

Java

Author: Daria Shutina

Java

23-02-02

Типизация

Статическая и динамическая типизация

Сильная и слабая типизация

О языке

Общие слова

Переменные

Целочисленные литералы

Вещественные литералы

equals()

String pool

Пример

Unicode в джаве

NaN

Константы времени компиляции

Type cast (приведение типов)

Преобразование типов

Расширяющее

Сжимающее

Детали

Преобразования типов в выражениях

Пример

Замечание

Массивы

Операции с массивом

23-02-16

Приоритет операций

Значения по умолчанию в функциях

Модификатор `static`

Object

Методы Object

Классы

Создание

Record

Статические переменные и методы

Self-referential class

`instanceof`

Интерфейсы

Использование `interface`

Абстрактные классы

Модификатор `final`

Модификатор `sealed`

Фабричный метод

Enum

Перечисление

switch, используя экземпляры enum-класса

Dependency Injection (DI)

Пакеты

Модули

23-02-23

Исключения

Иерархия

Конструкторы

Полезные методы у исключений

try-catch

Еще про `finally`

Логирование: пример использования

Надтипы и подтипы

Пример использования `extends`

Пример использования `super`

Covariant return type

Параметризация

Параметризация типов

Маскировочный тип

`? extends Number`

Наследование

Параметризация методов

Дженерики и массивы

Стирание (erasure)

Пример

23-03-02

`varargs`

varargs + generics

Collections

Иерархия

Одновременный read & write

Не надо использовать

23-03-16

Функциональный интерфейс

Функциональное выражение

Примеры

Лямбда-выражения

Возвращаемое значение

Захват значений

Пример

Optional<T>

Пример 1: update karma

Пример 2: update + throw

Stream API

Источники

Промежуточные операции

Пример `peek`

Пример `flatMap`

Пример, как не надо делать: shuffle

Пример, как не надо делать: порядок промежуточных операций

Терминальные операции

Пример `anyMatch`

Пример `reduce`

Collectors

Примеры

Пример: группируем строки по длине

Пример: вложенные коллекторы

Пример `toMap`

Как устроен коллектор

Создание коллектора

Пример с `toList`

Пример: коллектор, считающий и макс, и мин

23-04-18

InputStream

Реализации

Методы

Пример

Современный InputStream

Управление внешними ресурсами: раскрытие файла

OutputStream

Реализации

Методы

Пример

Reader

Реализации

Методы

Writer

Реализации

Методы

`java.io.File` (классика)

Методы

Пример

`java.nio.file.[Path/Paths/Files]`

Методы

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).

```
1 public class Main {  
2     public static void main(String[] args) {  
3         System.out.println("aboba");  
4     }  
5 }
```

`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.

Если попытаться вывести непроинициализированную переменную, код не скомпилируется:

```
1 int x;
2 System.out.println(x); // does not compile
```

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)
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 переменной 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.

```
1 public class Main {
2     public static void main(String[] args) {
3         var s1 = "aboba";
4         var s2 = new String("aboba");
5         System.out.println(s1 == s2.intern());
6     }
7 }
```

Пример

```
1 s1 = "aboba";
2 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`:

```
1 public static void main(String[] args) {
2     char c = '😊';
3     System.out.println(c); // ==> ?
4     String s = "😊";
5     System.out.println(s); // ==> 😊
6 }
```

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

✓ byte → short → int → long → float → double
char →

```
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

✓ byte ← short ← int ← long ← float ← double
char ←

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` → `int`, остальные не меняются
- Бинарные операции -- есть определённая иерархия:
 1. Любой операнд `double` ⇒ 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:

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

Пример

Дети, избегайте разных типов в одном выражении!!

(используйте промежуточные переменные)

```

1  double a = Long.MAX_VALUE;
2  long b = Long.MAX_VALUE;
3  int c = 1;
4
5  System.out.println(a+b+c); // 1.8446744073709552E19
6  // a + b -> double (переполнения нет)
7  //   + c -> double
8
9  System.out.println(c+b+a); // 0.0
10 // c + b -> long (переполнение)
11 //   + a -> double

```

Замечание

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

```

1  byte b = 1;
2  b = b + 1; // не норм
3  b += 1;    // норм
4
5  char c = 'a';
6  c *= 1.2; // законно, но втф

```

Массивы

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

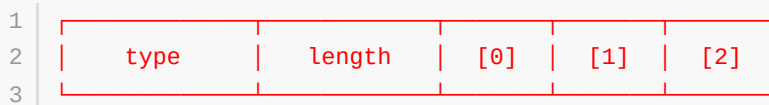
Длина массива -- всегда константа. Она может быть не известна на этапе компиляции, но после выделения памяти длину массива нельзя поменять.

```

1  int[] oneD = new int[10];
2
3  int[] oneD_init = new int[]{1, 2, 3, 4, 5};
4  // длина массива станет известна в момент выполнения этой строки
5
6  int[] oneDsimple = {1, 2, 3, 4, 5};
7
8  int[][] twoD = new int[2][5];
9
10 int[][] twoDsimple = { {1, 2, 3}, {4, 5}, {6} };
11 // массив ссылок на другие массивы
12 // длина компонент может быть разная

```

В памяти массив хранится в таком виде:



Размеры двумерных массивов могут по-разному влиять на память.

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

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

Операции с массивом

Методы класса `Array`: [java.util.Arrays](https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html)

```

1  int[][] tmp = new int[3][];
2
3  Arrays.fill(tmp, 0, tmp.length, new {1});
4  // заполнить массив начальными значениями
5
6  int[][] ints = { {1, 2, 3}, {4, 5}, {6} };
7
8  System.out.println(ints.length);
9
10 int[][] copy = ints.clone();
11 // скопировать значения массива. Для двумерных массивов копируется только
   // верхний слой, в компонентах будут храниться ссылки на те же объекты
12
13 int[][] copyRange = Arrays.copyOfRange(ints, 1, 3);
14
15 System.out.println(ints == copy); // false
16 // `==` сравнивает адреса объектов
17
18 System.out.println(ints.equals(copy)); // false
19 // по умолчанию, `equals` сравнивает по адресам

```

```

20
21 System.out.println(Arrays.equals(ints, copy)); // true
22
23 System.out.println(ints); // [I@3cb5cdba
24 // абра кадабра
25
26 System.out.println(Arrays.toString(ints)); // [1, 3, 3, 2, 1]
27 // обычный вывод массива
28
29 Arrays.sort(ints);

```

23-02-16

Приоритет операций

Название	Символ
access	[] .
postfix	x++ x--
unary	++x --x +x -x ~ !
cast/new	(type) new
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	
logical AND	&&
logical OR	
ternary	?:
assignment	= += -= *= /= %= &= ^= = <<= >>= >>>=

Пример тернарного оператора (ternary):

```

1 System.out.println(x > 0 ? "positive" : "negative or zero");

```

Значения по умолчанию в функциях

В джаве нет аргументов по умолчанию, но можно сделать так, с помощью перегрузки метода:

```

1 public static void main() {
2     class Point {
3         int x, y;

```

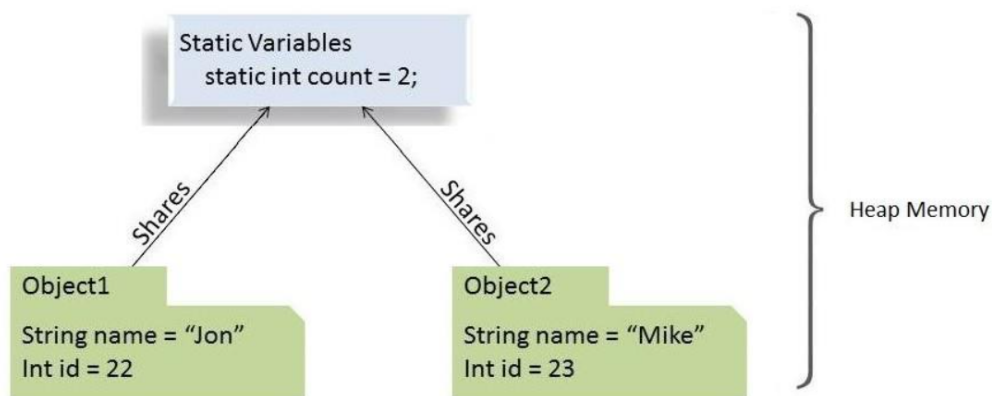
```

4
5     void shift(int dx, int dy) {
6         x += dx;
7         y += dy;
8     }
9
10    void shift(int dx) {
11        shift(dx, 0);
12    }
13 }
14
15 Point p = new Point();
16 p.x = 1;
17 p.y = 1;
18 p.shift(1, 1);
19 p.shift(1);
20 }

```

Модификатор `static`

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



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`).

Object

Объект -- область фиксированного размера в куче. Обладает фиксированной структурой, содержит заголовок и набор значений. Порядок данных внутри объекта не определен строго.

