции надо полностью запоминать всё состояние локальных переменных и место, куда надо вернуться после вызова

То есть, при большой глубине стека вызовов, каждый вызов висит в памяти, занимая её

Если размер стека становится слишком большим, программа экстренно завершается с ошибкой
Такая ситуация называется **stack overflow** или переполнение стека

Поэтому, рекурсию нужно использовать аккуратно и только при её полезности