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 ($1 \Rightarrow$ `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`:

```
int x;
x = 5;
var y = 5;
```

Тип выявляется неявно, но нужно обязательно проинициализировать переменную.

Если попытаться вывести непроинициализированную переменную, код не скомпилируется:

```
int x;
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 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`:

```
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

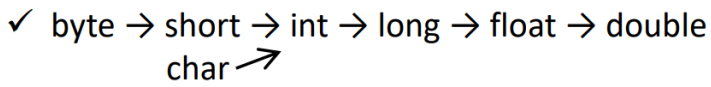
- Литералы (числа, строки)
- Инициализированные final-переменные примитивных типов и типа String, если инициализатор – константа (`final int i = 1;`)
- Операции над константами
- Конкатенация строк
- Скобки
- Условный оператор, если все операнды – константы.

Type cast (приведение типов)

```java
int x = 128;
byte c = (byte) x;
```

Преобразование типов

Расширяющее



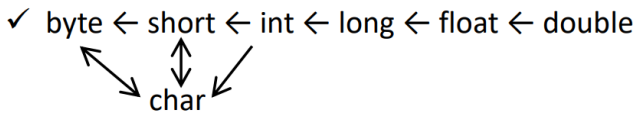

```java
short s = b;
int i = s;
long l = i;
float f = l;
double d = f;
```

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` $\rightarrow$ `int`, остальные не меняются
- Бинарные операции -- есть определённая иерархия:
    1. Любой операнд `double` $\Rightarrow$ другой операнд `double`
    2. Любой операнд `float` $\Rightarrow$ другой операнд `float`
    3. Любой операнд `long` $\Rightarrow$ 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`), компилятор 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);

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);
```

```java

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);
```

23-02-16

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):

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

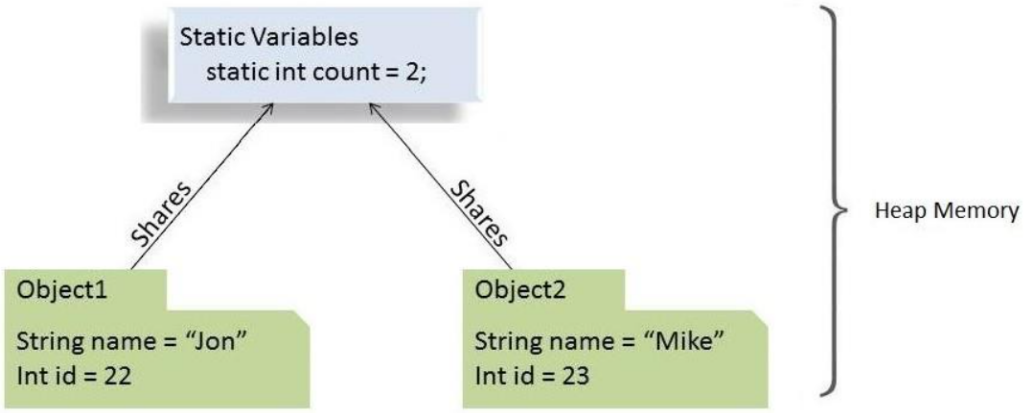
Значения по умолчанию в функциях

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

```
public static void main() {
    class Point {
        int x, y;

        void shift(int dx, int dy) {
            x += dx;
            y += dy;
        }

        void shift(int dx) {
            shift(dx, 0);
        }
    }

    Point p = new Point();
    p.x = 1;
    p.y = 1;
    p.shift(1, 1);
    p.shift(1);
}
```

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`).

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

```
class Point {
    int x = 0; // значения по умолчанию
    int y = 0;

    Point(int x, int y) {
        this.x = x; // `this.` используется, чтобы как-то различать
        this.y = y; // название поля и название аргумента
    }

    // деструкторов в джаве нет
}
```

```
public static void main(String[] args) {
    Point p1 = new Point(1, 2);
    Point p2 = new Point(3, 4);