Объект любого типа наследуется от типа `java.lang.Object`.

Значение-ссылка может ссылаться на объект или на `null`, но не на конкретное поле объекта. Например, нельзя создать ссылку на элемент массива, потому что не понятно, в какой последовательности машина хранит данные в памяти.

Методы Object

Методы, которые есть предопределенные у каждом объекте:

- `toString()` -- можно переопределить. По умолчанию, для классов выводит хэш-код(?), для record классов выводит `<className>[field1=value, field2=value]`
- `equals()` -- можно переопределить
- `hashCode()` -- можно переопределить
- `getClass()` -- final-метод, нельзя переопределить
- `wait()`, `notify()`, `notifyAll` -- для многопоточки. final-методы, нельзя переопределить
- `clone()` -- можно переопределить

Классы

Создание

```

1  class Point {
2      int x = 0; // значения по умолчанию
3      int y = 0;
4
5      Point(int x, int y) {
6          this.x = x; // `this.` используется, чтобы как-то различать
7          this.y = y; // название поля и название аргумента
8      }
9
10     // деструкторов в джаве нет
11 }
```

```

1 public static void main(String[] args) {
2     Point p1 = new Point(1, 2);
3     Point p2 = new Point(3, 4);
4
5     // Создали два объекта на куче. В переменных `p1` и `p2`
6     // хранятся ссылки на эти объекты.
7
8     p1 = p2;
9     // Поменяли ссылку, сохраненную в `p1`.
10    // Теперь нет прямого доступа к первому объекту,
11    // и сборщик мусора уничтожил первый объект.
12 }

```

Record

Records это неизменяемые дата классы, требующие только тип и имя полей. Используются как замена темплейтам, помогают избежать дублирование кода.

```

1 public record Person (String name, String address) {}

```

У records по умолчанию создаются и определяются:

- публичный конструктор
- публичные геттеры, сеттеры
- метод `equals()`, который возвращает `true`, если все соответствующие поля будут одинаковые.
- метод `hashCode()`. Хэш коды будут одинаковые, если все соответствующие поля будут одинаковые.
- метод `toString()`

Можно изменять или добавлять реализацию конструктора:

```

1 public record Person(String name, String address) {
2     public Person { // меняет дефолтный конструктор
3         Objects.requireNonNull(name);
4         Objects.requireNonNull(address);
5     }
6
7     public Person(String name) {
8         this(name, "Unknown");
9     }
10 }

```


Статические переменные и методы

```

1 public record Person(String name, String address) {
2     public static String UNKNOWN_ADDRESS = "Unknown";
3
4     public static Person unnamed(String address) {
5         return new Person("Unnamed", address);
6     }
7 }

```

Обратиться к ним можно, используя имя record класса:

```

1 Person.UNKNOWN_ADDRESS;
2 Person.unnamed("aboba address");

```

Self-referential class

Класс типа `Type`, у которого есть поле такого же типа `Type`:

```

1 class Aboba {
2     private Aboba abobaField;
3     private String data;
4
5     public Aboba() {}
6 }

```

При неправильном написании конструктора можно закончить в бесконечном цикле и получить ошибку `StackOverflowError` в рантайме.

Такая штука используется, например, при реализации Singleton pattern.

instanceof

Оператор, чтобы проверить, является ли объект экземпляром определенного класса:

```

1 record Point(int x, int y) {
2     void shift(int dx, int dy) {
3         x += dx;
4         y += dy;
5     }
6 }
7
8 public static void process(Object obj) {
9     if (obj instanceof Point) {
10        Point point = (Point) obj;
11        point.shift(1, 1);
12    }
13 }

```

Концепция "instanceof + каст" популярна, поэтому в язык добавили фичу:

```

1 if (obj instanceof Point point) {
2     point.shift(1, 1);
3 }

```

Объект `point` создаётся только в том случае, если условие вычисляется в `true`.

Интерфейсы

Интерфейс описывает поведение объектов, его реализующих, и способы взаимодействия с такими объектами. Важное отличие интерфейса от класса в том, что интерфейс не определяет внутреннее состояние объекта. Нельзя создать экземпляр интерфейса и в интерфейсе нет конструкторов.

У интерфейса есть методы, абстрактные по умолчанию (нет тела функции). Интерфейс также может содержать константы, обычные и статические методы, вложенные типы. Тела методов существуют только для обычных и статических методов.

Поля в интерфейсе могут быть, но они обязаны быть `static` и `final`.

Интерфейс определяет двухсторонний контракт -- требования к реализации и к использованию. Требования выражаются:

- в сигнатурах методов. Проверяется компилятором.
- в аннотациях. Проверяются процессорами аннотаций, статическим анализом и т.д. Пример -- Javadoc, комментарии специального формата.

Для функции можно расписать, какой аргумент что означает, какой смысл у возвращаемого значения:

```

1  /**
2   * Returns a sum of double numbers
3   * @param a first number
4   * @param b second number
5   * @return a result
6   */
7  double sum(double a, double b);

```

- в документации. Простое описание функций и методов.

Использование `interface`

```

1  public interface Swimmable {
2      public void swim();
3  }
4
5  public class Duck implements Swimmable {
6      @Override
7      public void swim() {
8          System.out.println("Уточка, плыви!");
9      }
10
11     public static void main(String[] args) {
12         Duck duck = new Duck();
13         duck.swim();
14     }
15 }

```

`Swimmable` -- интерфейс. `Duck` -- класс, объекты которого подходят под описание интерфейса. В классе должны быть определены все абстрактные методы интерфейса.

Хорошим тоном считается написать аннотацию `@Override`. Это явное обозначение, что метод переопределен. Если метод с аннотацией не определен в интерфейсе, то будет ошибка компиляции.

Абстрактные классы

Обычный абстрактный класс. Нельзя создать экземпляр абстрактного класса.

В отличие от интерфейса, может содержать поля.

```

1  abstract static class AbstractSwimmable extends Swimmable {
2      String name;
3
4      @Override
5      public void swim() {
6          System.out.println(name + ", плыви!");
7      }
8  }

```

Используются для того, чтобы, например, переопределить метод класса Object. Таким образом новое определение смогут использовать все классы, реализующие интерфейс.

Модификатор `final`

`final` класс -- класс, который нельзя унаследовать.

`final` метод -- метод, который нельзя переопределить.

`final` переменная -- переменная, значение которой нельзя поменять.

Модификатор `sealed`

Используется для классов или интерфейсов. Что-то среднее между `final` и не-`final`. Явно определяет список возможных наследников с помощью слова `permits`:

```

1  sealed interface Swimmable permits Duck {
2      ...
3  }

```

Фабричный метод

Статический метод интерфейса. Часто называется `of` или `from`. Суть этого метода в том, чтобы создать правильный класс в зависимости от того, какие аргументы были переданы. Как бы вместо явного вызова `new` можно вызвать метод.

```

1 interface Vector {
2     static Vector of(int x, int y, int z) {
3         if (x == 0 && y == 0 && z == 0) {
4             return new ZeroVector();
5         }
6         return new ArrayVector(x, y, z);
7     }
8 }

```

Enum

Класс, в котором фиксированное количество экземпляров. Экземпляры создаются все сразу при первом обращении к классу.

```

1 enum Weekday {
2     MONDAY,
3     TUESDAY,
4     WEDNESDAY,
5     THURSDAY,
6     FRIDAY,
7     SATURDAY,
8     SUNDAY
9 }

```

Может иметь свои поля, методы и конструкторы. Не может быть унаследован.

```

1 enum Weekday {
2     // экземпляры
3     MONDAY("MON", false), TUESDAY("TUE", false),
4     WEDNESDAY("WED", false), THURSDAY("THU", false),
5     FRIDAY("FRI", false), SATURDAY("SAT", true),
6     SUNDAY("SUN", true);
7
8     // поля
9     private final String shortName;
10    private final boolean weekend;
11
12    // конструктор
13    Weekday(String shortName, boolean weekend) {
14        this.shortName = shortName;
15        this.weekend = weekend;
16    }
17
18    // методы
19    public String getShortName() { return shortName; }
20    public boolean isWeekend() { return weekend; }
21 }

```

Перечисление

Некоторые полезные методы, сгенерированный компилятором автоматически:

- `<EnumClassName>.values()` -- возвращает массив из всех enum-констант
- `<EnumClassName>.valueOf(String)` -- значение по имени экземпляра (или исключение)
- `<EnumValue>.name()` -- узнать имя экземпляра
- `<EnumValue>.ordinal()` -- узнать порядковый номер экземпляра в enum-классе.

