ФУНКЦИОНАЛНО ПРОГРАМИРАНЕ

ЛЕКЦИОНЕН КУРС "ПРОГРАМИРАНЕ НА JAVA"

Distributed @learning center

ЛАМБДАС

- Ламбда – нова езикова конструкция, въведена в Java 8
- Използването ѝ изисква по-различен начин на мислене
- Води към нов (за Java) стил на програмиране, познат като “функционално програмиране“
- С нейна помощ, някои решения могат да бъдат формулирани по един елегантен начин
- Особено предоставят предимства в областта на:
  - Рамки – напр. колекции (Collections-Framework)
  - Паралелна обработка

ТИПИЧНА СОРТИРОВКА

- Сортиране имена по дължина
- Показване на дължината на имената, разделение с „,“

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

public class SortWithComparator {
    public static void main(String[] args) {
        final List<String> names = Arrays.asList(“Ангел", “Михаил", “Ани", “Владимир");
        Collections.sort(names, new Comparator<String>() {
            @Override
            public int compare(final String str1, final String str2) {
                return Integer.compare(str1.length(), str2.length());
            }
        });

        final Iterator<String> it = names.iterator();
        while (it.hasNext()) {
            System.out.println(it.next().length() + ", ");
        }
    }
```

Сортира списъка съответно зададения Comparator

```
3,
5,
6,
8,
```

Не може ли да се реализира с по-малко код и по-просто?

ОПРЕДЕЛЕНИЕ

- Ламбда: контейнер на първичен код, подобен на един метод, обаче без:
  - Име
  - Явно задаване на тип на резултата
  - Възможни изключения
- Т.е. анонимен метод със следния синтаксис:

(списък параметри) → { израз или оператори }

ПРИМЕРИ

1. `( int x, int y ) → { return x + y; }`
2. `( long x ) → { return x * 2; }`
3. `( ) → { String msg = “Lambda”; System.out.println(“Hello” + msg); }`

ЛАМБДА В СИСТЕМАТА ОТ ТИПОВЕ

- Как можем да използваме и извикваме ламбда изрази
- Да се опитаме да присвоим на ламбда java.lang.Object референция – до Java 8 възможно за всички типове

```
Object x = ( ) → { System.out.println("Hello Lambda"; }
```

- Грешка при компилиране: Object тук не е функционален интерфейс

ФУНКЦИОНАЛНИ ИНТЕРФЕЙСИ

Функционален интерфейс: нов тип, въведен в Java 8

- Интерфейс с точно един абстрактен метод
- Наричат се също SAM (Simple Abstract Method) тип

Бележка: такива интерфейси съществуваха също преди Java 8, но нямаха специално означение

ПРИМЕРИ

1

```
@FunctionalInterface
public interface Runnable {
        public abstract void run();
}
```

Анотация, въведена в Java 8

? Защо е функционален интерфейс?

2

```
@FunctionalInterface
public interface Comparator<T> {
        int compare (T o1, T o2);
        boolean equals (Object obj);
}
```

В Java Language Specification (JLS) всички методи в интерфейсите автоматично са public и abstract, независимо че не е явно зададено

Всички методи, дефинирани в Object могат да се задават допълнително в един функционален интерфейс

ИМПЛЕМЕНТАЦИЯ

- Обикновено, SAM-типовете (функционални интерфейси) се реализират като вътрешни класове
- В Java 8 – като алтернатива могат да се използват ламбдас
  - ✓ Предпоставка: ламбда може да изпълни абстрактния метод на функционалния интерфейс
  - ✓ Т.е. съвместимост между: брой и тип на параметрите, тип на резултата

Схема (абстрактен модел):

Обичайно (вътрешен клас)

```
new SAMTypeAnonymousClass() {
        public void samTypeMethod (Method-Parameters) {
                Method-Body
        }
}
```

Ламбда

```
(Method-Parameters) → { Method-Body }
```

ПРИМЕР: LISTENER

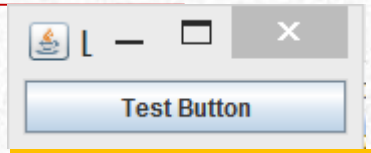
```
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
public class ListenerTest   {
  public static void main(String[] args)   {