    // Создали два объекта на куче. В переменных `p1` и `p2`
    // хранятся ссылки на эти объекты.

    p1 = p2;
    // Поменяли ссылку, сохраненную в `p1`.
    // Теперь нет прямого доступа к первому объекту,
    // и сборщик мусора уничтожил первый объект.
}
```

Record

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

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

У records по умолчанию создаются и определяются:

- публичныq коструктор
- 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:

```
public record Person(String name, String address) {
    public Person {  // меняет дефолтный конструктор
        Objects.requireNonNull(name);
        Objects.requireNonNull(address);
    }

    public Person(String name) {
        this(name, "Unknown");
    }
}
```

Статические переменные и методы

```
public record Person(String name, String address) {
    public static String UNKNOWN_ADDRESS = "Unknown";

    public static Person unnamed(String address) {
        return new Person("Unnamed", address);
    }
}
```

Обратиться к ним можно, используя имя record класса:

```
Person.UNKNOWN_ADDRESS;
Person.unnamed("aboba address");
```

`instanceof`

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

```
record Point(int x, int y) {
    void shift(int dx, int dy) {
        x += dx;
        y += dy;
    }
}

public static void process(Object obj) {
    if (obj instanceof Point) {
        Point point = (Point) obj;
        point.shift(1, 1);
    }
}
```

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

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

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 состояие объекта. Нельзя создать экземпляр интерфейса и в интерфейсе нет контрукторов.

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 двухсторонный контракт -- требования к реализации и к использованию. Требования выражаются:

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

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

```
/**
 * Returns a sum of double numbers
 * @param a first number
 * @param b second number
 * @return a result
 */
double sum(double a, double b);
```

- в документации. Простое описание функций и методов.

Использование `interface`

```
public interface Swimmable  {
     public void swim();
}

public class Duck implements Swimmable {
    @Override
    public void swim() {
        System.out.println("Уточка, плыви!");
    }

    public static void main(String[] args) {
        Duck duck = new Duck();
        duck.swim();
    }
}
```

`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.

```
abstract static class AbstractSwimmable extends Swimmable {
    String name;

    @Override
    public void swim() {
        System.out.println(name + ", плыви!");
    }
}
```

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`:

```
sealed interface Swimmable permits Duck {
    ...
}
```

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.

```
interface Vector {
    static Vector of(int x, int y, int z) {
        if (x == 0 && y == 0 && z == 0) {
            return new ZeroVector();
        }
        return new ArrayVector(x, y, z);
    }
}
```

Enum

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

```
enum Weekday {
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY,
    SUNDAY
}
```

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

```
enum Weekday {
    // экземпляры
    MONDAY("MON", false), TUESDAY("TUE", false),
    WEDNESDAY("WED", false), THURSDAY("THU", false),
    FRIDAY("FRI", false), SATURDAY("SAT", true),
    SUNDAY("SUN", true);

    // поля
    private final String shortName;
    private final boolean weekend;

    // конструктор
    Weekday(String shortName, boolean weekend) {
```

```
        this.shortName = shortName;
        this.weekend = weekend;
    }

    // методы
    public String getShortName() { return shortName; }
    public boolean isWeekend() { return weekend; }
}
```

Перечисление

Некоторые полезные методы, сгенерированный компилятором автоматически:

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

switch, используя экземпляры enum-класса

