ВОЩИЛО ЮРИЙ

РАЗРАБОТКА КОРПОРАТИВНЫХ РЕШЕНИЙ С ИСПОЛЬЗОВАНИЕМ ТЕХНОЛОГИЙ JAVA

МЕТОДЫ ИНТЕРФЕЙСОВ ПО УМОЛЧАНИЮ

Java 8 позволяет вам добавлять неабстрактные реализации методов в интерфейс, используя ключевое слово `default`. Эта фича также известна, как методы расширения:

```
package by.part6;

public class Example1 {

  interface Formula {

    double calculate(int a);

    default double sqrt(int a) {
      return Math.sqrt(a);
    }
  }

  public static void main(String[] args) {
    Formula formula = new Formula() {
      @Override
      public double calculate(int a) {
        return sqrt(a * 100);
      }
    };

    System.out.println(formula.calculate(100));     // 100.0
    System.out.println(formula.sqrt(16));               // 4.0

  }
}
```

ЛЯМБДА-ВЫРАЖЕНИЯ

```
package by.part6;

import java.util.Arrays;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

public class Example2 {

  public static void main(String[] args) {
//old style
    List<String> names = Arrays.asList("peter", "anna", "mike", "xenia");

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

//java8 style 1)
    Collections.sort(names, (String a, String b) -> {
      return b.compareTo(a);
    });

//java8 style 2)

    Collections.sort(names, (String a, String b) -> b.compareTo(a));

//java8 style 3)
    Collections.sort(names, (a, b) -> b.compareTo(a));
  }
}
```

ФУНКЦИОНАЛЬНЫЕ ИНТЕРФЕЙСЫ

Как лямбда-выражения соответствуют системе типов языка Java? Каждой лямбде соответствует тип, представленный интерфейсом. Так называемый *функциональный интерфейс* должен содержать **ровно один абстрактный метод**. Каждое лямбда-выражение этого типа будет сопоставлено объявленному методу. Также, поскольку методы по умолчанию не являются абстрактными, вы можете добавлять в функциональный интерфейс сколько угодно таких методов.

Мы можем использовать какие угодно интерфейсы для лямбда-выражений, содержащие ровно один абстрактный метод. Для того, чтобы гарантировать, что ваш интерфейс отвечает этому требованию, используется аннотация `@FunctionalInterface`. Компилятор осведомлен об этой аннотации, и выдаст ошибку компиляции, если вы добавите второй абстрактный метод в функциональный интерфейс

ФУНКЦИОНАЛЬНЫЕ ИНТЕРФЕЙСЫ

```java
package by.part6;

public class Example3 {

  @FunctionalInterface
  interface Converter<F, T> {

    T convert(F from);
  }

  public static void main(String[] args) {
    Converter<String, Integer> converter = (from) -> Integer.valueOf(from);
    Integer converted = converter.convert("123");
    System.out.println(converted);    // 123
  }
}
```

ССЫЛКИ НА МЕТОДЫ И КОНСТРУКТОРЫ

```
package by.part6;

public class Example4 {

  @FunctionalInterface
  interface Converter<F, T> {
    T convert(F from);
  }

  static class Something {
    String startsWith(String s) {
      return String.valueOf(s.charAt(0));
    }
  }

  static class Person {
    String firstName;
    String lastName;

    Person() {
    }

    Person(String firstName, String lastName) {
      this.firstName = firstName;
      this.lastName = lastName;
    }
  }

  interface PersonFactory<P extends Person> {
    P create(String firstName, String lastName);
  }

    public static void main(String[] args) {
      //Ссылки на методы интерфейса
      Converter<String, Integer> converter = Integer::valueOf;
      Integer converted = converter.convert("123");
      System.out.println(converted);    // 123

      //Ссылки на методы класса
      Something something = new Something();
      Converter<String, String> converter2 =
  something::startsWith;
      String converted2 = converter2.convert("Java");
      System.out.println(converted2);    // "J"

      //Ссылки на конструктор
      PersonFactory<Person> personFactory = Person::new;
      Person person = personFactory.create("Peter", "Parker");
      System.out.println(person.firstName + " " +
  person.lastName);
    }
  }
```