Индексация начинается с нуля.

switch, используя экземпляры enum-класса

```

1  String workingHours(Weekday weekday) {
2      return switch (weekday) {
3          case MONDAY, FRIDAY -> "9:30-13:00";
4          case TUESDAY, THURSDAY -> "14:00-17:30";
5          case WEDNESDAY, SATURDAY, SUNDAY -> "Выходной";
6      };
7  }
8
9  for (Weekday weekday : Weekday.values()) {
10     System.out.println(workingHours(weekday))
11 }

```

Dependency Injection (DI)

Dependency injection is a way of organizing and connecting different parts of a software application. It allows the dependencies of a class to be provided externally rather than being created or managed within the class itself.

There are three types of DI:

- **Constructor Injection** -- dependencies are provided as parameters in the constructor:

```

1 public class MyClass {
2     private Dependency dependency;
3
4     public MyClass(Dependency dependency) {
5         this.dependency = dependency;
6     }
7 }

```

- **Setter Injection** -- dependencies are set through the setter methods:

```

1 public class MyClass {
2     private Dependency dependency;
3
4     public void setDependency(Dependency dependency) {
5         this.dependency = dependency;
6     }
7 }

```

- **Method Injection** -- dependencies are passed as parameters to the method of the class:

```

1 public class MyClass {
2     ...
3     public void doSomething(Dependency dependency) {
4         // Use the dependency here
5     }
6 }

```

Пакеты

Пакет -- это объединение классов. Название начинается с маленькой буквы. Чтобы создать пакет, нужно перед определением классов вставить строку

```

1 package packageName;

```

Пакеты должны располагаться в соответствующих директориях, то есть файл пакета `Name` должен быть сохранен в папке `Name`.

Пакеты могут быть вложенными. По внешнему виду сложно сказать, где класс, а где пакет. Для этого есть договоренность, что пакеты именуются с маленькой буквы, а классы -- с большой.

Полнокавалифицированное имя класса
Fully-qualified class name (FQN)

`java.util.PrimitiveIterator.OfInt`

Имя пакета Package name	Имя класса верхнего уровня Top level class name	Простое имя класса Simple name
----------------------------	---	--------------------------------------

Имя класса в виртуальной машине
JVM class name

`java/util/PrimitiveIterator$OfInt`

Модули

Модуль -- объединение пакетов. Задаёт явный ациклический граф зависимостей. Обладает сильной инкапсуляцией: если какой-то пакет не экспортирован, то и использовать его не получится.

```

1 module demo {
2     requires java.xml;
3     requires java.desktop;
4     exports com.example.demo; // экспорт пакета
5 }
```

Название модуля с маленькой буквы.

23-02-23

Исключения

```

1 public int get(int index) {
2     if (index < 0) {
3         throw new IllegalArgumentException("index < 0");
4     }
5     return array[index];
6 }
```


В джаве `stack trace` заполняется в момент создания исключения, а не в момент его выкидывания. Это отличает джаву от других языков. Если `new` и `throw` в разных местах кода, может возникнуть непонимание.

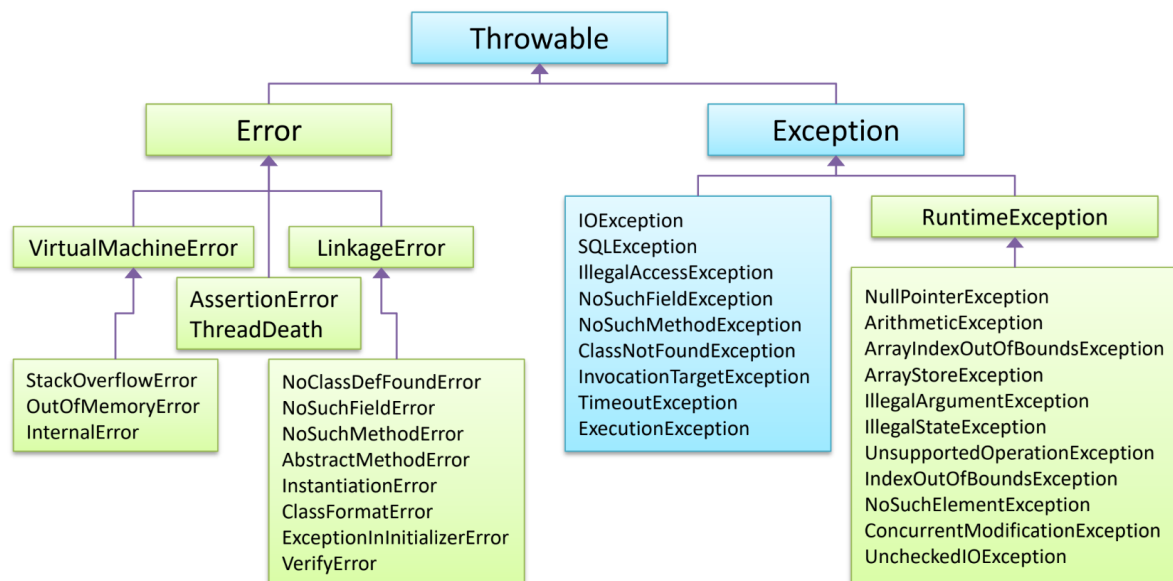
Можно у метода явно указать, что он кидает исключение:

```
1 public int get(int index) throws IllegalArgumentException {
2     if (index < 0) {
3         throw new IllegalArgumentException("index < 0");
4     }
5     return array[index];
6 }
```

Иерархия

Главный класс -- `Throwable`. Живет в библиотеке `java.lang.Throwable`. Все исключения, в том числе кастомные, наследуются от него.

Основные классы исключений представлены на картинке. Голубым отмечены проверяемые исключения (то есть желательно -- но не обязательно, -- чтобы они были указаны после слова `throws`), желтым -- непроверяемые. Суть голубых исключений в том, что



Конструкторы

```

1 Throwable();
2 Throwable(message); // exception with message
3 Throwable(message, cause); // exception with message and cause. Cause is
  another exception
4                               // which was the reason for throwing current
  exception.

```

Полезные методы у исключений

```

1 Exception exception = new Exception();
2 exception.getMessage(); // get message of exception
3 exception.getCause();   // get cause of exception
4
5 exception.getStackTrace(); // stack trace is an array of objects. Each object
  stores name of class,
6                               // name of method and number of line where
  exception was thrown.
7
8 exception.getSuppressed(); // if exception B is thrown inside exception A,
  exception A will be suppressed and saved.

```

try-catch

```

1 public int get(int index) {
2     if (index < 0) {
3         throw new IllegalArgumentException("get(index = " + index.toString()
4         + "): index < 0\n");
5     }
6     return array[index];
7 }
8
9 public static void main(String[] args) {
10     try {
11         get(-1);
12     } catch (IllegalArgumentException ex) {
13         System.out.println("message: " + ex.getMessage());
14         System.out.println(ex.getStackTrace());
15     }
16 }

```

Есть еще ключевое слово `finally`, тогда при любом исходе блок с `finally` будет выполнен:

```

1 public static void main(String[] args) {
2     try {
3         get(-1);
4     }
5     catch (IllegalArgumentException ex) {
6         System.out.println("message: " + ex.getMessage());
7         System.out.println(ex.getStackTrace());
8     }
9     finally {
10        System.out.println("Finishing main\n");
11    }
12 }

```

Еще про `finally`

Пример, как не надо делать:

```

1 public static int test() {
2     try {
3         return 5;
4     }
5     finally {
6         return 6;
7     }
8 }

```

Функция вернет `6`. Мы заходим в `try` блок и собираемся вернуть `5`. Перед тем, как сработает `return`, мы переходим в блок `finally`.

Логирование: пример использования

```

1 public class PoolComposerPrincipalEventIdentifier {
2     private static final Logger LOG = Logger.getLogger(
3         PoolComposerPrincipalEventIdentifier.class.getName()
4     );
5
6     public void identifyPrincipalEvent() {
7         try {
8             doIdentifyPrincipalEventUsingPoolComposer();
9         }
10        catch (Exception ex) {
11            LOG.log(Level.SEVERE, "Error while identifying principal event",
12                ex);
13        }
14    }
15 }

```

Надтипы и подтипы

Если какой-то класс реализует интерфейс, то интерфейс является *надтипом*, а соответствующий класс -- *подтипом*.

Если класс A расширяет класс B , то B -- это *надтип*, а A -- это *подтип*.

Надтипы и подтипы образуют направленный граф. На них определен частичный порядок.

Надтип не должен предоставлять больше возможностей, чем предоставляет подтип (вообще, кажется, это, в принципе, невозможно. У наследников всегда больше функционала, чем у классов, от которых наследуемся). Подтип предоставляет больше конкретики, а надтип -- больше абстракции.

`A extends B` -- `A` является подтипом `B`.

`A super B` -- `A` является надтипом `B`.

Если тип `Derived` является подтипом `Base`, это не значит, что `MyClass<Derived>` будет подтипом `MyClass<Base>`.

