Java

Author: Daria Shutina

23-02-02

Типизация

Статическая и динамическая типизация

Статически типизированные языки чётко определяют типы переменных. Типы проверяются во время компиляции. Код не скомпилируется, если типы не совпадают.

Каждое выражение в статически типизированном языке относится к определенному типу, который можно определить без запуска кода. Иногда компилятор может сам вывести тип, если он не указан явно. Например, в хаскелле функция `add x y = x + y` принимает числа (и возвращает число), потому что `+` работает только на числах.

Динамически типизированные языки не требуют указывать тип, но и не определяют его сами. Например, в питоне функция `def f(x, y): return x + y` может принимать и числа, и строки. Переменные `x` и `y` имеют разные типы в разные промежутки времени.

Говорят, что в динамических языках значения обладают типом (`1` ⇒ `Integer`), а переменные и функции — нет (`x` и `y` могут быть чем угодно в примере выше).

Сильная и слабая типизация

Типизация сильная, если нет неявных преобразований типов.

Типизация слабая, если возможны неявные преобразования типов.

Граница между "сильным" и "слабым" размыта. Мяу.

О языке

- Кроссплатформенный, преимущественно объектно-ориентированный ЯП.
- Обладает совместимостью: старый код будет компилироваться на джаве более новой версии либо без изменений, либо с минимальными изменениями.
- Есть строгая спецификация языка и JVM. У каждой версии своя спецификация.
- Статическая типизация
- Автоматическое выделение памяти (thanks tgarbage collection)

Общие слова

`jshell` (джей эс хелл) -- интерактивная штука для мгновенного запуска кода; является REPL (read-evaluate-print-loop).

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

`System.out.println("aboba");` -- это statement.

`System.out.println("aboba")` -- это expression и вызов метода.

`System.out` является *квалификатором* для метода `println` (как и `System` для `out` в `System.out`).

Переменные

Название	Класс-обёртка	MIN_VALUE	MAX_VALUE
byte	Byte	$-2^7 = -128$	$2^7-1 = 127$
short	Short	$-2^{15} = -32\,768$	$2^{15}-1 = 32\,767$
char	Character	0	$2^{16}-1 = 65\,535$
int	Integer	$-2^{31} = -2\,147\,483\,648$	$2^{31}-1 = 2\,147\,483\,647$
long	Long	$-2^{63} \approx -9 \cdot 10^{18}$	$2^{63}-1 \approx 9 \cdot 10^{18}$

При объявлении переменной её тип не обязательно писать. Для этого используется `var`:

```
1 int x;
2 x = 5;
3 var y = 5;
```

Тип выявляется неявно, но нужно обязательно проинициализировать переменную.

При передаче переменной в метод значение переменной копируется. Если это ссылочный объект, то внутри него хранится ссылка, следовательно, ссылка и копируется для метода.

Целочисленные литералы

число	префикс
int	-
long	L
двоичное	0b
восьмиричное	0 (это ноль)

число	префикс
шестнадцатиричное	0x

Чтобы визуально разделить число на фрагменты, можно использовать нижнее подчеркивание: `1_000_000`.

Вещественные литералы

число	суффикс
double	D (1D)
float	F (1F)
с экспонентой	1.6e-19 = $1,6 \cdot 10^{-19}$

equals()

В случае объектов, в перменной хранится ссылка на него. Получается, что оператор `==` будет сравнивать адреса объектов, а не сами объекты.

Можно использовать метод `equals()`, живущий в классе `Object`. По умолчанию, он работает, как и оператор `==`, но его можно переопределить:

```
1  public class MyClass {
2      int id;
3
4      public boolean equals(MyClass otherClass) {
5          return this.id ==  otherClass.id;
6      }
7
8      public static void main(String[] args) {
9          var o1 = new MyClass();
10         o1.id = 1;
11         var o2 = new MyClass();
12         o2.id = 1;
13         System.out.println(o1.equals(o2));
14     }
15 }
```

```
16
```

Дети, всегда используйте метод `equals()`!!

String pool

String pool -- специальная область для хранения строк, созданная ради экономии памяти. В него помещается нужная строка (если её там не было ранее), и в дальнейшем новые переменные ссылаются на одну и ту же область памяти.

Если создавать переменную как `new String("aboba")`, то оператор `new` принудительно создает для строки новую область памяти, не добавляя ее в пулл строк.

У строк есть метод `intern()`. Он проверяет, есть ли строка в пулле; если нет, создаёт её; потом возвращает адрес строки.

```
public class Main {
   public static void main(String[] args) {
       var s1 = "aboba";
       var s2 = new String("aboba");
       System.out.println(s1 == s2.intern());
   }
}
```

Пример

```
s1 = "aboba";
s2 = new String("aboba");
```

Выражение `s1 == s2` вернет `false`, потому что у объектов разные адреса.