ОБЛАСТИ ДЕЙСТВИЯ ЛЯМБД

Доступ к переменным внешней области действия из лямбда-выражения очень схож к доступу из анонимных объектов. Вы можете ссылаться на переменные, объявленные как `final`, на экземплярные поля класса и статические переменные.

```
package by.part6;

import by.part6.Example4.Converter;

public class Example5 {

  public static void main(String[] args) {
    final int num = 1;
    Converter<Integer, String> stringConverter = (from) -> String.valueOf(from + num);

    stringConverter.convert(2);     // 3
  }
}
```

ПРЕДИКАТЫ

Предикаты – это функции, принимающие один аргумент, и возвращающие значение типа boolean. Интерфейс содержит различные методы по умолчанию, позволяющие строить сложные условия (`and`, `or`, `negate`).

```
package by.part6;

import java.util.Objects;
import java.util.function.Predicate;

public class Example6 {

  public static void main(String[] args) {
    Predicate<String> predicate = (s) -> s.length() > 0;

    System.out.println(predicate.test("foo"));              // true
    System.out.println(predicate.negate().test("foo"));     // false

    Predicate<Boolean> nonNull = Objects::nonNull;
    Predicate<Boolean> isNull = Objects::isNull;

    Predicate<String> isEmpty = String::isEmpty;
    Predicate<String> isNotEmpty = isEmpty.negate();

  }
}
```

ФУНКЦИИ, ПОСТАВЩИКИ И ПОТРЕБИТЕЛИ

Функции принимают один аргумент и возвращают некоторый результат. Методы по умолчанию могут использоваться для построения цепочек вызовов (compose, andThen).

Поставщики (suppliers) предоставляют результат заданного типа. В отличии от функций, поставщики не принимают аргументов.

Потребители (consumers) представляют собой операции, которые производятся на одним входным аргументом.

Компараторы хорошо известны по предыдущим версиям Java. Java 8 добавляет в интерфейс различные методы по умолчанию.

ФУНКЦИИ, ПОСТАВЩИКИ И ПОТРЕБИТЕЛИ

```
package by.part6;

import by.part6.Example4.Person;
import java.util.function.Consumer;
import java.util.function.Function;
import java.util.function.Supplier;

public class Example7 {

  public static void main(String[] args) {
//Функции
    Function<String, Integer> toInteger = Integer::valueOf;
    Function<String, String> backToString = toInteger.andThen(String::valueOf);
    System.out.println(backToString.apply("123"));     // "123"

//Поставщики
    Supplier<Person> personSupplier = Person::new;
    Person person = personSupplier.get();// new Person
    person.print();

//Потребители
    Consumer<Person> greeter = (p) -> System.out.println("Hello, " + p.firstName);
    greeter.accept(new Person("Luke", "Skywalker"));

  }
}
```

ОПЦИОНАЛЬНЫЕ ЗНАЧЕНИЕ

Опциональные значение – это по сути контейнер для значения, которое может быть равно null. Например, вам нужен метод, который возвращает какое-то значение, но иногда он должен возвращать пустое значение. Вместо того, чтобы возвращать null, в Java 8 вы можете вернуть опциональное значение.

```
package by.part6;

import java.util.Optional;

public class Example8 {

  public static void main(String[] args) {
    Optional<String> optional = Optional.of("bam");

    optional.isPresent();           // true
    optional.get();                 // "bam"
    optional.orElse("fallback");    // "bam"

    optional.ifPresent((s) -> System.out.println(s.charAt(0)));     // "b"
  }
}
```

ПОТОКИ

Тип java.util.Stream представляет собой последовательность элементов, над которой можно производить различные операции. Операции над потоками бывают или промежуточными (intermediate) или конечными (terminal).

Конечные операции возвращают результат определенного типа, а промежуточные операции возвращают тот же поток.