```
String workingHours(Weekday weekday) {
    return switch (weekday) {
        case MONDAY, FRIDAY -> "9:30-13:00";
        case TUESDAY, THURSDAY -> "14:00-17:30";
        case WEDNESDAY, SATURDAY, SUNDAY -> "Выходной";
    };
}

for (Weekday weekday : Weekday.values()) {
    System.out.println(workingHours(weekday))
}
```

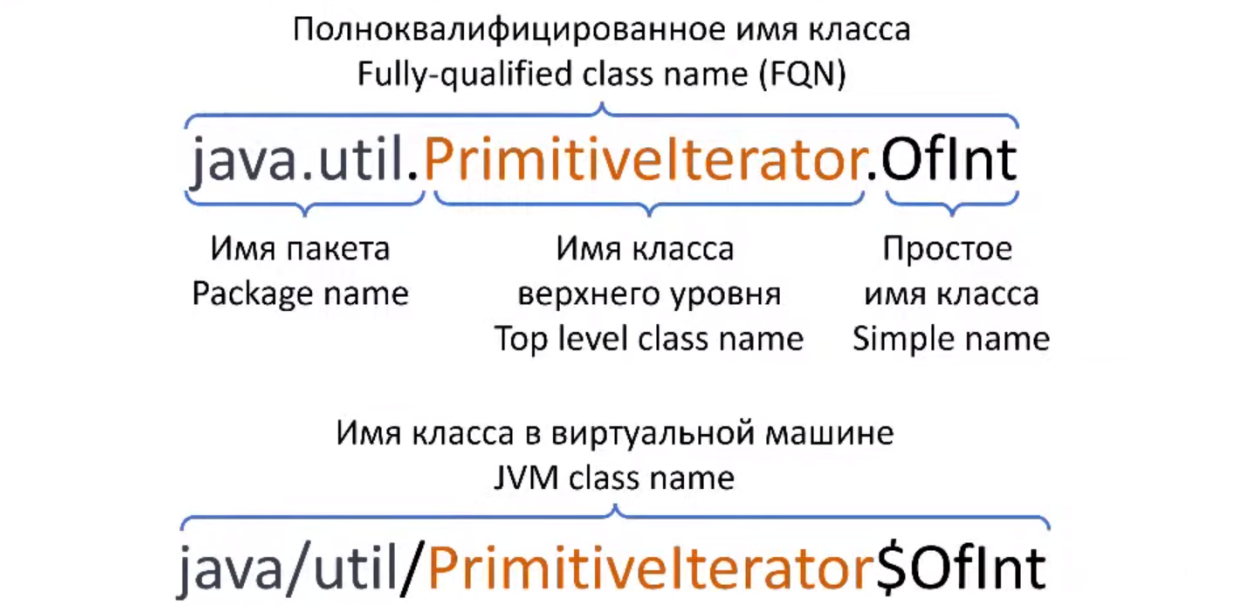
Пакеты

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

```
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.



Модули

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

```
module demo {
    requires java.xml;
    requires java.desktop;
    exports com.example.demo; // экспорт пакета
}
```

Название модуля с маленькой буквы.

23-02-23

Исключения

```java
public int get(int index) {
    if (index < 0) {
        throw new IllegalArgumentException("index < 0");
    }
    return array[index];
}
```

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.