Пример использования `extends`

```
1 static double getDoubleValue(Shmoption<Number> shmopt) {
2     return shmopt.get().doubleValue();
3 }
4
5 getDoubleValue(new IntegerShmoption(123));
```

Не скомпилируется, функция ожидает класс с типом `Number`, а не `Integer`.

```
1 static double getDoubleValue(Shmoption<? extends Number> shmopt) {
2     return shmopt.get().doubleValue();
3 }
4
5 getDoubleValue(new IntegerShmoption(123));
6 // В данном случае тип переданного аргумента -- Integer.
```

Теперь аргумент функции -- это класс любого типа, который является подтипом `Number`.

Пример использования `super`

```

1 static void setInteger(Shmoption<Integer> shmopt) {
2     shmopt.set(42);
3 }
4
5 NumberShmoption<Number> n = new NumberShmoption<>(123.45);
6 setInteger(n);

```

Не скомпилируется. Функция ожидает класс типа `Integer`.

```

1 static void setInteger(Shmoption<? super Integer> shmopt) {
2     shmopt.set(42);
3 }
4
5 NumberShmoption<Number> n = new NumberShmoption<>(123.45);
6 setInteger(n);

```

Теперь в функцию можно передавать класс с типом, который является надтипом `Integer`. Например, `Double`.

Covariant return type

Ковариантность -- это про приведение типов.

```

1 interface Supplier {
2     Object get();
3 }
4
5 interface StringSupplier extends Supplier {
6     @Override
7     String get();
8 }

```

Параметризация

Параметризация типов

Дженерики -- классы, объявленные обобщенными. Тип дженерика не может быть примитивным (поэтому существуют обертки `Integer`, `Long`, etc).

Пример

```

1  static class Option<T> {
2      private T value;
3
4      public Option(T value) { this.value = value; }
5
6      /** Never returns null */
7      public T get() {
8          if(value == null) throw new NoSuchElementException();
9          return value;
10     }
11     public T orElse(T other) { return value == null ? other : value; }
12     public boolean isPresent() { return value != null; }
13 }
14
15 public static void main(String[] args) {
16     Option<String> aboba = new Option<String>("aboba");
17
18     Option<String> abobaImplicit = new Option<>("aboba");
19     // в джаве 8 придумали оператор ромб:)
20     // нет необходимости явно указывать тип в конструкторе
21
22     Option<String> abobaImplicit = new Option("aboba");
23     // raw type. Можно не указывать тип и не использовать `<>`.
24     // нужно избегать их, как огня
25 }

```

```

1  public static void main(String[] args) {
2      Option<String> abobaImplicit = new Option<>("aboba");
3
4      Option<var> abobaImplicit = new Option<>("aboba");
5      // бывают ситуации, когда компилятор не справляется определить нужный
6      // пользователю тип.
7      // Например, поставит тип Object вместо String.
8      // Поэтому дети, не используйте одновременно `var` и оператор `<>`.
9  }

```

Маскировочный тип

? (wildcard)

? ⇔ используется какой-то тип, который расширяет класс `Object`.

```

1  static class Shmoption<T> {
2      T value;
3      public Shmoption(T value) { this.value = value; }
4      public T get() {
5          if(value == null) throw new NoSuchElementException();
6          return value;
7      }
8      public void set(T newValue) { value = newValue; }

```

```

9      public T orElse(T other) { return value == null ? other : value; }
10     public boolean isPresent() { return value != null; }
11 }
12
13 public static void main(String[] argv) {
14     Shmoption<> present = new Shmoption<>("yes");
15
16     System.out.println(present.isPresent()); // в данном случае не зависит от
        типа элемента
17
18     Object value = present.get(); // вернет какой-то тип, не известный.
19                                     // Но мы точно знаем, что это наследник
        `Object`
20
21     present.set(???); // сработает только с null.
22                       // Тип у объекта `present` не определен, он так и
        остается `?`.
23 }

```

Если поставить курсор между скобками и нажать `ctrl+P`, то идея покажет, аргументы какого типа ожидаются `0_0`

`? extends Number`

```

1 public static void main(String[] argv) {
2     Shmoption<? extends Number> number = new Shmoption<>(123);
3     Number n = number.get();
4     number.set(124);
5 }

```

В данном случае тип всё еще не известен, но мы точно знаем, что он числовой. Таким образом, метод `get()` возвращает не просто `Object`, а `Number`.

Наследование

```

1 // наследник типа Number
2 class NumberShmoption<N extends Number> extends Shmoption<N> {
3     public NumberShmoption(N value) { super(value); }
4 }
5
6 // наследник типа Integer
7 class IntegerShmoption extends NumberShmoption<Integer> {
8     public IntegerShmoption(Integer value) { super(value); }
9 }

```

Параметризация методов

Чаще всего делают статические параметризованные методы.

```
1 static <T> String returnString(T value) {
2     return value.toString();
3 }
```

```
1 static <T> void setNotNull(Shmoption<? super T> shmoption, T value) {
2     if (value == null) throw new IllegalArgumentException();
3     shmoption.set(value);
4 }
5
6 setNotNull(n, 123);
7 ShmoptionUtils.<Number>setNotNull(n, 123);
```

Тип указывать не обязательно. Если компилятор не справился самостоятельно вывести тип, его можно указать в `<>`, при этом нужно обязательно указать квалификатор (штука до точки, может быть `this` или название класса, если метод статический).

Дженерики и массивы

Первый вариант не скомпилируется не потянуло почему.

Второй вариант скомпилируется, потому что в байткоде не будет указан тип

```
1 Shmoption<Integer>[] arrayInt = new Shmoption<Integer>[10]; // does not
   compile
2 Shmoption<?>[] array = new Shmoption<?>[10]; // OK
```

Еще один вариант создать массив дженериков -- использование сырых типов:

```
1 Shmoption<Integer>[] arrayInt = new Shmoption[10]; // raw-type. Получаем
   предупреждение
2 Object[] obj = arrayInt;
3 obj[0] = new Shmoption<>("foo");
4 Shmoption<Integer> shmoption = arrayInt[0];
5 Integer x = shmoption.get(); // ClassCastException: в `arrayInt[0]` лежит
   `String`
6 System.out.println(x);
```


Стирание (erasure)

В процессе выполнения программы типовые параметры не существуют (стираются). Есть сами классы, но они не параметризованы с точки зрения виртуальной машины.

Преобразование типов может быть небезопасным (например, в коде `return (T)obj;` во время исполнения превращается в `return obj;`), это может привести к исключению `ClassCastException`.

Пример

```

1  Shmoption<Integer> integer = new Shmoption<>(10);
2
3  Shmoption<String> string = ((Shmoption<String>) integer);
4  // ошибка: `Integer` и `String` в разных ветках иерархии
5
6  Shmoption<?> any = integer;
7  Shmoption<String> string2 = (Shmoption<String>) any;
8  // скомпилируется, но появится предупреждение,
9  // потому что в байткоде будет просто `string2 = any`
10
11 String s = string2.get();
12 // ClassCastException: внутри `string2` живет `Integer`, а не `String`
13
14 NumberShmoption<Integer> number = (NumberShmoption<Integer>)integer;
15 // OK

```

23-03-02

varargs

Переменное число аргументов

```

1 static void printAll(Object... objects) {
2     for (Object object : objects) {
3         System.out.println(object);
4     }
5 }
6
7 printAll("a", 1, "b", 2.0);
8
9 Object[] objects = {"a", 1, "b", 2.0};
10 printAll(objects);
11
12 printAll(null, null); // OK. An array of nulls
13 printAll(null); // does not compile.

```

varargs + generics

```

1 static <T> boolean isOneOf(T value, T... options) {
2     for (T option : options) {
3         if (Objects.equals(value, option)) return true;
4     }
5     return false;
6 }

```

`Objects.equals` использует метод `equals` объекта, но делает это аккуратно: сначала проверяет, что объект не `null`, потом вызывает метод.