    JButton testButton = new JButton("Test Button");

    testButton.addActionListener( new ActionListener(){
        @Override
         public void actionPerformed(ActionEvent ae)   {
            System.out.println("Click Detected by Anon Class");
         }
    });

    testButton.addActionListener(e -> System.out.println("Click Detected by Lambda Listner"));

    JFrame frame = new JFrame("Listener Test");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.add(testButton, BorderLayout.CENTER);
    frame.pack();
    frame.setVisible(true);
```



Click Detected by Lambda Listener
Click Detected by Anon Class

Като вътрешен клас

Като ламбда

Минимални размери за да се покажат всички елементи

ПРИМЕР: RUNNABLE

```
public class RunnableTest {
    public static void main(String[] args) {
        System.out.println("=== RunnableTest ===");

        // Anonymous Runnable
        Runnable r1 = new Runnable(){
        @Override
         public void run() {
                System.out.println("Hello world one!");
         }

    };

    // Lambda Runnable
    Runnable r2 = () → System.out.println("Hello world two!");

    r1.run();
    r2.run();
  }
}
```

```
=== RunnableTest ===
Hello world one!
Hello world two!
```

@OVERRIDE АНОТАЦИЯ

- @Override анотацията се използва за информиране препокриване на метод в подклас - обикновено не се указва явно, понеже не е задължително (програмите работят коректно и без тази анотация)
- Добра практика - да се използва понеже:
  - ✓ Ако програмист направи някаква грешка при предефинирането на меттода, като напр. грешно име на метод, грешни типове параметри, ще получим грешка по време на компилация. С помощта на тази анотация инструктираме компилатора, че препокриваме метод. Ако не използвате анотацията тогава метода на подкласа ще се държат като нов метод (не препокрития метод
  - ✓ Подобрява читаемостта на кода. Така че, ако променим сигнатурата на препокрития метод след това всички подкласове, които препокриват този метод ще предизвикат грешка по време на компилация, която в крайна сметка ще ни помогне да променим сигнатурата в подкласовете. Ако имаме много класове в приложението, тогава анотацията наистина ще ни помогне да определим класовете, които изискват промени, когато променим сигнатурата на метод.

ПРИМЕР: COMPARATOR

```
Comparator<String> compareByLength = new Comparator<String>() {
    @Override
    public int compare(final String str1, final String str2) {
        final int length1 = str1.length();
        final int length2 = str2.length();
        if (length1 < length2)
            return -1;
        if (length1 > length2)
            return 1;
        return 0;
    }
};
```

Интерфейс Comparator:

- Comparator реализира сравнение на два обекта от тип T
- За целта трябва да се реализира по подобяващ начин абстрактният метод int compare(T, T)

Обичайно (вътрешен клас)

```
Comparator<String> compareBylength = new Comparator<String>() {
    @Override
    public int compare(final String str1, final String str2) {
        return Integer.compare(str1.length(), str2.length());
    }
};
```

- Класът Integer беше разширен с метода compare(int, int)
- Така използването му значително по-компактно

Java 7 (Integer.compare)

```
Comparator<String> compareBylength = (final String str1, final String str2) → {
    return Integer.compare(str1.length(), str2.length());
};
```

Lava 8 (lambda)

ОСОБЕНОСТИ НА СИНТАКСИСА

- Тип интерфейс – можем да спестим задаването на тип на параметрите, където компилаторът разбира от контекста на използване
  - Подобен на оператора „<>“ (диамант)
  - За съкращаване на кода на програмите
- Други съкращения – ако кодът е израз тогава могат да отпаднат също “{” и “}”, както и return
- Освен по-компактен запис има нещо по-значимо:
  - Ламбдас могат да се използват по-гъвкаво, като строго типизирани методи
- Общ принцип: всичко, което може да се изведе трябва да отпадне от синтаксиса

ПРИМЕРИ

```
Comparator<String> compareByLength = (str1, str2) → {
        return Integer.compare(str1.length(), str2.length());
}
```

```
(int x, int y) → { return x + y };
(long x) → { return x * 2 };