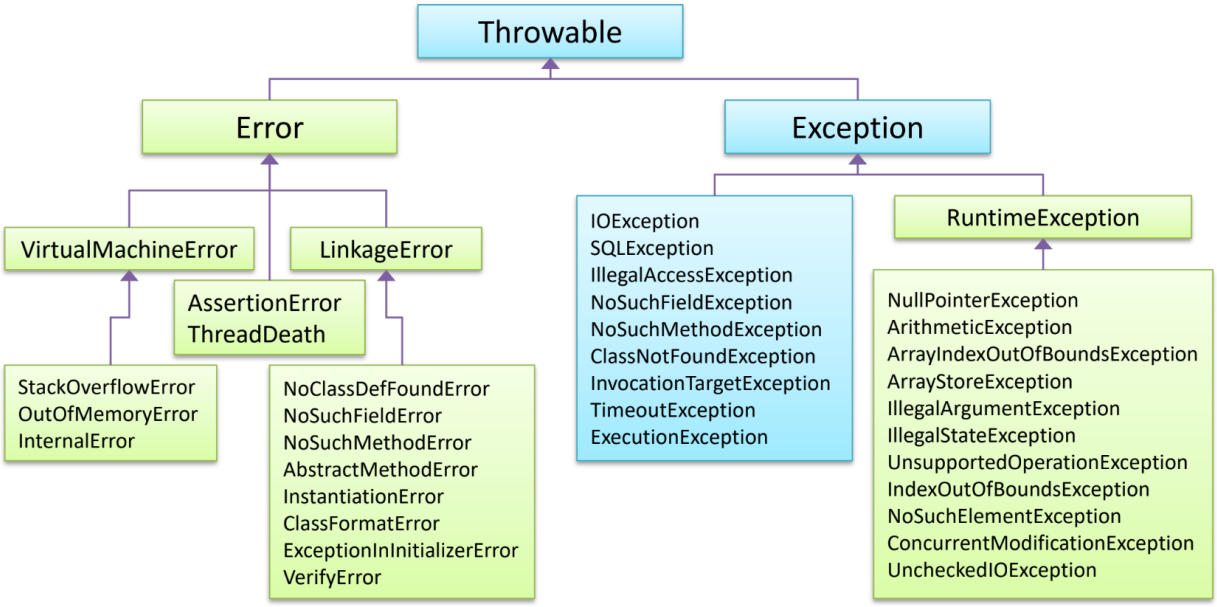
Можно у метода явно указать, что он кидает исключение:

```java
public int get(int index) throws IllegalArgumentException {
    if (index < 0) {
        throw new IllegalArgumentException("index < 0");
    }
    return array[index];
}
```

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

```
Throwable();
Throwable(message);  // exception with message
Throwable(message, cause);  // exception with message and cause. Cause is another exception
                            // which was the reason for throwing currect exception.
```

Полезные методы у исключений

```
Exception exception = new Exception();
exception.getMessage();  // get message of exception
exception.getCause();    // get cause of exception

exception.getStackTrace();  // stack trace is an array of objects. Each object stores name of class,
                            // name of method and number of line where exception was thrown.

exception.getSuppressed(); // if exception B is thrown inside exception A, exception A will be suppressed and saved.
```

try-catch

```
public int get(int index) {
    if (index < 0) {
        throw new IllegalArgumentException("get(index = " + index.toString() +
"): index < 0\n");
    }
    return array[index];
}

public static void main(String[] args) {
    try {
        get(-1);
    }
    catch (IllegalArgumentException ex) {
        System.out.println("message: " + ex.getMessage());
        System.out.println(ex.getStackTrace());
    }
}
```

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

```
public static void main(String[] args) {
    try {
        get(-1);
    }
    catch (IllegalArgumentException ex) {
        System.out.println("message: " + ex.getMessage());
        System.out.println(ex.getStackTrace());
    }
    finally {
        System.out.println("Finishing main\n");
    }
}
```

Еще про `finally`

Пример, как не надо делать:

```java
public static int test() {
    try {
        return 5;
    }
    finally {
        return 6;
    }
}
```

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

```java
public class PoolComposerPrincipalEventIdentifier {
    private static final Logger LOG = Logger.getLogger(
        PoolComposerPrincipalEventIdentifier.class.getName()
    );

    public void identifyPrincipalEvent() {
        try {
            doIdentifyPrincipalEventUsingPoolComposer();
        }
        catch(Exception ex) {
            LOG.log(Level.SEVERE, "Error while identifying principal event",
ex);
        }
    }
}
```

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`

```
static double getDoubleValue(Shmoption<Number> shmopt) {
    return shmopt.get().doubleValue();
}

getDoubleValue(new IntegerShmoption(123));
```

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

```
static double getDoubleValue(Shmoption<? extends Number> shmopt) {
    return shmopt.get().doubleValue();
}

getDoubleValue(new IntegerShmoption(123));
// В данном случае тип переданного аргумента -- Integer.
```