Используя `varargs` и дженерики в методе, можно получить ворнинг типа `unchecked generic array creation for varargs parameter`. Если метод не делает ничего противозаконного, его можно пометить как безопасный с помощью аннотации `@SafeVarargs`:

```

1 @SafeVarargs
2 static <T> boolean isOneOf(T value, T... options) {
3     for (T option : options) {
4         if (Objects.equals(value, option)) return true;
5     }
6     return false;
7 }
8
9 isOneOf(shmoption, new Shmoption<>("foo"),
10          new Shmoption<>("bar"));

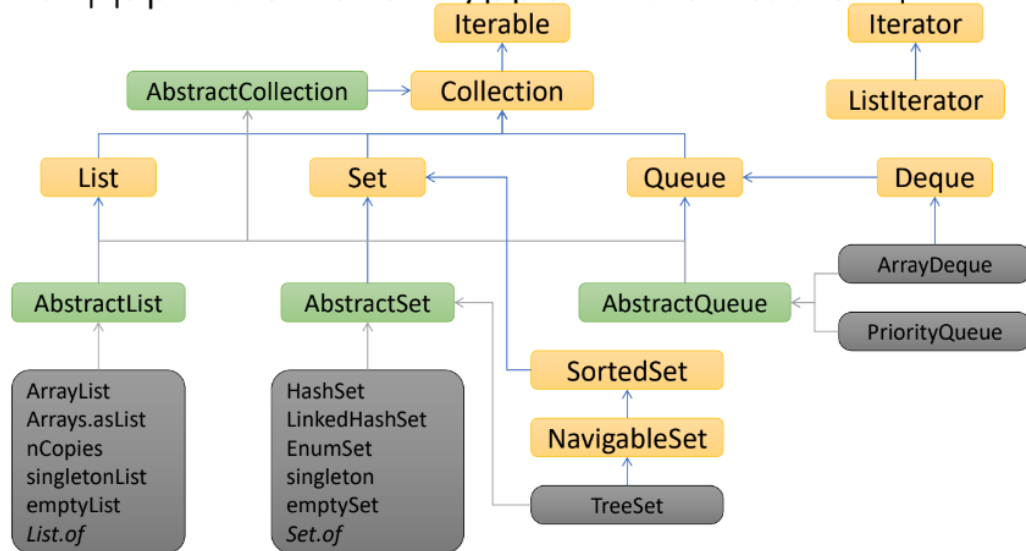
```

Collections

Иерархия

Желтым отмечены интерфейсы, зеленым -- абстрактные классы, серым -- конкретные реализации (классы или методы).

Стандартные неконкурентные коллекции



Одновременный read & write

Например, если мы в цикле `for` обходим элементы коллекции и одновременно с этим модифицируем коллекцию, вылетит исключение `ConcurrentModificationException`:

```

1  import java.util.List;
2  import java.util.ArrayList;
3
4  public class MyClass {
5      public static void main(String[] args) {
6          List<Integer> list = new ArrayList<>(List.of(1, 2, 3));
7          for (Integer a : list) {
8              System.out.println(a);
9              list.add(4);
10         }
11     }
12 }
  
```

```

1  Exception in thread "main" java.util.ConcurrentModificationException
2      at
3      java.base/java.util.ArrayList$Itr.checkForComodification(ArrayList.java:1013)
4      at java.base/java.util.ArrayList$Itr.next(ArrayList.java:967)
   at MyClass.main(MyClass.java:7)
  
```

Не надо использовать

- ✓ Enumeration -> Iterator
- ✓ Vector -> ArrayList
- ✓ Stack -> ArrayDeque
- ✓ Dictionary -> Map
- ✓ Hashtable -> HashMap
- ✓ LinkedList -> ArrayList/ArrayDeque

У штук слева старая реализация, которая может работать долго.

23-03-16

Функциональный интерфейс

Интерфейс (не абстрактный класс), содержащий единственный абстрактный метод (single abstract method -- SAM). Может быть аннотирован как `@FunctionalInterface` (тогда компилятор будет проверять, что интерфейс функциональный).

Пример SAM -- интерфейс `Runnable` с абстрактный методом `Run()`. Другой пример -- интерфейс `Function`, у которого всего 4 метода, но абстрактным является только один.

Еще пример (неочев):

```

1  @FunctionalInterface
2  interface LongSupplier {
3      long getLong(); // sam, since not implemented
4  }
5
6  @FunctionalInterface
7  interface IntSupplier extends LongSupplier {
8      int getInt(); // sam, since not implemented
9      @Override
10     default long getLong() { return getInt(); }
11 }
```

Интерфейс `IntSupplier` является подтипом функционального интерфейса, появляется реализация `getLong()`. Но внутри создается новый (и единственный) абстрактный метод `getInt()`, поэтому `IntSupplier` становится функциональным интерфейсом.

Функциональное выражение

Экземпляры функциональных интерфейсов создаются с помощью функциональных выражений.

У функционального выражения нет внутреннего состояния, оно не может хранить значения. Функциональное выражение -- это всегда реализация какого-то функционального интерфейса.

Тип функционального выражения явно не указан, поэтому он определяется по контексту (aka [поливывражения](#)). Есть два возможных типа:

- лямбда `((a, b) -> a + b)`
- ссылка на метод `(Integer::sum)`

Не гарантируется идентичность функциональных выражений. Одно и то же функциональные выражение не обязательно создает один и тот же объект класса.

Не гарантируется идентичность `getClass()` у функциональных выражений. Не обязательно, что одно и то же функциональные выражение создает объект одного и того же класса.

Короче, методы вроде `getClass()`, `toString()` не используем.

Примеры

```
1 public static void main(String[] args) {
2     Runnable r = () -> System.out.println("Hello World!");
3     r.run();
4 }
```

Функциональное выражение -- лямбда -- присвоилась переменной `r`.

В строке `r.run()` вызывается тот самый единственный абстрактный метод интерфейса (в данном случае, метод `run()`).

При использовании функциональных выражений нужно, чтобы тип интерфейса явно определялся.

```
1 static void run(Runnable r) {
2     r.run();
3 }
4 public static void main(String[] args) {
5     run(() -> System.out.println("Hello World!"));
6 }
```

В этом примере тип интерфейса, используемого для лямбды, определяется внутри функции `run(Runnable)`.

Лямбда-выражения

- аргументы: `(type1 name1, type2 name2), ()`
- стрелочка: `->`
- тело
 - выражение: `System.out.println("aboba");`
 - блок: `{ System.out.println("aboba"); }`

Возвращаемое значение

- Void-compatible (void SAM):
 - Выражение: допустимое в утверждении (вызов метода, присваивание и т. д.)
 - Блок: каждый `return` не содержит выражения
- Value-compatible (non-void SAM):
 - Выражение: имеет значение не `void`
 - Блок: каждый `return` содержит выражение и нормальное завершение невозможно.

Захват значений

1. Захватываем локальную переменную:

Она должна быть `effectively final` и инициализирована. Это означает, что нельзя изменять значение захваченной переменной внутри лямбды.

В лямбду отправляется *значение* переменной.

2. Захватываем поле класса:

Захватывается не само значение, а `this`.

3. Захватываем статическое поле:

Ничего не захватывается, тело лямбды становится `stateless`. Так как переменная статическая, ее значение подставится в момент вызова функции.

Пример:

```
1 import java.util.ArrayList;
2 import java.util.List;
```

```

3  import java.util.function.IntSupplier;
4
5  public class Demo {
6      static int x = 5;
7      public static void main(String[] args) {
8          List<IntSupplier> List = new ArrayList<>();
9          for (int i = 0; i < 5; i++) {
10             x++;
11             IntSupplier l1 = () -> x * x;
12             List.add(l1);
13         }
14         for (IntSupplier elem : List) {
15             System.out.println(elem.getAsInt());
16         }
17     }
18 }

```

Статическая переменная `x` увеличилась на 5. В строках 14-16 ее значение подставляется в лямбды.

Вывод:

```

1  100
2  100
3  100
4  100
5  100

```

Пример

Хотим создать несколько лямбд, которые возвращают разные целочисленные значения.

```

1  import java.util.ArrayList;
2  import java.util.List;
3  import java.util.function.IntSupplier;
4
5  public class Demo {
6      public static void main(String[] args) {
7          int x = 1;
8          List<IntSupplier> List = new ArrayList<>();
9          for (int i = 0; i < 5; i++) {
10             int xx = x++;
11             IntSupplier l1 = () -> xx * xx;
12             List.add(l1);
13         }
14         for (IntSupplier elem : List) {
15             System.out.println(elem.getAsInt());
16         }
17     }
18 }

```

Переменная `xx` является для лямбды effectively final, в нее копируется текущее значение `x`.
Лямбда создается с захватом значения из `xx`.

Вывод:

```
1 1
2 4
3 9
4 16
5 25
```

Optional<T>

Коробочка для значений. Можно применять разные функции в зависимости от того, есть ли что-то внутри:

```
1 // filter(Predicate<T>)
2 boolean isFooEqualsBar =
  Optional.of("foo").filter("bar"::equals).isPresent();
3
4 // <U> map(Function<T, U>)
5 String trimmed = Optional.of(" foo ").map(String::trim).get();
6
7 // <U> flatMap(Function<T, Optional<U>>)
8 Integer num = Optional.of("1234").flatMap(x -> toInteger(x)).orElse(-1);
9
10 // orElseGet(Supplier<T>)
11 Double random = Optional.<Double>empty().orElseGet(Math::random);
12
13 // <X extends Throwable> orElseThrow(Supplier<T>) throws X
14 Double random = Optional.<Double>empty().orElseThrow(Exception::new);
15
16 //or(Supplier<? extends Optional<? extends T>>)
17 getOneOptional().or(() -> getAnotherOptional());
```

Пример 1: update karma

```
1 interface User {
2     String name();
3     boolean isActive();
4     void updateCarma(int delta);
5 }
6
7 interface UserRepository {
8     Optional<User> findUser(String name);
```



```

9   }
10
11  void increaseUserCarma(UserRepository repository, String name) {
12      repository.findUser(name)
13          .filter(User::isActive)
14          .ifPresent(user -> user.updateCarma(1));
15
16  }

```

Хотим по имени найти юзера в репозитории юзеров. Если он активный, улучшим карму. Иначе ничего не делаем.