(x, y) → x + y;
x → x * 2;
```

Дадените изчисления могат да се прилагат навсякъде, където за параметрите са дефинирани операциите „+“ и „*“

ЛАМБДАС КАТО ПАРАМЕТРИ НА МЕТОДИ И ВРЪЩАНИ СТОЙНОСТИ

- Досега за ламбдас:
  - Могат да се използват вместо анонимни вътрешни класове за имплементиране на SAM типове
- Освен това:
  - Могат да се използват като параметри и като връщани като резултат стойности

ПРИМЕР ОТ НАЧАЛОТО НА ЛЕКЦИЯТА

- Сортиране имена по дължина
- Показване на дължината на имената, разделение с „,“

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

public static void main(String[] args) {
      final List<String> names = Arrays.asList(“Ангел", “Михаил", “Ани", “Владимир");

      Collections.sort(names,  (str1, str2) → Integer.compare(str1.length(), str2.length()));

       names.sort(compareByLength());
}

public static Comparator<String> compareByLength() {
      return (str1, str2) → Integer.compare(str1.length(), str2.length());
}
```

Ламбда като параметър на метод

Ламбда като връщана стойност

3,
5,
6,
8,

ПРИМЕР ОТ НАЧАЛОТО НА ЛЕКЦИЯТА

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

public static void main(String[] args) {
      final List<String> names = Arrays.asList(“Ангел", “Михаил", “Ани", “Владимир");

      Collections.sort(names,  (str1, str2) → Integer.compare(str1.length(), str2.length()));

       names.sort(compareByLength());
}

public static Comparator<String> compareByLength() {
      return (str1, str2) → Integer.compare(str1.length(), str2.length());
}
```

ПРИМЕР ОТ НАЧАЛОТО НА ЛЕКЦИЯТА

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

public static void main(String[] args) {
      final List<String> names = Arrays.asList("Ангел", "Михаил", "Ани", "Владимир");

      Collections.sort(names,  (str1, str2) → Integer.compare(str1.length(), str2.length()));

       names.sort(compareByLength());
}

public st
      retu
}
```

- При този вариант не се използва Collections.sort
- Директно се извиква сортировка върху обект от тип List<String>
- Как работи това?
  - ✓ До Java 7 методът не се съдържа в java.util.List<T>
  - ✓ В Java 8 този интерфейс (и много други) бяха разширени

DEFAULT МЕТОДИ

- При разработване на ламбдас и интеграцията им в JDK 8 се установява, че за смисленото им използване
  - Съществуващи класове и интерфейс също трябва да бъдат разширени
- До Java 8 разширяването на интерфейс не беше възможно, без това да има ефект върху използващите ги класове
- Разширяването на един интерфейс винаги води до несъвместимост
  - Когато един метод се добави към един интерфейс, той трябва да бъде реализиран във всички класове, имплементиращи интерфейса

РАЗШИРЕНИЕ НА ИНТЕРФЕЙСИ

- За справяне с тази дилема в Java 8 е възможно
  - В първичния код на един интерфейс да се даде така наречена имплементация по подразбиране (default implementation)
- За тази цел се използва ново свойство на езика, наречено методи по подразбиране
  - Специални имплементации на методи, които могат да бъдат дефинирани в интерфейса
  - За да се различават от обикновените методи те са означени с ключовата дума default
- Ще разгледаме два примера за разширения
  - Сортировка
  - Итерация

МЕТОД SORT()

- Методът е включен в интерфейса List<E>
- Директно се прилага върху обекти от този тип

```
public interface List<E> extends Collection<E>    {
    default void sort(Comparator<? super E> c)  {
        Collections.sort(this, c);
    }
}
```

МЕТОД FOREACH()

- Итерация върху всички елементи на колекция и обработка на всеки отделен елемент
- Използва default метода forEach(Consumer<? Super T> action) от интерфейса java.lang.Iterabale<T>, който е основа за java.util.Collection<E> и List<E>
- В него е деклариран абстрактният метод accept(), имплементирането на който доставя функционалността, изпълнявана върху елементите

```
public interface Iterable<T>  {
    default void forEach(Consumer<? super T> action)  {
       Objects.requireNonNull(action);
        for ( T t : this )  {
           action.accept(t);
        }
    }
}
```