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

Пример использования `super`

```
static void setInteger(Shmoption<Integer> shmopt) {
    shmopt.set(42);
}

NumberShmoption<Number> n = new NumberShmoption<>(123.45);
setInteger(n);
```

Не скомпилится. Функция ожидает класс типа `Integer`.

```java
static void setInteger(Shmoption<? super Integer> shmopt) {
shmopt.set(42);
}
NumberShmoption<Number> n = new NumberShmoption<>(123.45);
setInteger(n);
```

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

Covariant return type

Ковариантность -- это про приведение типов.

```java
interface Supplier {
    Object get();
}

interface StringSupplier extends Supplier {
    @Override
    String get();
}
```

Параметризация

Параметризация типов

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

Пример

```java
static class Option<T> {
    private T value;

    public Option(T value) { this.value = value; }

    /** Never returns null */
    public T get() {
```

```
        if(value == null) throw new NoSuchElementException();
        return value;
    }
    public T orElse(T other) { return value == null ? other : value; }
    public boolean isPresent() { return value != null; }
}

public static void main(String[] args) {
    Option<String> aboba = new Option<String>("aboba");

    Option<String> abobaImplicit = new Option<>("aboba");
    // в джаве 8 придумали оператор ромб:)
    // нет необходимости явно указывать тип в конструкторе

    Option<String> abobaImplicit = new Option("aboba");
    // raw type. Можно не указывать тип и не использовать `<>`.
    // нужно избегать их, как огня
}
```

```
public static void main(String[] args) {
    Option<String> abobaImplicit = new Option<>("aboba");

    Option<var> abobaImplicit = new Option<>("aboba");
    // бывают ситуации, когда компилятор не справляется определить нужный пользователю тип.
    // Например, поставит тип Object вместо String.
    // Поэтому дети, не используйте одновременно `var` и оператор `<>`.
}

```

Маскировочный тип

`?` (wildcard)

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

```
static class Shmoption<T> {
    T value;
    public Shmoption(T value) { this.value = value; }\
    public T get() {
        if(value == null) throw new NoSuchElementException();
        return value;
    }
    public void set(T newValue) { value = newValue; }
    public T orElse(T other) { return value == null ? other : value; }
    public boolean isPresent() { return value != null; }
}

```

```
public static void main(String[] argv) {
    Shmoption<?> present = new Shmoption<>("yes");

    System.out.println(present.isPresent()); // в данном случае не зависит от типа элемента

    Object value = present.get(); // вернет какой-то тип, не известный.
                                  // Но мы точно знаем, что это наследник `Object`

    present.set(???);  // сработает только с null.
                       // Тип у объекта `present` не определен, он так и остается `?`.
}
```

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

`? extends Number`

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

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

```
// наследник типа Number
class NumberShmoption<N extends Number> extends Shmoption<N> {
    public NumberShmoption(N value) { super(value); }
}

// наследник типа Integer
class IntegerShmoption extends NumberShmoption<Integer> {
    public IntegerShmoption(Integer value) { super(value); }
}
```

Параметризация методов

Чаще всего делают статические параметризированные методы.

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

```
static <T> void setNotNull(Shmoption<? super T> shmoption, T value) {
    if (value == null) throw new IllegalArgumentException();
    shmoption.set(value);
}

setNotNull(n, 123);
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 скомпилится не потняно почему.

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

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

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

```
Shmoption<Integer>[] arrayInt = new Shmoption[10]; // raw-type. Получаем предупреждение
Object[] obj = arrayInt;
obj[0] = new Shmoption<>("foo");
Shmoption<Integer> shmoption = arrayInt[0];
Integer x = shmoption.get(); // ClassCastException: в `arrayInt[0]` лежит `String`
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 исплючению `ClassCastException`.

