

Java 8

Java 7

```
Collections.sort(list,  
    new Comparator<String>() {  
        @Override  
        public int compare(String a, String b) {  
            return b.compareTo(a);  
        }  
    });
```



Java 8

```
Collections.sort(list, (String a, String b)-> {  
    return b.compareTo(a);  
});
```

```
Collections.sort(list, (a, b) -> b.compareTo(a));
```

DON'T KNOW IF JOKE

OR JUST JAVA

quickmeme.com



Интерфейсы

JAVA 8



Статические методы

Ключевое слово static

```
public interface Function<T, R> {  
    R apply(T t);  
    static <T> Function<T, T> identity() {  
        return new Function<T, T>() {  
            public T apply(T t) {  
                return t;  
            }  
        }  
    }  
}
```

Методы по-умолчанию

Ключевое слово default

```
public interface Function<T, R> {  
    default <V> Function<V, R> compose(  
        Function<? super V, ? extends T> before) {  
        return new Function<V, R>() {  
            public R apply(V v) {  
                return apply(before.apply(v));  
            }  
        }  
    }  
}
```

Не доступны для

`equals()`, `hashCode()`, `toString()`

Default methods: multiple inheritance?

```
interface Foo {  
    default int f(int x) { return x + 42; }  
}
```

```
interface Bar {  
    default int g(int x) { return x * x; }  
}
```

```
class FooBar implements Foo, Bar {} // OK
```

Default methods: multiple inheritance?

```
interface Foo {  
    default int f(int x) { return x + 42; }  
}
```

```
interface Bar {  
    default int f(int x) { return x * x; }  
}
```

```
class FooBar implements Foo, Bar {} // FAIL
```

Default methods: multiple inheritance?

```
interface Foo {  
    default int f(int x) { return x + 42; }  
}
```

```
interface Bar {  
    default int f(int x) { return x * x; }  
}
```

```
class FooBar implements Foo, Bar {  
    @Override // is required  
    public int f(int x) { return x * x; }  
}
```

Default methods: multiple inheritance?

```
interface Foo {  
    default int f(int x) { return x + 42; }  
}  
interface Bar {  
    default int f(int x) { return x * x; }  
}  
class FooBar implements Foo, Bar {  
    @Override // even better  
    public int f(int x) {  
        return Bar.super.f(x);  
    }  
}
```

Default methods: multiple inheritance?

```
interface Foo {  
    default int f(int x) { return x + 42; }  
}
```

```
class Bar {  
    int f(int x) { return x * x; }  
}
```

```
class FooBar extends Bar implements Foo {  
    // OK, implementation from Bar wins  
}
```

Функциональные интерфейсы

- Один метод без умолчания
 - *Function<T, R> / R apply(T t)*
 - *BiFunction<T, U, R> / R apply(T t, U u)*
 - *Consumer<T> / void accept(T t)*
 - *Runnable / void run()*
 - *Callable<T> / T call()*
- Аннотация
 - *@FunctionalInterface*
 - Интерфейсы с единственным абстрактным методом называются функциональными и помечаются аннотацией - **Не обязательная**

Лябмда





Лямбда-функции

- Новый компактный синтаксис для инстанцирования функциональных интерфейсов
- `List < User > users = getAllUsers ();`
`Collections.sort(users, (u1, u2 ) ->`
`u1.getName().compareTo(u2.getName()));`
- Компилятор сам выводит типы
- Фигурные скобки не нужны, если внутри одно выражение

Лямбда-функции

- Реализация функционального интерфейса
 - `BiFunction<String, Long, String> f =
 (String s, Long v) -> s + v;`
 - `BiFunction<String, Long, String> f = (s, v) -> s + v;`
 - `Function<String, String> f = s -> s + s;`
 - `Callable<String> f = () -> "!";`
 - `Consumer<String> c = s -> { System.out.println(s); }`

Применение лямбда-функций

- Композиция

```
default <V> Function<V, R>  
    compose(Function<V, T> before) {  
        return (V v) -> apply(before.apply(v));  
    }
```

- Обратная композиция

```
default <V> Function<T, V>  
    andThen(Function<R, V> after) {  
        return (T t) -> after.apply(apply(t));  
    }
```

Применение лямбда-функций

- `Iterable.forEach()`
`List <User> users = getAllUsers ();`
`list.forEach(( u) -> System.out.println (u.getName()));`
- `Collection.removeIf()`
`void removeUserByName ( String userName ) {`
`List <User> users = getAllUsers ();`
`users.removeIf((u)-> u.getName().equals(userName));`
`}`
- `List.replaceAll()`
`List <String> list = getList ();`
`list.replaceAll ((s) -> {`
`StringBuilder sb = new StringBuilder (s );`
`return sb . reverse (). toString ();`
`});`

Ссылки на методы

- Метод класса

```
Function<String, Integer> f1 = Integer::parseInt;  
// x -> Integer.parseInt(x);
```