Проверява дали дадената референция към обект не е null

```
@FunctionalInterface
public interface Consumer<T>  {
    void accept( T t );

    default Consumer<T> andThen(Consumer<? super T> after)  {
      Objects.requireNonNull(after);
       return ( T t ) → { accept(t); after.accept(t); }
    }
}
```

CONSUMER

Consumer е функционален интерфейс, който „консумира“ един обект – взема обекта и работи с него по някакъв начин, без да връща резултат

```
import java.util.function.Consumer;
public class ConsumerTest {
  public static void main(String[] args) {
    Consumer<String> c = (x) → System.out.println(x.toLowerCase());
    c.accept("JAva2s.COM");
  }
}
```

java2s.com

```
import java.util.function.Consumer;
public class ConsumerTest1 {
    public static void main(String[] args) {
        Consumer<String> consumer = ConsumerTest1::printNames;
         consumer.accept("Иван");
         consumer.accept("Ани");
         consumer.accept("Димитър");
    }
     private static void printNames(String name) {
        System.out.println(name);
    }
}
```

Иван
Ани
Димитър

СОРТИРАНЕ С CONSUMER

```
import java.util.Arrays;
import java.util.List;

public class SortWithConsumer {
    public static void main(String[] args) {
        final List<String> names = Arrays.asList("Ангел", "Михаил", "Ани", "Владимир");
        names.sort( (str1, str2) → Integer.compare(str1.length(), str2.length()) );
        names.forEach(it → System.out.println(it.length() + ", "));
    }
}
```

```
3,
5,
6,
8,
```

РЕФЕРЕНЦИИ НА МЕТОДИ

- В Java 8, посредством референциите на методи може да бъде подобрена четаемостта на кода
- Синтаксис – клас::име_на_метод
- Реферира:
  - ✓ Метод – напр., System.out::println, Person::getName
  - ✓ Конструктор – напр., ArrayLiast::new, Person[]::new
- Референциите на методи могат да се използват вместо ламбдас за опростяване на записа

```
import java.util.Arrays;
import java.util.List;

public class MethodRef {
    public static void main(String[] args) {
        final List<String> names = Arrays.asList("Ангел", "Михаил", "Ани", "Владимир");
        names.forEach(it → System.out.println(it.length() + ", "));
        names.forEach(System.out::println);
    }
}
```

```
4,
7,
3,
6,
Ангел
Михаил
Ани
Владимир
```

МАСОВИ ОПЕРАЦИИ С КОЛЕКЦИИ

- Ламбдас могат да се използват за формулиране на алгоритми за обработка на колекции от данни
- В Java 8 са въведени два варианта на масови операции (bulk operations) върху колекции:
  - Разширение на съществуващи интерфейси на колекции – напр., List<E>
  - Стриймове
- Като въведение ще разгледаме два варианта на итерация:
  - Явна (външна)
  - Неявна (вътрешна)

ЯВЕН ИТЕРАТОР

- Колекциите поддържат траверса посредством индексиран достъп или един java.util.Iterator<E>
- Процесът се контролира от извикващия код

```
import java.util.Arrays;
import java.util.Iterator;
import java.util.List;

public class ExternIteration {
  public static void main(String[] args) {
     final List<String> names = Arrays.asList(“Ангел", “Михаил", “Ани", “Владимир");
     final Iterator<String> it = names.iterator();
     while (it.hasNext()) {
        System.out.println(it.next());
      }
     for (int i = 0; i < names.size(); i++) {
        System.out.println(names.get(i));
     }
     for (final String name : names) {
        System.out.println(name);
     }
  }
}
```

НЕЯВЕН ИТЕРАТОР

- Процесът е капсулиран и се реализира в класа на колекцията - детайлите на реализацията са скрити и възможностите за влияние са ограничени (до Java 8 обикновено се използваха for или while цикли)
- Итерациите не имплементират от програмиста, а се реализират в рамката – ние предаваме само операцията, която трябва да се извърши

```
import java.util.Arrays;
import java.util.Iterator;
import java.util.List;

public class InternIteration {
  public static void main(String[] args) {
     final List<String> names = Arrays.asList("Ангел", "Михаил", "Ани", "Владимир");
     names.forEach((String name) → { System.out.println(name); });
     names.forEach(name → System.out.println(name));
     names.forEach(System.out::println);
   }
}
```