Пример

```
Shmoption<Integer> integer = new Shmoption<>(10);

Shmoption<String> string = ((Shmoption<String>) integer);
// ошибка: `Integer` и `String` в разных ветках иерархии

Shmoption<?> any = integer;
Shmoption<String> string2 = (Shmoption<String>) any;
// скомпилируется, но появится предупреждение,
// потому что в байткоде будет просто `string2 = any`

String s = string2.get();
// ClassCastException: внутри `string2` живет `Integer`, а не `String`

NumberShmoption<Integer> number = (NumberShmoption<Integer>)integer;
// OK
```

23-03-02

`varags`

Переменное число аргументов

```java
static void printAll(Object... objects) {
    for (Object object : objects) {
        System.out.println(object);
    }
}

printAll("a", 1, "b", 2.0);

Object[] objects = {"a", 1, "b", 2.0};
printAll(objects);

printAll(null, null); // OK. An array of nulls
printAll(null); // does not compile.
```

varargs + generics

```java
static <T> boolean isOneOf(T value, T... options) {
    for (T option : options) {
        if (Objects.equals(value, option)) return true;
    }
    return false;
}
```

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

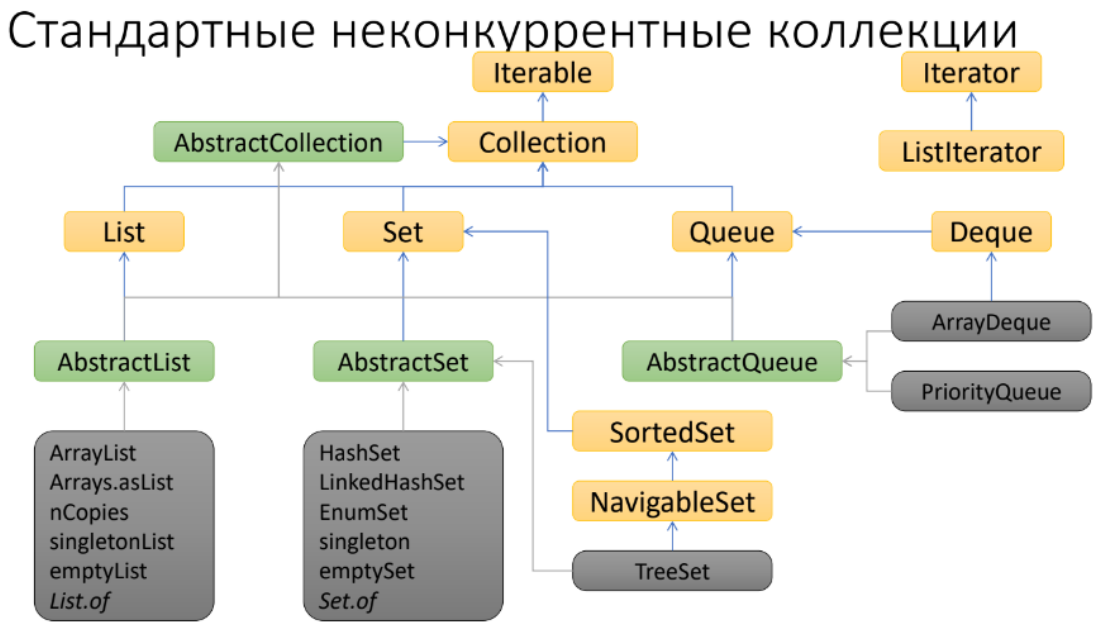
Используя `varargs` и дженерики в методе, можно получить ворнинг типа `unchecked generic array creation for varargs parameter`. Если метод не делает ничего противозаконного, его можно пометить как безопасный с помощью аннотации `@SafeVarargs`:

```java
@SafeVarargs
static <T> boolean isOneOf(T value, T... options) {
    for (T option : options) {
        if (Objects.equals(value, option)) return true;
    }
    return false;
}

isOneOf(shmoption, new Shmoption<>("foo"),
                   new Shmoption<>("bar"));
```

Collections

Иерархия

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



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 }
```

```
Exception in thread "main" java.util.ConcurrentModificationException
    at
java.base/java.util.ArrayList$Itr.checkForComodification(ArrayList.java:1013)
    at java.base/java.util.ArrayList$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):

```java
@FunctionalInterface
interface LongSupplier {
    long getLong(); // sam, since not implemented
}

@FunctionalInterface
interface IntSupplier extends LongSupplier {
    int getInt(); // sam, since not implemented
    @Override
    default long getLong() { return getInt(); }
}
```

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 поливыражения). 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

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

Функциональные 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.