Пример 2: update + throw

```

1  interface User {
2      String name();
3      boolean isActive();
4      void updateCarma(int delta);
5  }
6
7  interface UserRepository {
8      Optional<User> findUser(String name);
9  }
10
11  void increaseUserCarma(UserRepository repository, String name) {
12      repository.findUser(name)
13          .filter(User::isActive)
14          .orElseThrow(() -> new IllegalArgumentException("No such user:
15      "+name))
16          .updateCarma(1);
17  }

```

Находим юзера по имени. Если не нашли, кидаем исключение. Если нашли и он активный, улучшаем карму.

Stream API

Из библиотеки `java.util.stream`.

Есть

- источник (создает стрим)
- промежуточные операции (формируют последовательность команд, но сами команды не выполняют. Команды выполняются, когда появляется терминальная операция)
- терминальные операции (функции, которые вызываются в конце)

Существуют `Stream`, `IntStream`, `LongStream`, `DoubleStream`.

Источники

```

1 Stream.empty();
2
3 Stream.of(x, y, z); // из существующих элементов
4
5 // если null, вернется пустой стрим
6 // если не null, вернется стрим из одного элемента
7 Stream.ofNullable(x);
8
9 // бесконечный стрим
10 Stream.generate(Math::random);
11
12 // следующий элемент зависит от предыдущего
13 // val -- начальное значение
14 Stream.iterate(val, x -> x+1);
15
16 // то же, но в условии
17 Stream.iterate(0, x -> x < 100, x -> x + 1);
18
19 // любую коллекцию можно превратить в стрим
20 collection.stream();
21
22 // массив можно превратить в стрим
23 // должен быть тип int, long, double или Object
24 // с остальными упр
25 Arrays.stream(array);
26
27 // стрим чаров, полученный из строки
28 // внутри хранятся не символы, а их численные значения
29 String.chars();
30
31 // как ленивый стрим. Хорош для длинных строк
32 Pattern.splitAsStream();

```

Промежуточные операции

операции	комментарии
map	Из стрима получается стрим. Есть <code>mapToInt</code> , <code>mapToLong</code> , <code>mapToDouble</code> , <code>mapToObj</code> , превращают старый тип в новый из названия.

операции	комментарии
filter	Выкидывает элементы по предикату. Не меняет изначальную сортированность
flatMap	Преобразует стрим стримов в стрим. Есть flatMapToInt, flatMapToLong, flatMapToDouble
mapMulti	mapMultiToInt, mapMultiToLong, mapMultiToDouble
distinct	Избавиться от повторяющихся элементов
sorted	Сортирует
limit	Ограничивает количество элементов в стриме
skip	Скипает первые сколько-то элементов
peek	Можно применить переданную функцию к каждому элементу, чтобы посмотреть, какие элементы в стриме
takeWhile	Берем элементы, пока они удовлетворяют условию (предикату).
dropWhile	Скипаем элементы, пока они удовлетворяют условию (предикату).
boxed	Преобразует стрим в стрим от <code>Integer</code> .
asLongStream	
asDoubleStream	

Пример `peek`

```

1 long count = Stream.of(1, 2, 3, 4, 5)
2   .peek(System.out::println)
3   .count();

```

В старших версиях джавы промежуточные операции (в данном случае, `peek`) могут скипаться, если результат известен заранее.

Здесь изначально известно, чему равен `count`.

Пример flatMap

```

1 List<List<String>> listOfLists =
2     List.of(List.of("foo", "bar", "baz"),
3     List.of("Java", "Kotlin", "Groovy"),
4     List.of("Hello", "Good Bye"));
5
6 System.out.println(listOfLists.stream()
7     .flatMap(List::stream) // стрим стримов -> стрим
8     .anyMatch("Java"::equals));

```

Пример, как не надо делать: shuffle

Хотим рандомизировать порядок в стриме. Так плохо:

```

1 IntStream.range(1, 10)
2     .boxed()
3     .sorted((a, b) -> Math.random() > 0.5 ? 1 : -1)
4     .forEach(System.out::println);

```

`.boxed()` -- превращаем инты в `Integer`, только так можно использовать кастомный компаратор

`.sorted()` -- передаем компаратор

Проблема: при изменении $10 \rightarrow 50$ нарушается контракт (какой-то). Какой-то там алгоритм меняется при увеличении количества элементов, и все взрывается.

Правильное решение -- через `Collections`:

```

1 List<Integer> list =
2     IntStream.range(1, 50)
3         .boxed()
4         .collect(Collectors.toList());
5
6 Collections.shuffle(list);
7
8 list.forEach(System.out::println);

```

Пример, как не надо делать: порядок промежуточных операций

```

1 IntStream.iterate(1, x -> x * 2)
2     .sorted()
3     .limit(10)
4     .forEach(System.out::println);

```

Упадет с ошибкой, потому что пытаемся сортировать бесконечный список. Правильный вариант:

```
1 IntStream.iterate(1, x -> x * 2)
2     .limit(10)
3     .sorted()
4     .forEach(System.out::println);
```

Терминальные операции

- ✓ count (возвращает `long`)
- ✓ findFirst/findAny -> Optional
- ✓ anyMatch/noneMatch/allMatch
- ✓ forEach/forEachOrdered
- ✓ max/min -> Optional
- ✓ reduce
- ✓ collect
- ✓ toArray
- ✓ toList (Java 16+)
- ✓ sum/average/summaryStatistics (мин, макс, среднее) (примитивы)

Пример `anyMatch`

```
1 boolean hasOption = Stream.of(args)
2     .filter(x -> x.startsWith(PREFIX))
3     .findFirst()
4     .isPresent();
```

Можно проще:

```
1 boolean hasOption = Stream.of(args).anyMatch(x -> x.startsWith(PREFIX));
```

Пример `reduce`

Функция редукции выводит результат какой-то функции, примененной ко всем элементам.

```

1 static BigInteger factorial(int n) {
2     return IntStream.rangeClosed(1, n)
3         .mapToObj(BigInteger::valueOf)
4         .reduce(BigInteger.ONE, BigInteger::multiply);
5 }

```

Первый аргумент `reduce` -- начальное значение, должно быть нейтральным элементом для выбранной функции. В коде `1` -- нейтральный элемент для умножения: `1 * 1 = 1`.

Второй аргумент `reduce` -- ассоциативная функция. Может быть лямбдой, принимающей два аргумента.

Dont overreduce! Можно получить кучу мусора в виде временных элементов.

Collectors

Способ комбинирования элементов в единое целое. Вызывается с помощью `.collect()`.

Некоторые коллекторы можно комбинировать.

Можно написать кастомный коллектор (`Collector.of`).

✓ **`toList()`** -- возвращает `ArrayList`

✓ **`toSet()`**

✓ **`toCollection(supplier)`** -- создает пустую коллекцию, пример:

```
toCollection(ArrayList::new)
```

✓ **`toMap(keyMapper, valueMapper[, merger[, mapSupplier]])`** -- передаются две функции: одна создает ключи карты, другая -- значения. Если ключи повторяются, по умолчанию, кидается исключение. Если хочется этого избежать, передается третья функция, решающая коллизии. Можно передать четвертый параметр, который выберет реализацию карты (`HashMap::new` или `TripleMap::new`)

✓ **`joining(separator[, prefix, suffix])`** -- склеивает *строки*. Есть необязательные параметры. Первый -- это сепаратор. Второй -- это префикс. Третий -- суффикс. Пример: `joining(' ', '(', ')')`

✓ **`groupingBy(classifier[, mapSupplier], downstream)`** -- группирует элементы по ключам. Первый обязательный аргумент -- классификатор -- значение для ключей. Второй необязательный -- способ объединения элементов, у которых одинаковый ключ. По умолчанию, такие элементы объединяются в массив. В итоге получаются две группы с ключом `true` и с ключом `false`, даже если одна из них пустая.

✓ **partitioningBy(predicate[, downstream])** -- разновидность `groupingBy`, но в качестве классификатора берется предикат, который возвращает `true` или `false`. Первый аргумент (предикат) обязательный.