РАЗШИРЕНИЯ НА КОЛЕКЦИИ

- Ще разгледаме три съществени примера на разширение на колекции:
  - Интерфейс java.util.function.Pradicate<T> - позволява формулиране на предикати
    - Това са логически условия, които се оценяват посредством метода на интерфейса boolean test(T)
  - Метод Collection.removeIf – изтриване на елементи, удовлетворяващи някакво условие
  - Интерфейс UnaryOperator<T> - преобразува един тип T в друг тип R

ИНТЕРФЕЙС PREDICATE<T>

```
@Functional Interface
public interface Predicate<T> {
        boolean test(T t);
        default Predicate<T> and(Predicate<? Super T> other) { …};
        default Predicate<T> negate( ) { …};
        default Predicate<T> or(Predicate<? Super T> other) { …};
}
```

Някои от методите на интерфейса

```
import collections.Person;
import java.util.function.Predicate;

public class FirstPredicatesExample {
   public static void main(final String[] args) {
        final Predicate<String> isNull = str → str == null;
        final Predicate<String> isEmpty = String::isEmpty;
        final Predicate<Person> isAdult = person → person.getAge() >= 18;

        System.out.println("isNull(''):     " + isNull.test(""));
        System.out.println("isEmpty(''):    " + isEmpty.test(""));
        System.out.println("isEmpty('Nia'): " + isEmpty.test("Nia"));
        System.out.println("isAdult(Nia):   " + isAdult.test(new Person("Nia", "Plovdiv",
23)));
    }
}
```

```
isNull(''):     false
isEmpty(''):    true
isEmpty('Nia'): false
isAdult(Nia):   true
```

ПРИМЕР: REMOVEIF()

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

public class RemoveIfExample {
    public static void main(final String[] args) {
        final List<String> names = createDemoNames();

        names.removeIf(String::isEmpty);
        System.out.println(names);
    }

    private static List<String> createDemoNames() {
        final List<String> names = new ArrayList<>();
        names.add("Ангел");
        names.add("");
        names.add("Ани");
        names.add("Михаил");
        names.add("   ");
        names.add("Владимир");
        return names;
    }
}
```

[Ангел, Ани, Михаил, , Владимир]

ПРИМЕР: UNARYOPERATOR()

```
import java.util.function.UnaryOperator;
public class UnaryOperatorExample {
  public static void main(final String[] args) {
     final UnaryOperator<String> markTextWithM = str → str.startsWith("M") ?
                                    ">>" + str.toUpperCase() + "<<" : str;
    printResult("Mark 1", "unchanged", markTextWithM);
    printResult("Mark 2", "Michael", markTextWithM);
    final UnaryOperator<String> trimmer = String::trim;
    printResult("Trim 1", "no_trim", trimmer);
    printResult("Trim 2", "   trim me   ", trimmer);
    final UnaryOperator<String> mapNullToEmpty = str → str == null ? "" : str;
    printResult("Map same", "same", mapNullToEmpty);
    printResult("Map null", null, mapNullToEmpty);
  }
 private static void printResult(final String text, final String value, final UnaryOperator<String> op) {
    System.out.println(text + ": '" + value + "' → '" + op.apply(value) + "'");
 }
}
```

маркира низове, започващи с “M”

премахва празни символи

присвоява на null желана стойност (тук “”)

apply(): преобразува типовете

ПРИМЕР: UNARYOPERATOR()