Но `s1.equals(s2)` вернет `true`, потому что у класса `String` переопределен метод `equals`.

Если при сравнении не важен регистр, можно использовать метод `equalsIgnoreCase()`.

Unicode в джаве

По умолчанию, в джаву добавили только Unicode. От остальных кодировок отказались, чтобы не было путаницы.

Некоторые термины Unicode

сурс: http://www.unicode.org/glossary/

- ✓Code point – номер символа (0-0x10FFFF = 1114111)
- ✓Basic Multilingual Plane (BMP) – символы 0-65535 (‘\uFFFF’), «плоскость 0»
- ✓Supplementary Plane – плоскости 1-16 (символы 0x10000-0x10FFFF)
- ✓Surrogate code point – 0xD800-0xDFFF
- ✓High surrogate code point – 0xD800-0xDBFF
- ✓Low surrogate code point – 0xDC00-0xDFFF
- ✓Code unit – минимальная последовательность бит для представления кодовых точек в заданной кодировке
  - ✓UTF-8: 1 байт, 0..255
  - ✓UTF-16: 2 байта, 0..65535 (Java!)
  - ✓UTF-32: 4 байта

Символы за пределами плоскости 0 называются surrogate pair. Состоят из двух code-юнитов: первый -- high surrogate, второй -- low surrogate.

Специальные символы в джаве состоят из двух байтов. `char` может содержать только один. Поэтому если хочется использовать какие-то специальные символы, то писать их нужно в `String`, а не в `char`:

```
public static void main(String[] args) {
    char c = '😀';
    System.out.println(c);  // ==> ?
    String s = "😀";
    System.out.println(s);  // ==> 😀
}
```

NaN

= Not-A-Number. Используется, чтобы представить математически неопределенное число (i.e. деление на ноль, $\sqrt{-1}$).

Значение NaN нельзя ни с чем сравнить. Выражение `Float.NaN == Float.NaN` вернет `false`, а `Float.NaN != Float.NaN` вернет `true`. Чтобы проверить, является ли значение NaN, используется метод `Float.isNaN()`.

Константы времени компиляции

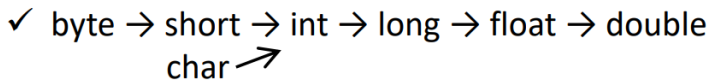
- Литералы (числа, строки)
- Инициализированные final-переменные примитивных типов и типа String, если инициализатор – константа (`final int i = 1;`)
- Операции над константами
- Конкатенация строк
- Скобки
- Условный оператор, если все операнды – константы.

Type cast (приведение типов)

```
1 int x = 128;
2 byte c = (byte) x;
```

Преобразование типов

Расширяющее

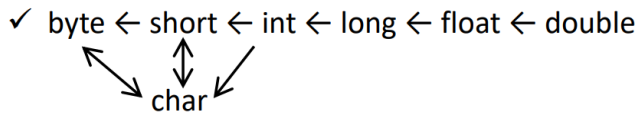



```
short s = b;
int i = s;
long l = i;
float f = l;
double d = f;
```

Потеря точности:
✓ int → float
✓ long → float
✓ long → double

Откуда потеря точности? У `double` есть ограничение на целую часть: она не может быть больше 10^{15}, потому что чисел после запятой может быть и бесконечное количество. А `long` может быть больше 10^{15}.

Сжимающее



Приведение типа (type cast)

```
float f = (float) d;
long l = (long) f;
int i = (int) l;
short s = (short) i;
char c = (char) s;
byte b = (byte) c;
```

Потеря точности/переполнение: всегда

Детали

Расширяющие и сжимающие преобразования требуют явного каста. Но можно написать и так:

```
1 final int i = 2;
2 byte c = i;
```

Это константа времени компиляции. Компилятор неявно преобразует число в нужный тип.

При этом если значение переменной выйдет за рамки значений типа, к которому кастуем, код не скомпилируется:

```
1 final int i = 128;
2 byte c = i;
3 // error: incompatible types: possible lossy conversion from int to byte
```

Преобразования типов в выражениях

- Унарные операции, битовый сдвиг, индекс массива: `byte`, `short`, `char` $\longrightarrow$ `int`, остальные не меняются
- Бинарные операции -- есть определённая иерархия:
    1. Любой операнд `double` $\Rightarrow$ другой операнд `double`

2. Любой операнд `float` ⇒ другой операнд `float`
3. Любой операнд `long` ⇒ другой операнд `long`
4. Иначе оба операнда `int`

Данный код не скомпилируется:

```
byte a1 = 1;
byte a2 = 2;
byte sum = a1 + a2;
// error: incompatible types: possible lossy conversion from int to byte
```

Пример

Дети, избегайте разных типов в одном выражении!!