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

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.ptintln("aboba");`
    - блок: `{ System.out.ptintln("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++; // does not actually influence on anything
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.

```
import java.util.ArrayList;
import java.util.List;
import java.util.function.IntSupplier;

public class Demo {
    public static void main(String[] args) {
        int x = 1;
        List<IntSupplier> List = new ArrayList<>();
        for (int i = 0; i < 5; i++) {
            int xx = x++;
            IntSupplier l1 = () -> xx * xx;
            List.add(l1);
        }
        for (IntSupplier elem : List) {
            System.out.println(elem.getAsInt());
        }
    }
}
```

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

Вывод:

```
1
4
9
16
25
```

Optional<T>

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

```
// filter(Predicate<T>)
boolean isFooEqualsBar = Optional.of("foo").filter("bar"::equals).isPresent();

// <U> map(Function<T, U>)
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 {
```

```
    Optional<User> findUser(String name);
}

void increaseUserCarma(UserRepository repository, String name) {
    repository.findUser(name)
        .filter(User::isActive)
        .orElseThrow(() -> new IllegalArgumentException("No such user: "+name))
        .updateCarma(1);
}
```

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 комманд, но сами комманды не выполняют. Команды выполняютя, когда появляется терминальная операция)
- терминальные операции (функции, которые вызываются в конце)

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

Источники

```
Stream.empty();

Stream.of(x, y, z);  // из существующих элементов

// если null, вернется пустой стрим
// если не null, вернется стрим из одного элемента
Stream.ofNullable(x);

// бесконечный стрим
Stream.generate(Math::random);

// следующий элемент зависит от  предыдущего
// val -- начальное значение
```

```java
Stream.iterate(val, x -> x+1);

// то же, но в условием
Stream.iterate(0, x -> x < 100, x -> x + 1);

// любую коллекцию можно превратить в стрим
collection.stream();

// массив можно превратить в стрим
// должен быть тип int, long, double или Object
// с остальными упс
Arrays.stream(array);

// стрим чаров, полученный из строки
// внутри хранятся не символы, а их численные значения
String.chars();

// как ленивый стрим. Хорош для длинных строк
Pattern.splitAsStream();
```

Промежуточные операции

щперации	комментарии
map	Из стрима получается стрим. Есть mapToInt, mapToLong, mapToDouble, mapToObj, предвращают старый тип в новый из названия.
filter	Выкидывает элементы по предикату. Не меняет изначальную сортированность
flatMap	Превращает 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).

щперации	комментарии
dropWhile	Скипаем элементы, *пока* они удовлетворяют условию (предикату).
boxed	Превращает стрим в стрим от `Integer`.
asLongStream	
asDoubleStream	

Пример `peek`

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

```
List<List<String>> listOfLists =
    List.of(List.of("foo", "bar", "baz"),
    List.of("Java", "Kotlin", "Groovy"),
    List.of("Hello", "Good Bye"));

System.out.println(listOfLists.stream()
    .flatMap(List::stream) // стрим стримов -> стрим
    .anyMatch("Java"::equals));
```

Пример, как не надо делать: shuffle

Хотим рандомизировать порядок в стриме. Так плохо:

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

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

`.sorted()` -- передаем компаратор

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

Правильное решение -- через `Collections`:

```
List<Integer> list =
    IntStream.range(1, 50)
        .boxed()
        .collect(Collectors.toList());

Collections.shuffle(list);

list.forEach(System.out::println);
```

Пример, как не надо делать: порядок промежуточных операций

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

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

```
IntStream.iterate(1, x -> x * 2)
    .limit(10)
    .sorted()
    .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`

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

Можно проще:

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

Пример `reduce`

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

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

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 мапы, 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 мапы (`HashTable::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.