```
import java.util.function.UnaryOperator;
public class UnaryOperatorExample {
  public static void main(final String[] args) {
     final UnaryOperator<String> markTextWithM = str → str.startsWith("M") ?
                                    ">>" + str.toUpperCase() + "<<" : str;
    printResult("Mark 1", "unchanged", markTextWithM);
    printResult("Mark 2", "Michael", markTextWithM);
    final UnaryOperator<String> trimmer = String::trim;
    printResult("Trim 1", "no_trim", trimmer);
    printResult("Trim 2", "   trim me   ", trimmer);
    final UnaryOperator<String> mapNullToEmpty = str → str == null ? "" : str;
    printResult("Map same", "same", mapNullToEmpty);
    printResult("Map null", null, mapNullToEmpty);
  }
 private static void printResult(final String text, final String value, final UnaryOperator<String> op) {
    System.out.println(text + ": '" + value + "' → '" + op.apply(value) + "'");
 }
}
```

```
Mark 1: 'unchanged' -> 'unchanged'
Mark 2: 'Michael' -> '>>MICHAEL<<'
Trim 1: 'no_trim' -> 'no_trim'
Trim 2: '   trim me   ' -> 'trim me'
Map same: 'same' -> 'same'
Map null: 'null' -> ''
```

СТРИЙМОВЕ

- Нова концепция в Java 8
  - Ключова роля интерфейс java.util.stream.Stream<T>
- Стрийм: абстракция за последователност от действия за обработка на данни
  - Подобни на колекциите и итераторите, но не съхраняват данни и могат да бъдат траверсирани само веднъж
  - Аналогия също: pipeline
- Три типа операции:
  - Create – създаване на стриймове
  - Intermediate – изчисления
  - Terminal – подготовка на резултата

ПРИМЕР

```
List<Person> adults = persons.stream().          //Create
                      filter(Person::isAdult).   //Intermediate
                      collect(Collections.toList()); //Terminal
```

СЪЗДАВАНЕ НА СТРИЙМОВЕ

- Две възможности:
  - Стриймове за масиви и колекции
    - Използва се метода stream()
    - Съществуват последователен и паралелен вариант
  - Стриймове за предварително дефинирани области от стойности
    - Използват се методите of(), range(), chars()

ПРИМЕР

```
final String[] namesData = { “Иван”, “Мария”, “Ана” }
final List<String> names = Array.asList(namesData);

final Stream<String> streamFromArray = Array.stream(namesData);
final Stream<String> streamFromList = names.stream;
```

```
final Stream<String> sequentialStream = names.stream();
final Stream<String> parallelStream = names.parallelStream;
```

ПРИМЕР

```
final Stream<String>  names = Stream.of(“Иван”, “Мария”, “Ана”);
final Stream<Integer> integres = Stream.of(1, 4, 7, 9, 7, 2);

final IntStream  values = IntStream.range(0, 100);
final IntStream  chars = “This is a stream”.chars();
```

ОСОБЕНОСТИ

- Съществуват варианти на стриймове за примитивни типове
  - Причини:
    - Обработката на примитивни типове данни намира широко приложение
    - Ускорява обработката в сравнение с Wrapper типовете
- IntStream, LongStream, DoubleStream
- Освен изчислителни и конвертиращи операции, съществуват възможности за превръщане във:
  - Wrapper обекти - boxed()
  - Произволни обекти - mapToObj()

ПРИМЕР:СТРИЙМОВЕ ЗА ПРИМИТИВНИ ТИПОВЕ

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

public class PrimitivesStreamExample {
    public static void main(final String[] args) {
        final List<String> names = Arrays.asList("Мери", "Стефан", "Владимир");
        Stream<String> values = names.stream().
                IntStream  → mapToInt(String::length).
                             asLongStream()
            Stream<Long>  → boxed().
                             mapToDouble(x -> x * .75).
         Stream<String>   → mapToObj(val -> "Val: " + val);

        values.forEach(System.out::println);
    }
}
```

Stream<String> ←

LongStream ←

DoubleStream ←

```
Val: 3.0
Val: 4.5
Val: 6.0
```

ОБРАБОТКА НА СТРИЙМОВЕ

- Обичайни приложения, използващи стриймове:
  - Филтриране
  - Трансформации
  - Сортировки
- За тях се използват intermediate операции – описват обработващи стъпки, които се извършват последователно
- Два вида оператори:
  - Без състояние – за всеки елемент, независимо от другите, се извършва операцията
    - Напр., филтриране
    - Добри за паралелна обработка
  - Със състояние – обработката изисква знание за другите елементи
    - Напр. сортировка

