Advanced Java Programming

Programming Technologies

2016/2017 spring

Kollár, Lajos

Kocsis, Gergely (English version)

Advanced Java Programming

- Java 7
  - Strings in switch
  - try-with-resources
- Java 8
  - Default methods of interfaces
  - Lambda-expressions
  - Functional interfaces
  - Streams
  - Others

String in switch

See more:

http://www.theserverside.com/tutorial/The-Switch-to-Java-7-Whats-New-with-Conditional-Switches

http://docs.oracle.com/javase/tutorial/java/nutsandbolts/switch.html

```
public static String getDayType(String dayOfWeek) {
     String dayType;
     if (dayOfWeek == null)
         throw new IllegalArgumentException("No such day: " + dayOfWeek);
     switch (dayOfWeek) {
         case „monday":
             dayType = "Start of the week";
             break;
         case „Tuesday":
         case „Wednesday":
         case „Thursday":
             dayType = "Middle of the week";
             break;
         case „Friday":
             dayType = "End of the week";
             break;
         case „Saturday":
         case „Sunday":
             dayType = "Weekend";
             break;
         default:
             throw new IllegalArgumentException ("No such day: " + dayOfWeek);
     }
     return dayType;
}
```

Strings in switch

- More easy to read that nested if-else statements
  - The generated byte code is also more efficient
- Strings are case-sensitive
- Strings are compared by the `equals` method
  - Avoid `NullPointerException`
- Case labels are constant expressions
- The static type of the selector has to be `String`
  - It is not enough if it is dynamically a `String`

try–with–resources

See more:

http://tutorials.jenkov.com/java-exception-handling/try-with-resources.html

http://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html

Resource handling before…

Where can an exception be thrown?

```java
private static void printFile() throws IOException {
  InputStream input = null;

  try {
    input = new FileInputStream("file.txt");
    int data = input.read();
    while (data != -1) {
      System.out.print((char) data);
      data = input.read();
    }
  } finally {
    if (input != null)
      input.close();
  }
}
```

This has to go to another try

…and now (since Java 7)

```
private static void printFileJava7() throws IOException {

  try (FileInputStream input = new FileInputStream("file.txt")) {
    int data = input.read();
    while (data != -1) {
      System.out.print((char) data);
      data = input.read();
    }
  }
}
```

try–with–resources

- Makes sure that the used resources in the statement are closed automatically
- The resource has to implement the `java.lang.AutoCloseable` interface
- More resources can be used:

```
private static void printFileJava7() throws IOException {
    try (FileInputStream input = new FileInputStream("file.txt");
         BufferedInputStream bufferedInput = new BufferedInputStream(input)) {
        int data = bufferedInput.read();
        while (data != -1) {
            System.out.print((char) data);
            data = bufferedInput.read();
        }
    }
}
```

Default methods in interfaces

Problems with interfaces

- Evolution of already existing interfaces
  - What happens if we need to add a new method to an interface?
    - E.g.. Add `stream(),parallelStream()` and. `forEach()` methods to the collection interfaces
  - We have to implement the new method in all the implementing classes
    - If we do not do the classes will not even compile
    - Especially problematic in case of 3rd party applications

Problems with interfaces

- Not enough flexible design
  - We can distribute functionalities between classes through abstract classes
    - Only one superclass is allowed this restricts the use
  - In many cases so called adapter classes are needed in order not to have to implement all the methods in all the classes e.g:
    - SAX-fprocessing: `ContentHandler` etc. interfaces vs. `DefaultHandler`
    - Swing: `MouseListener` etc. interfaces vs. `MouseAdapter` class
  - These classes are of good technical use but make the code much harder to read, and also hide which exact interfaces are implemented

Solution: default methods

- Alternative names: defender methods, virtual extension methods
- Default methods cannot have methods with signatures matching to non final methods of the Object class
  - Implies runtime exception
- Default methods are given in the interfaces, but can be overridden in the implementing classes
  - No problem in the implementing classes in case of evolution
  - No need to use adapter classes
- Compatible with classes depending on old interfaces
- Required tools:
  - **`default`** keyword
  - A block containing the default implementation
- Drawback: implementation appears in the interface

Example

```java
public interface Addressable {
  String getStreet();

  String getCity();

  default String getFullAddress() {
    return getStreet() + ", " + getCity();
  }
}
```

```
123 AnyStreet, AnyCity
```

```java
public class Letter implements Addressable {
  private String street, city;

  public Letter(String street, String city) {
    this.street = street; this.city = city;
  }

  @Override
  public String getCity() {
    return city;
  }

  @Override
  public String getStreet() {
    return street;
  }

  @Override
  public String getFullAddress() {
    return city + ", " + street;
  }

  public static void main(String[] args) {
    Letter l = new Letter("123 AnyStreet", "AnyCity");
    System.out.println(l.getFullAddress());
  }
}
```

Possible options when inheriting default methods from interfaces

- Do not mention the method
  - The subinterface inherits the method as it is
- Redeclare the method
  - This makes it `abstract`
- Redefine the
  - We give a new body to it.