✓ **reducing/counting/mapping/minBy/maxBy**

✓ **averagingInt/averagingLong/averagingDouble**

✓ **summingInt/summingLong/summingDouble**

Примеры

Пример: группируем строки по длине

Хотим сделать мапу, в которой ключи -- это длина, значения -- строки этой длины.

```
1 static Map<Integer, String> stringsByLength(List<String> list) {
2     return list.stream().collect(
3         Collectors.groupingBy(String::length, Collectors.joining(","))
4     );
5 }
6
7 stringsByLength(Arrays.asList("a", "bb", "c", "dd", "eee"));
8 // {1=a,c, 2=bb,dd, 3=eee}
```

Пример: вложенные коллекторы

Сначала группируем по длине, потом внутри значения с одинаковым ключом группируем по первой букве.

```
1 static Map<Integer, Map<Character, List<String>>>
2     stringsByLengthAndFirstLetter(List<String> list) {
3     return list.stream().collect(
4         Collectors.groupingBy(
5             String::length,
6             Collectors.groupingBy(s -> s.charAt(0)))
7     );
8 }
9 stringsByLengthAndFirstLetter(Arrays.asList("a", "b", "aa", "ab", "ba"));
10 // {1={a=[a], b=[b]}, 2={a=[aa, ab], b=[ba]}}
```

Пример toMap

Хотим получить мапу, где ключи -- категории, значения -- минимальная стоимость в категории.

```

1  interface Book {
2      String getCategory();
3      int getPrice();
4  }
5
6  static Map<String, Integer> getLowestBookPriceByCategory(List<Book> list) {
7      return list.stream().collect(Collectors.toMap(
8          Book::getCategory,
9          Book::getPrice,
10         BinaryOperator.minBy(Comparator.naturalOrder())
11     ));
12 }

```

Ключи -- `Book::getCategory`, значения -- `Book::getPrice`. Если значений несколько, применяем сапплайер `BinaryOperator.minBy()`.

Как устроен коллектор

Это интерфейс, у которого четыре метода, возвращающие функции. Есть три типовых параметра: `T` (тип стрима), `A` (аккумулятор - накопитель), `R` (результат).

```

1  public interface Collector<T, A, R> {
2      // Создать накопитель
3      Supplier<A> supplier();
4
5      // Добавить элемент в накопитель
6      BiConsumer<A, T> accumulator();
7
8      // Склеить два накопителя
9      BinaryOperator<A> combiner();
10
11     // Преобразовать накопитель в результат
12     Function<A, R> finisher();
13
14     // Флаги (ерунда всякая)
15     Set<Characteristics> characteristics();
16 }

```

Тип накопителя не должен быть известен пользователю

Создание коллектора

Первый аргумент -- сапплайер -- создает накопитель.

- Каждый раз должен создаваться новый независимый объект, чтобы данный колллектор можно было использовать в многопоточке

Второй аргумент -- накопитель -- описывает, как элемент будет добавляться в накопитель.

- Тип накопителя не должен быть известен пользователю (так остается возможность оптимизировать накопитель в будущем, без взрыва чужого кода)

Третий аргумент -- комбайнер -- описывает, как склеить два накопителя.

- Когда вызывается комбайнер, гарантируется, что накопители потом не будут использоваться, поэтому их состояние можно менять.
- Не реализовывать комбайнер -- плохой тон. Комбайнер хорош для многопоточки, а создавать коллектор, который не поддерживает многопоточку, -- неправильно.

Четвертый необязательный аргумент -- finisher -- преобразовывает накопитель в результат.

```

1  public static<T, R> Collector<T, R, R> of
2      (Supplier<R> supplier,
3       BiConsumer<R, T> accumulator,
4       BinaryOperator<R> combiner,
5       Characteristics... characteristics)
6  {
7      ...
8  }
9
10 public static<T, A, R> Collector<T, A, R> of
11     (Supplier<A> supplier,
12      BiConsumer<A, T> accumulator,
13      BinaryOperator<A> combiner,
14      Function<A, R> finisher,
15      Characteristics... characteristics)
16 {
17     ...
18 }
```

Пример с `toList`

```

1 static <T> Collector<T, ?, List<T>> toList() {
2     return Collector.of(
3         ArrayList::new,
4         List::add,
5         (acc1, acc2) -> {
6             acc1.addAll(acc2);
7             return acc1;
8         }
9     );
10 }

```

Элементы типа `T`, накопитель типа `List<T>`, результат типа `List<T>`.

В качестве накопителя используется `ArrayList`. Сапплайер создает накопитель, поэтому в качестве первого аргумента берется `ArrayList::new`.

Второй аргумент -- `BiConsumer` -- как будем добавлять элемент. Подходит `List::add`.

Третий аргумент -- комбайнер. Используем лямбду. Состояние переданных накопителей можно менять.

Возвращаемый тип -- `Collector<T, ?, List<T>>`. С вопросиком, потому что пользователю не надо знать, какой тип у накопителя.

Пример: коллектор, считающий и макс, и мин

Хотим использовать коллектор так:

```

1 Stream.of("one", "two", "three", "four", "five", "six", "seven", "eight")
2     .collect(minMax(Comparator.comparingInt(String::length),
3                     (min, max) -> min+"|"+max))
4     .ifPresent(System.out::println);

```

`minMax` -- коллектор, который хотим реализовать. Пользователь может сам решить:

- по какому критерию сравниваются элементы (первый аргумент `minMax` -- компаратор)
- как комбинировать минимум и максимум (второй аргумент `minMax` -- лямбда)

`minMax` возвращает `Optional<R>` (чтобы в том числе работало на пустом стриме)

Реализация

```

1 // возвращаемый тип -- коллектор с `Optional<R>`
2 static <T, R> Collector<T, ?, Optional<R>> minMax
3     // `cmp` -- компаратор
4     (Comparator<? super T> cmp,
5     // `finisher` -- функция, получающая макс и мин и возвращающая их
6     комбинацию
7     BiFunction<? super T, ? super T, ? extends R> finisher)

```

```

7  {
8      // накопитель, реализованный как внутренний класс
9      // не может быть null
10     // функция для сапплайера -- `Acc::new`
11     class Acc {
12         T min;
13         T max;
14         boolean present; // =false, если стрим пустой
15
16         void add(T t) {
17             if(present) {
18                 if(cmp.compare(t, min) < 0) min = t;
19                 if(cmp.compare(t, max) > 0) max = t;
20             } else {
21                 min = max = t;
22                 present = true;
23             }
24         }
25
26         // функция для комбайнера
27         // используем `cmp`, переданный пользователем
28         Acc combine(Acc other) {
29             if(!other.present) return this;
30             if(!present) return other;
31             if(cmp.compare(other.min, min) < 0) min = other.min;
32             if(cmp.compare(other.max, max) > 0) max = other.max;
33             return this;
34         }
35
36         // функция для финишера
37         // используем `finisher`, переданный пользователем
38         Optional<T> finish(Acc acc) {
39             if (acc.present) {
40                 return Optional.of(finisher.apply(acc.min, acc.max));
41             }
42             return Optional.empty();
43         }
44     }
45
46     // возвращаем результат
47     return Collector.of(
48         Acc::new,
49         Acc::add,
50         Acc::combine,
51         Acc::finish
52     );
53 }

```

23-04-18

- ✓ `InputStream` – читать байты (низкоуровневый API)
- ✓ `OutputStream` – писать байты (низкоуровневый API)
- ✓ `Reader` – читать символы
- ✓ `Writer` – писать символы

InputStream

Для работы с байтами

Реализации

- ✓ `FileInputStream` – файл
- ✓ `ByteArrayInputStream` – массив байт
- ✓ `BufferedInputStream` – буферизированная обёртка
- ✓ `DataInputStream` – структурированные двоичные данные
- ✓ `PipedInputStream` – плохой, негодный
- ✓ `ZipFile.getInputStream(ZipEntry)` – файл из зипа (без распаковки!)
- ✓ `Process.getInputStream()` – стандартный вывод процесса
- ✓ `URL.openStream()` – содержимое по URL (http/https/file/ftp/etc.)
- ✓ ...

Методы

Есть три метода для чтения. Все методы блокирующие (блокируется текущий thread)

- ✓ `int read()` → `-1` или `0..255` -- считывает один байт из потока и возвращает его или `-1`, если поток закончился (EOF)
- ✓ `int read(byte[] b)` → количество прочитанных байт (длина массива `b`)
- ✓ `int read(byte[] b, int offset, int length)` → количество прочитанных байт в фрагменте массива `b`
- ✓ `long skip(long n)` → количество пропущенных байт -- пропустить какое-то число байт
- ✓ `int available()` → количество байт, доступных без блокировки

- ✓ `void mark(int readlimit)` -- пометить, что, например, прочитаем не больше 100 байт, и потом прочитать не больше 100 байт
- ✓ `void reset()` -- вернуть поток в состояние, в котором он был до вызова `mark`
- ✓ `boolean markSupported()`

Пример

```

1  import java.io.InputStream;
2  import java.util.Arrays;
3  import java.io.IOException;
4
5  public class Main {
6      static void readFully(InputStream is, byte[] b) throws IOException {
7          int offset = 0;
8          while(offset < b.length) {
9              int count = is.read(b, offset, b.length - offset);
10             if(count == -1) {
11                 throw new IOException("Stream has less than " + b.length + "
bytes");
12             }
13             offset += count;
14         }
15     }
16
17     public static void main(String[] args) throws IOException {
18         byte[] b = new byte[10];
19         readFully(System.in, b);
20
21         System.out.println("result is " + Arrays.toString(b));
22         for (byte el : b) {
23             System.out.println(el + " --> " + (char) el);
24         }
25     }
26 }

```

Современный InputStream

- ✓ `byte[] readAllBytes()` → всё содержимое (Java 9+)
- ✓ `byte[] readNBytes(int len)` → не больше len байт (Java 11+)
- ✓ `int readNBytes(byte[] b, int offset, int length)` → количество прочитанных байт (до ошибки или конца потока, Java 11+)
- ✓ `void skipNBytes(long n)` → пропускает n байт (EOFException, если меньше)
- ✓ `long transferTo(OutputStream out)` → перекачивает в выходной поток (Java 9+)

✓ `static InputStream nullInputStream()` → пустой поток (Java 11+)

Управление внешними ресурсами: раскрытие файла

Явное закрытие файла -- метод `close()`.