БЕЗ СЪСТОЯНИЯ

Метод	Функция
filter( )	Филтрира елементите от стрийма, които не удовлетворяват предадения Predicate<T>
map( )	Трансформира елементи с помощта на Function<T,R> - от тип T към R
flatmap( )	Превръща вградени стриймове в един плосък
peek( )	Изпълнява едно действие за всеки елемент – полезен метод за тестване

СЪС СЪСТОЯНИЯ

Метод	Функция
distinct( )	Премахва дубликати – спрямо equals(Object)
sorted( )	Сортира елементите на основата на Comparator<T>
limit( )	Ограничава максимален брой елементи за стрийма
skip( )	Прескача първите n елемента на стрийма

ПРИМЕР: ФИЛТРИРАНЕ

```
import java.util.stream.*;
import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;

public class FilterExample {
    public static void main(final String[] args) {
        final List<Person> persons = new ArrayList<>();
        persons.add(new Person("Иван", 31));
        persons.add(new Person("Мария", 28));
        persons.add(new Person("Ния", 3));

        final Predicate<Person> isAdult = person → person.getAge() >= 18;
        final Stream<Person> adults = persons.stream().filter(isAdult);

        adults.forEach(System.out::println);
    }
}
```

```
public final class Person {
    private String name;
    private int   age;
    public Person(final String name, final int age)  {
        this.name = name;
       this.age = age;
   }
   public final String getName()  {  return name; }
    public void setName(String name)   { this.name = name; }
    public int getAge()  { return age; }
    public void setAge(int age)   {  this.age = age; }
}
```

? Защо така?

```
streams.Person@2ff4acd0
streams.Person@54bedef2
```

ПРИМЕР: ФИЛТРИРАНЕ

```
import java.util.stream.*;
import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;

public class FilterExample {
    public static void main(final String[] args) {
        final List<Person> persons = new ArrayList<>();
        persons.add(new Person("Micha", 43));
        persons.add(new Person("Barbara", 40));
        persons.add(new Person("Yannis", 5));

        final Predicate<Person> isAdult = person → person.getAge() >= 18;
        final Stream<Person> adults = persons.stream().filter(isAdult);

        adults.forEach(System.out::println);
    }
}
```

```
public final class Person {
   private String name;
   private int   age;
   public Person(final String name, final int age)  {
       this.name = name;
      this.age = age;
  }
  public final String getName()  {  return name; }
   public void setName(String name)   { this.name = name; }
   public int getAge()  { return age; }
   public void setAge(int age)   {  this.age = age; }

   public String toString()   {  return name; }
}
```

Иван
Мария

ПРИМЕР: ФИЛТРИРАНЕ

```
import java.util.stream.*;
import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;

public class FilterExample {
    public static void main(final String[] args) {
        final List<Person> persons = new ArrayList<>();
        persons.add(new Person("Micha", 43));
        persons.add(new Person("Barbara", 40));
        persons.add(new Person("Yannis", 5));

        //final Predicate<Person> isAdult = person -> person.getAge() >= 18;
        final Stream<Person> adults = persons.stream().filter(Person::isAdult);

        adults.forEach(System.out::println);
    }
}
```

```
public final class Person {
    private String name;
    private int   age;
    public Person(final String name, final int age)  {
        this.name = name;
       this.age = age;
   }
   public final String getName()  {  return name; }
    public void setName(String name)   { this.name = name; }
    public int getAge()  { return age; }
    public void setAge(int age)   {  this.age = age; }

    public boolean isAdult() { return age >= 18; }

    public String toString()   {  return name; }
}
```

Иван
Мария

ПРИМЕР: ИЗВЛИЧАНЕ НА ЕЛЕМЕНТИ

```
import java.util.stream.*;
import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;
public class AttributeExtractionExample {
   public static void main(final String[] args) {
        final List<Person> persons = new ArrayList<>();
        persons.add(new Person("Мария", 28));
        persons.add(new Person("Ния", 3));

        final Stream<Person> adults = persons.stream().filter(Person::isAdult);
        final Stream<String> namesStream = adults.map(person → person.getName());

        final Stream<Integer> agesStream = persons.stream().map(Person::getAge).
                                                          filter(age -> age >= 18);

        namesStream.forEach(System.out::println);
        agesStream.forEach(System.out::println);
    }
}
```

```
Мария
28
```