(используйте промежуточные переменные)

```
double a = Long.MAX_VALUE;
long b = Long.MAX_VALUE;
int c = 1;

System.out.println(a+b+c); // 1.8446744073709552E19
// a + b -> double (переполнения нет)
//   + c -> double

System.out.println(c+b+a); // 0.0
// c + b -> long (переполнение)
//   + a -> double
```

Замечание

Если операции написаны сокращенно (`+=` вместо `var + var`), коспилятор добавляет каст неявно:

```
byte b = 1;
b = b + 1; // не норм
b += 1;    // норм

char c = 'a';
c *= 1.2; // законно, но втф
```

Массивы

Массив -- это объект, живущий в на куче. Внутри массива хранятся *ссылки* на другие объекты.

Длина массива -- всегда константа. Она может быть не известна на этапе компиляции, но после выделения памяти длину массива нельзя поменять.

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

int[] oneD_init = new int[]{1, 2, 3, 4, 5};
// длина массива станет известна в момент выполнения этой строки

int[] oneDsimple = {1, 2, 3, 4, 5};

int[][] twoD = new int[2][5];

int[][] twoDsimple = { {1, 2, 3}, {4, 5}, {6} };
// массив ссылок на другие массивы
// длина компонент может быть разная
```

В памяти массив хранится в таком виде:

```
┌──────────────┬──────────────┬──────┬──────┬──────┐
│    type      │   length     │ [0]  │ [1]  │ [2]  │
└──────────────┴──────────────┴──────┴──────┴──────┘
```

Размеры двумерных массивов могут по-разному влиять на память.

`new int[2][500]` займет 4056 байт: $2 + 2 \cdot 2 + \textit{память для ячеек}$.

`new int[500][2]` займет 14016 байт: $2 + 500 \cdot 2 + \textit{память для ячеек}$. Тут нужно больше памяти для хранения информации `type+length`.

Операции с массивом

Методы класса `Array`: java.util.Arrays

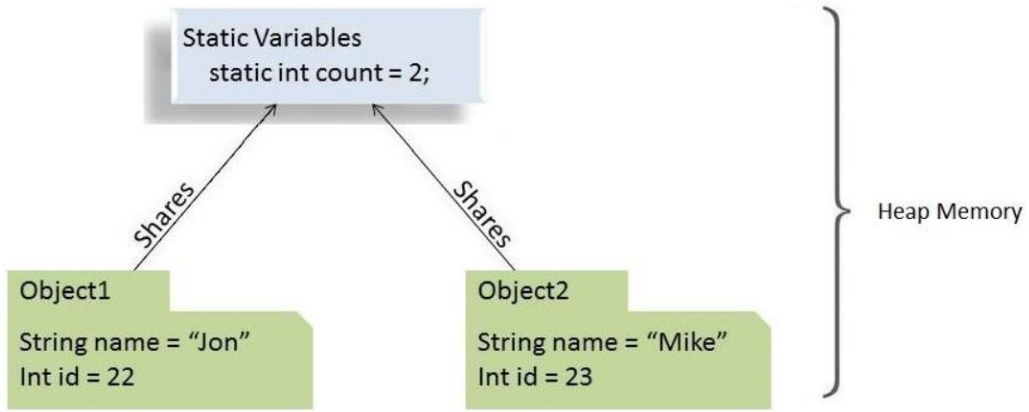
```
int[][] ints = { {1, 2, 3}, {4, 5}, {6} };

System.out.println(ints.length);

```

```java
int[][] copy = ints.clone();
// скопировать значения массива. Для двумерных массивов клонируется только верхний слой, в компонентах будут храниться ссылки на те же объекты

int[][] copyRange = Arrays.copyOfRange(ints, 1, 3);

System.out.println(ints == copy); // false
// `==` сравнивает адреса объектов

System.out.println(ints.equals(copy)); // false
// по умолчанию, `equals` сравнивает по адресам

System.out.println(Arrays.equals(ints, copy)); // true

System.out.println(ints); // [I@3cb5cdba
// абра кадабра

System.out.println(Arrays.toString(ints)); // [1, 3, 3, 2, 1]
// обычный вывод массива

Arrays.sort(ints);
```

23-02-07

Модификатор `static`

`static` у метода означает, что метод относится к самому типу, а не к инстансу этого типа. То есть статический метод будет общим у всех инстансов.



В статическом методе не передается ссылка на текущий объект `this` (он же общий), поэтому внутри него нет доступа к нестатическим объектам или нестатическим полям.

Обычно рекомендуется делать методы статическими, когда они никак не используют состояние объекта.

В джаве без использования класса нельзя даже использовать функцию `main`. Иногда бывает удобным вызывать методы без какого-то конкретного объекта (например, для `main`). Для этого методы классов помечаются `static`. Таким образом мы можем использовать метод, не создавая инстанс класса (вспомним функцию `main`).