Простой, но неправильный пример:

```
1  InputStream f = new FileInputStream("/etc/passwd");
2  try {
3      int b = f.read();
4  } finally {
5      f.close();
6  }
```

Без try-блока будет плохо. Например, вылетит ошибка при чтении байта, и файл будет вечно висеть открытым, пока работа приложения не завершится.

По-хорошему надо делать большую лесницу из try-блоков, которая увеличивается с увеличением числа открытых файлов. Вместо этого используется "продвинутый" try-блок [НАЗВАНИЕ УДАЛЕНО]:

```
1  try (InputStream in = new FileInputStream("/etc/passwd");
2      OutputStream out = new FileOutputStream("/etc/passwd.bak")) {
3      in.transferTo(out);
4  }
5
6  // после `try` в скобках указываются файлы на открытие
7  // в теле `try`-блока происходят махинации с открытыми файлами
```

Такая штука гарантирует, что

- открытие файлов будет корректным
- ресурсы будут закрыты в порядке, обратном открыванию
- при выкидывании исключения, все файлы будут корректно закрыты

OutputStream

Для работы с байтами

Реализации

- ✓ `FileOutputStream` – файл
- ✓ `ByteArrayOutputStream` – массив байт
- ✓ `BufferedOutputStream` – буферизированная обёртка
- ✓ `PrintStream` – для удобства печати (`System.out`)
- ✓ `DataOutputStream` – структурированные двоичные данные
- ✓ `PipedOutputStream` (тоже плохой)
- ✓ `Process.getOutputStream()` – стандартный ввод процесса
- ✓ `URL.openConnection().getOutputStream()` – писать в URL (HTTP POST)
- ✓ ...

Методы

Методы для записи, могут кинуть `IOException`.

✓ `void write(int b)` → записывает младшие 8 бит. Передается тип `int` $\in [0..255]$ вместо `byte`, который знаковый и $\in [-128, 127]$.

✓ `void write(byte[] b)` → пишет весь массив (если не упал, то всё записал. Не может быть такого, что записалась часть)

✓ `void write(byte[] b, int offset, int length)` → часть массива

✓ `void flush()` -- сбрось кэши куда-нибудь (например, в файл)

✓ `void close()`

✓ `static OutputStream nullOutputStream()` → всё, записанное сюда, игнорируется, писать можно бесконечно

Note: у `ByteArrayOutputStream` реализация такая, что перечисленные методы не кидают `IOException`. Кроме `write(byte[] b)`. Прикол.

Пример

```

1 public static void main(String[] args) {
2     // создаем массив байт для записи
3     ByteArrayOutputStream baos = new ByteArrayOutputStream();
4
5     baos.write(0x01);
6     baos.write(0x02);
7     baos.write(0x03);
8
9     byte[] result = baos.toByteArray();
10
11     System.out.println(Arrays.toString(result));
12     // [1, 2, 3]
13 }

```

Reader

Для работы с символами

Реализации

- ✓ `InputStreamReader (InputStream + charset)` -- чтение из `InputStream`
- ✓ `FileReader` (осторожно) -- чтение из файла
 - В Java 11 появился `FileReader(file, charset)`
 - В Java 18 используется UTF-8, по умолчанию
- ✓ `StringReader`
- ✓ `BufferedReader`
 - `String readLine()` -- считать строку до ближайшего `\n`
 - `Stream<String> lines()` -- получить стрим, состоящий из строк
- ✓ ...

Методы

- ✓ `int read()` → -1 или 0..0xFFFF
- ✓ `int read(char[] cb)` → количество прочитанных символов
- ✓ `int read(char[] cb, int offset, int length)` → количество прочитанных символов

✓ `long skip(long n)` → количество пропущенных символов. Работает не так быстро, как `skip()` у байтов, потому что один символ может занимать > 1 байта.

✓ `boolean ready()` → можно ли прочитать хоть что-то без блокировки

✓ `void mark(int readlimit)`

✓ `void reset()`

✓ `boolean markSupported()`

✓ `long transferTo(Writer out)` – Java 10

✓ `static Reader nullReader()` – Java 11

Writer

Для работы с символами

Реализации

✓ `OutputStreamWriter(OutputStream + charset)` -- запись в `OutputStream`

✓ `FileWriter` (осторожно)

- В Java 11 появился `FileWriter(file, charset)`
- В Java 18 используется UTF-8, по умолчанию

✓ `StringWriter`

✓ `BufferedWriter`

✓ ...

Методы

✓ `void write(int b)` → пишет 16 младших бит

✓ `void write(char[] cb)` → пишет весь массив (если не упал, то всё записал)

✓ `void write(char[] cb, int offset, int length)` → часть массива

✓ `void write(String str)` → пишет всю строку (если не упал, то всё записал)

✓ `void write(String str, int offset, int length)` → часть строки

✓ `void flush()`

✓ `void close()`

✓ `static Writer nullWriter()` – Java 11

`java.io.File` (классика)

- ✓ Представляет и путь к файлу, и сам файл (каталог и т. д.)
- ✓ Может быть абсолютным или относительным
- ✓ Относительный считается относительно текущего каталога Java-процесса (и его крайне трудно поменять)

```
1 // new File(parentDirectory + "/" + fileName); -- bad
2 // new File(parentDirectory + File.separator + fileName); -- bad
3 File aboba = new File(parentDirectory, fileName); // OK
```

Методы

- ✓ `getName()/getPath()/getParent()/getParentFile()`
- ✓ `getAbsolutePath()/getAbsolutePath()` -- абсолютный путь/файл
- ✓ `getCanonicalFile()/getCanonicalPath()` -- то же самое, но можно переходить и по символическим ссылкам
- ✓ `exists()/canRead()/canWrite()/canExecute()/length()`
- ✓ `isDirectory()/isFile()/isHidden()`
- ✓ `createNewFile()/mkdir()/mkdirs()`
- ✓ `renameTo()/delete()`
- ✓ `list([filter])` (возвращает массив строк) / `listFiles([filter])` (возвращает массив файловых объектов)

Пример

```
1 import java.io.File;
2
3 public class Main {
4     public static void main(String[] args) {
5         String fname = args[0];
6         File f = new File(fname);
7
8         System.out.println("File name : " + f.getName());
```

```

9      System.out.println("Path: " + f.getPath());
10     System.out.println("Absolute path:"
11                          + f.getAbsolutePath());
12     System.out.println("Parent:" + f.getParent());
13     System.out.println("Exists :" + f.exists());
14
15     if (f.exists()) {
16         System.out.println("Is writable:"
17                             + f.canWrite());
18         System.out.println("Is readable" + f.canRead());
19         System.out.println("Is a directory:"
20                             + f.isDirectory());
21         System.out.println("File Size in bytes "
22                             + f.length());
23     }
24 }
25 }

```

java.nio.file.[Path/Paths/Files]

Превратить строку (строки) в путь

```

1 Path p = Paths.get("/etc/passwd");
2 Path p = Paths.get(parentDirectory, fileName);
3 Path p = Paths.get("foo", "bar", "baz");
4
5 // начиная с Java 11:
6 Path p = Path.of("/etc/passwd");
7 Path p = Path.of(parentDirectory, fileName);
8 Path p = Path.of("foo", "bar", "baz");

```

Методы

✓ `getFileName()`

✓ `getParent()`

✓ `getRoot()`

✓ `resolve(Path/String)`, `resolveSibling(Path/String)` -- ищет файл относительно текущего пути (первый -- в предках, второй -- в детях)

✓ `isAbsolute()` / `toAbsolutePath()`

✓ `startsWith(Path/String)`, `endsWith(Path/String)`

✓ `getName(int)`, `getNameCount()`, `iterator()`

✓ `toString()` / `toFile()` / `toUri()`

✓ `register(WatchService, WatchEvent.Kind...)`