ПРИМЕР: СОРТИРАНЕ И ПРЕМАХВАНЕ НА ДУБЛИКАТИ

```
import java.util.stream.*;
import java.util.List;
public class SortedAndDistinctExample {
    public static void main(final String[] args) {
        final Stream<Integer> distinct = createIntStream().distinct();
        final Stream<Integer> sorted = createIntStream().sorted();
        final Stream<Integer> sortedAndDistinct = createIntStream().sorted().
                                                                    distinct();
        printResult("distinct:          ", distinct);
        printResult("sorted:            ", sorted);
        printResult("sortedAndDistinct: ", sortedAndDistinct);
    }

    private static Stream<Integer> createIntStream() {
        return Stream.of(7, 1, 4, 3, 7, 2, 6, 5, 7, 9, 8);
    }

    private static void printResult(final String hint, final Stream<Integer> stream) {
        final List<Integer> result = stream.collect(Collectors.toList());
        System.out.println(hint + result);
    }
}
```

```
distinct:          [7, 1, 4, 3, 2, 6, 5, 9, 8]
sorted:            [1, 2, 3, 4, 5, 6, 7, 7, 7, 8, 9]
sortedAndDistinct: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

ЗАВЪРШВАЩИ ОПЕРАЦИИ

- Все някога резултатите от изчисленията в стриймовете трябва да се обобщят и да се изведат върхуопределено устройство или да се предадат за последваща обработка
- За тази цел се използват завършващите операции

ПРИМЕР

Един обект от този клас се използва за генериране на стрийм от (псевдо) случайни числа

```
import java.util.Arrays;
import java.util.stream.*;
import java.util.Random;
import java.util.function.Supplier;

public class StreamToArrayExample {
    public static void main(String[] args) {
        final Random random = new Random();
        final Supplier<Float> randomSupplier = () → random.nextFloat() * 100;

        final Object[] randomNumbers = Stream.generate(randomSupplier).
        limit(7).toArray();
        System.out.println(Arrays.toString(randomNumbers));
        System.out.println("Element type: " + randomNumbers[0].getClass());

        final int[] intRandoms = Stream.generate(randomSupplier).
        limit(7).mapToInt(val → val.intValue()).toArray();

        System.out.println(Arrays.toString(intRandoms));
    }
}
```

КЛАС SUPPLIER

- Supplier е срещуположен на Consumer
- Не приема аргументи, но връща някаква стойност посредством извикване на метода get()

```
import java.util.function.Supplier;

public class SupplierTest {

  public static void main(String[] args) {
    Supplier<String> i  = ()-> "java2s.com";

    System.out.println(i.get());
  }
}
```

java2s.com

ПРИМЕР

```
import java.util.Arrays;
import java.util.stream.*;
import java.util.Random;
import java.util.function.Supplier;

public class StreamToArrayExample {
    public static void main(String[] args) {
        final Random random = new Random();
        final Supplier<Float> randomSupplier = () -> random.nextFloat() * 100;

        final Object[] randomNumbers = Stream.generate(randomSupplier).
        limit(7).toArray();
        System.out.println(Arrays.toString(randomNumbers));
        System.out.println("Element type: " + randomNumbers[0].getClass());

        final int[] intRandoms = Stream.generate(randomSupplier).
        limit(7).mapToInt(val -> val.intValue()).toArray();

        System.out.println(Arrays.toString(intRandoms));
    }
}
```

```
[58.110195, 72.41586, 52.62925, 40.51487, 11.887455, 74.29938, 68.15032]
Element type: class java.lang.Float
[24, 75, 71, 79, 35, 0, 59]
```

БЛАГОДАРЯ ЗА ВНИМАНИЕТО!

КРАЙ “ФУНКЦИОНАЛНО ПРОГРАМИРАНЕ”

Distributed
@learning center