Таким образом вы можете строить цепочки из несколько операций над одним и тем же потоком.

Поток создаются на основе источников, например типов, реализующих java.util.Collection, такие как списки или множества (ассоциативные массивы не поддерживаются). Операции над потоками могут выполняться как последовательно, так и параллельно.

ПОТОКИ

```
package by.part6;

import java.util.Arrays;
import java.util.List;
import java.util.stream.Stream;

public class Example9 {

  public static void main(String[] args) {
    List<String> stringCollection = Arrays.asList(
        "ddd2",
        "aaa2",
        "bbb1",
        "aaa1",
        "bbb3",
        "ccc",
        "bbb2",
        "ddd1"
    );

    Stream<String> stream = stringCollection.stream();
    Stream<String> stream1 = Stream.of("ddd2",
        "aaa2",
        "bbb1",
        "aaa1",
        "bbb3",
        "ccc",
        "bbb2",
        "ddd1"
    );
  }
}
```

FILTER

Операция Filter принимает предикат, который фильтрует все элементы потока. Эта операция является промежуточной, т.е. позволяет нам вызвать другую операцию (например, forEach) над результатом. ForEach принимает функцию, которая вызывается для каждого элемента в (уже отфильтрованном) поток. ForEach является конечной операцией. Она не возращает никакого значения, поэтому дальнейший вызов потоковых операций невозможен.

SORTED

Операция Sorted является промежуточной операцией, которая возвращает отсортированное представление потока. Элементы сортируются в обычном порядке, если вы не предоставили свой компаратор.

Помните, что sorted создает всего лишь отсортированное представление и не влияет на порядок элементов в исходной коллекции. Порядок строк в stringCollection остается нетронутым:

MAP

Промежуточная операция map преобразовывает каждый элемент в другой объект при помощи переданной функции. Вы можете использовать map для преобразования каждого объекта в объект другого типа. Тип результирующего потока зависит от типа функции, которую вы передаете при вызове map.

MATCH

Для проверки, удовлетворяет ли поток заданному предикату, используются различные операции сопоставления (match). Все операции сопоставления являются конечными и возвращают результат типа boolean.

COUNT

Операция Count является конечной операцией и возвращает количество элементов в потоке. Типом возвращаемого значения является long.

АССОЦИАТИВНЫЕ МАССИВЫ

putIfAbsent позволяет нам не писать дополнительные проверки на null; forEach принимает потребителя, который производит операцию над каждым элементом массива.

```
package by.part6;

import java.util.HashMap;
import java.util.Map;

public class Example11 {

  public static void main(String[] args) {
    Map<Integer, String> map = new HashMap<>();

    for (int i = 0; i < 10; i++) {
      map.putIfAbsent(i, "val" + i);
    }

    map.forEach((id, val) -> System.out.println(val));
  }
}
```

АССОЦИАТИВНЫЕ МАССИВЫ

код показывает как использовать для вычислений код при помощи различных функций:

```
map.computeIfPresent(3, (num, val) -> val + num);
map.get(3);             // val33

map.computeIfPresent(9, (num, val) -> null);
map.containsKey(9);     // false

map.computeIfAbsent(23, num -> "val" + num);
map.containsKey(23);    // true

map.computeIfAbsent(3, num -> "bam");
map.get(3);             // val33
```

АССОЦИАТИВНЫЕ МАССИВЫ

как удалить объект по ключу, только если этот объект ассоциирован с ключом:

```
map.remove(3, "val3");
map.get(3);             // val33
map.remove(3, "val33");
map.get(3);             // null
System.out.println(map);
```

еще один полезный метод:

```
map.getOrDefault(42, "not found");  // not found
```

ОБЪЕДИНИТЬ ЗАПИСИ ДВУХ МАССИВОВ

```
map.merge(9, "val9", (value, newValue) -> value.concat(newValue));
map.get(9);              // val9

map.merge(9, "concat", (value, newValue) -> value.concat(newValue));
map.get(9);              // val9concat
```