- Метод экземпляра

```
Function<Integer, Float> f2 = Integer::floatValue;  
// x -> x.floatValue();
```

- Метод экземпляра объекта

```
Integer i = 2;  
Supplier<String> f3 = i::toString; // () -> i.toString();
```

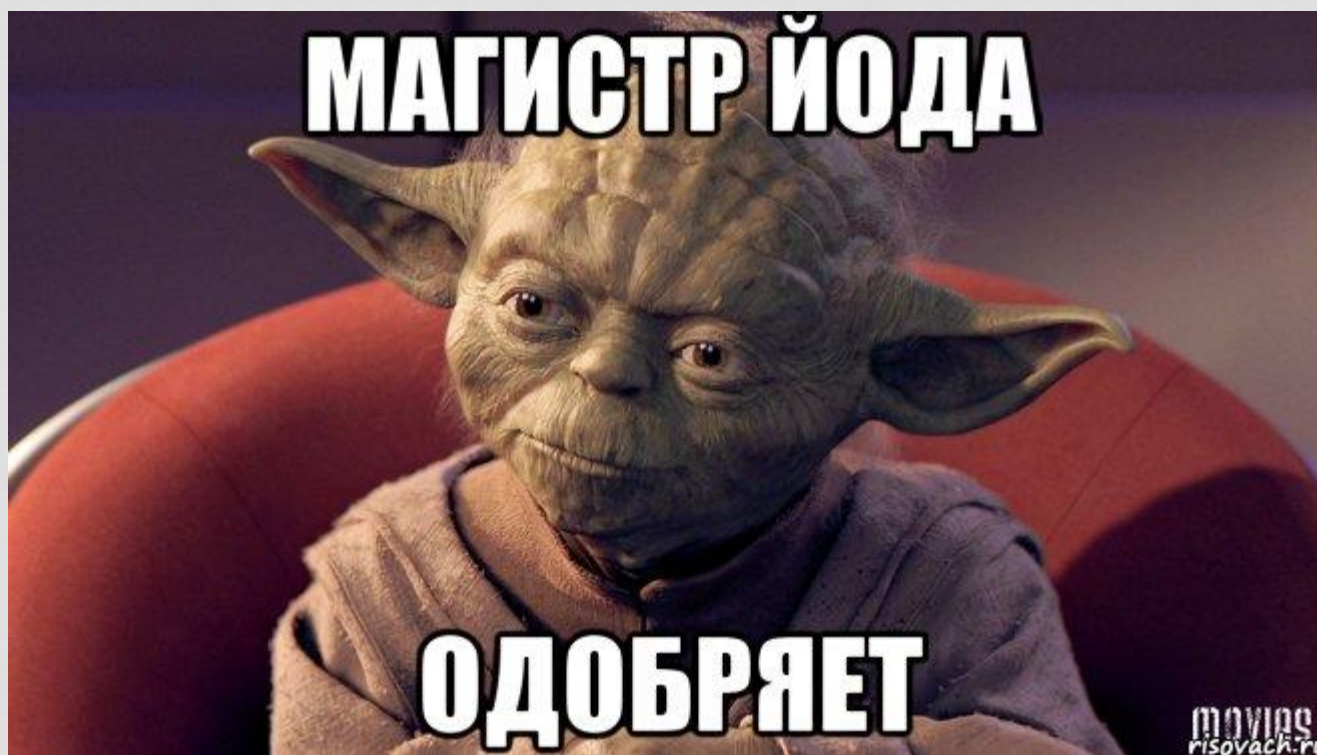
- Конструктор

```
Function<String, Integer> f = Integer::new;  
// s -> new Integer(s);
```

Ссылки на методы

```
public void example () {  
    List <String> names = Arrays.asList ("John", "Frank", "Sam");  
    List <User> users = map (names, User::new );  
}
```

```
public <S, T> List <T> map (List <S> list, Function <S, T> function) {  
    List <T> result = new ArrayList <T>(list. size());  
    list.forEach ((s) -> result.add (function.apply(s)));  
    return result;  
}
```



Замыкания

- Effectively final переменные
 - Ровно одно присваивание
 - Может быть без модификатора final
 - Могут использоваться в лямбда-выражениях
- Замыкание → объект в куче
 - Разные вызовы → разные объекты

Ограничения лямбда-выражений

- Захват не-final переменных
- Прозрачный проброс исключений
// Не работает, так как write бросает IOException
`Consumer<String> c = writer.write(c);`
- Изменение потока управления
`collection.forEach(e -> {
 if (e.equals("done")) {
 // Что делать?
 }
});`
- Реализация классов
Лямбды — не классы

java.util.stream

Примеры использования потоков

- `collection.stream()`
 `.filter(s -> s.endsWith("s"))`
 `.mapToInt(String::length)`
 `.max();`
- `collection.parallelStream()`
 `.filter(s -> s.contains("a"))`
 `.sorted(String.CASE_INSENSITIVE_ORDER)`
 `.limit(3)`
 `.reduce((s1, s2) -> s1 + ", " + s2);`