```
static Map<Integer, String> stringsByLength(List<String> list) {
    return list.stream().collect(
        Collectors.groupingBy(String::length, Collectors.joining(","))
    );
}

stringsByLength(Arrays.asList("a", "bb", "c", "dd", "eee"));
// {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.

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

Пример `toMap`

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

```java
interface Book {
    String getCategory();
    int getPrice();
}

static Map<String, Integer> getLowestBookPriceByCategory(List<Book> list) {
    return list.stream().collect(Collectors.toMap(
        Book::getCategory,
        Book::getPrice,
        BinaryOperator.minBy(Comparator.naturalOrder())
    ));
}
```

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).

```java
public interface Collector<T, A, R> {
    // Создать накопитель
    Supplier<A> supplier();

    // Добавить элемент в накопитель
    BiConsumer<A, T> accumulator();

    // Склеить два накопителя
    BinaryOperator<A> combiner();

    // Преобразовать накопитель в результат
    Function<A, R> finisher();

    // Флаги (ерунда всякая)
    Set<Characteristics> characteristics();
}
```

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 соптимизировать 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.

```
public static<T, R> Collector<T, R, R> of
    (Supplier<R> supplier,
     BiConsumer<R, T> accumulator,
     BinaryOperator<R> combiner,
     Characteristics... characteristics)
{
    ...
}

public static<T, A, R> Collector<T, A, R> of
    (Supplier<A> supplier,
     BiConsumer<A, T> accumulator,
     BinaryOperator<A> combiner,
     Function<A, R> finisher,
     Characteristics... characteristics)
{
    ...
}
```

Пример с `toList`

```
static <T> Collector<T, ?, List<T>> toList() {
    return Collector.of(
        ArrayList::new,
        List::add,
        (acc1, acc2) -> {
            acc1.addAll(acc2);
            return acc1;
        }
    );
}
```

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:

```
Stream.of("one", "two", "three", "four", "five", "six", "seven", "eight")
    .collect(minMax(Comparator.comparingInt(String::length),
                    (min, max) -> min+"|"+max))
    .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

```
// возвращаемый тип -- коллектор с `Optional<R>`
static <T, R> Collector<T, ?, Optional<R>> minMax
    // `cmp` -- компаратор
    (Comparator<? super T> cmp,
   // `finisher` -- функция, получающая макс и мин и возвращающая их комбинацию
```

```
     BiFunction<? super T, ? super T, ? extends R> finisher)
{
    // накопитель, реализованный как внутренний класс
    // не может быть null
    // функция для сапплайера -- `Acc::new`
    class Acc {
        T min;
        T max;
        boolean present; // =false, если стрим пустой

        void add(T t) {
            if(present) {
                if(cmp.compare(t, min) < 0) min = t;
                if(cmp.compare(t, max) > 0) max = t;
            } else {
                min = max = t;
                present = true;
            }
        }

        // функция для комбайнера
        // используем `cmp`, переданный пользователем
        Acc combine(Acc other) {
            if(!other.present) return this;
            if(!present) return other;
            if(cmp.compare(other.min, min) < 0) min = other.min;
            if(cmp.compare(other.max, max) > 0) max = other.max;
            return this;
        }

        // функция для финишера
        // используем `finisher`, переданный пользователем
        Optional<T> finish(Acc acc) {
            if (acc.present) {
                return Optional.of(finisher.apply(acc.min, acc.max));
            }
            return Optional.empty();
        }
    }

    // возвращаем результат
    return Collector.of(
        Acc::new,
        Acc::add,
        Acc::combine,
        Acc::finish
    );
}
```