Lambda-expressions

See more:

http://javarevisited.blogspot.hu/2014/02/10-example-of-lambda-expressions-in-java8.html

http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

http://www.slideshare.net/langer4711/lambdas-funct-progjdays2013

http://www.slideshare.net/martyhall/lambda-expressions-and-streams-in-java-8-presented-by-marty-hall-to-google-inc

http://www.drdobbs.com/jvm/lambda-expressions-in-java-8/240166764

Antecedents – Nested class

- Static nested class
- Inner class
  - Local class
  - Anonymous class

See more: https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html
https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html

Nested class

- A class declared inside another
  - All four visibilities are can be used(public, protected, package private, private)
    - Ordinary classes can only be public or private
  - a member of the enclosing class
- Static
  - Terminology: static nested class
  - Cannot reach the members of the enclosing class
- Non static
  - Terminology: inner class
  - Can reach the members of the enclosing class even if they are private

Advantage of nested classes

- We can group logically our classes that we use only in one place
  - If a class is useful for only one other class it is natural that it it better to store them together.
- Increases closure
  - If classes A and B are both top-level classes B cannot reach private members of A. However if we nest B to A, B can reach these members and moreover B gets also hidden.
- The code gets easier to be read and maintain

Functional interfaces

See more:

http://winterbe.com/posts/2014/03/16/java-8-tutorial/

Functional Interface

- In other name: SAM (Single Abstract Method) interface
- All lambda equals to a type given by an interface
- A functional interface declares <u>explicitly one</u> abstract method
- Any such interface can be used as a lambda expression
    - Mark these with the **`@FunctionalInterface`** annotation
    - As a result the compiler throws compile exception if the interface is not functional

Example

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

```java
Converter<String, Integer> converter = (from) -> Integer.valueOf(from);
Integer converted = converter.convert("123");
System.out.println(converted);    // 123
```

Lambda-expressions

- We can refer to an instance of a class implementing a functional interface.

```
new SomeInterface(){
  @Override
  public SomeType someMethod(parameters) {
    body
  }
}
```

Instead we can use:

```
(parameters) -> {body}
```

Lambda-expressions

```
p -> {
       return p.getGender() == Person.Sex.MALE
              && p.getAge() >= 18
              && p.getAge() <= 25;
     }
```

- Formal parameters between parentheses separated by commas
  - The type of the parameter is optional
  - If there is only one parameter the () are also optional
  - If there are no parameters however () is mandatory
- -> (arrow) token
- A body holding only one expression or block
  - If it is an expression it gets evaluated
  - **`return`** can also be used
    - Since this is not an expression we put it into a block
  - void methods do not have to be put to a block:

    ```
    email -> System.out.println(email)
    ```

Lambda expressions with multiple formal parameters

```
public class Calculator {
  interface IntegerMath {
    int operation(int a, int b);
  }

  public int operateBinary(int a, int b, IntegerMath op) {
    return op.operation(a, b);
  }

  public static void main(String... args) {
    Calculator myApp = new Calculator();
    IntegerMath addition = (a, b) -> a + b;
    IntegerMath subtraction = (a, b) -> a - b;
    System.out.println("40 + 2 = " + myApp.operateBinary(40, 2, addition));
    System.out.println("20 - 10 = " + myApp.operateBinary(20, 10, subtraction));
  }
}
```

Visibility

- From the point of visibility lambda expressions do not introduce a new level („lexically scoped”)
  - No hole in the scope
- Declarations inside are handled as if they were in the enclosing environment
- They cannot change the value of a local variable
  - final or effective final

Method- and constructor references

- We can refer to a method without actually calling it
  - Instantiation and creating arrays work also like this
- `::`
- Examples:

```
String::length             // instance method
System::currentTimeMillis  // static method
List<String>::size         // explicit generic type parameter
List::size                 // inferred type parameter
System.out::println
"abc"::length
super::toString
ArrayList::new
int[]::new
```

Example of method reference

```
public class Something {

  public String startsWith(String s) {
    return String.valueOf(s.charAt(0));
  }

  public static void main(String[] args) {
    Something something = new Something();
    Converter<String, String> converter = something::startsWith;
    String converted = converter.convert("Java");
    System.out.println(converted); // "J"
  }
}
```

Example of constructor reference

- Given the below class with two constructors:

```
class Person {
    String firstName, lastName;

    Person() {}

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

- Create a Factory interface:

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

- The compiler automatically selects the right constructor based on the signature of create:

```
PersonFactory<Person> personFactory = Person::new;
Person person = personFactory.create("Peter", "Parker");
```

Built-in functional interfaces

- Many old interfaces are now annotated with `@FunctionalInterface` annotation, e.g.: Runnable, Comparator
- New ones are introduced (java.util.function package):
  - Predicate
  - Function
  - Supplier
  - Consumer

Predicate

- Predicates are one-parameter logical functions
- Functional method: `test(…)` – evaluates the predicate with the given parameter
  Default methods: `and(…)`, `or(…)`, `negate()`

```
Predicate<String> predicate = (s) -> s.length() > 0;

predicate.test("foo");              // true
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();
```

Function

- One-parameter functions that produce a value
- Functional method: `apply(…)` –applies the function to the parameter
- Default methods:
  - `compose(before)`: applies first the „before” funtion given as a parameter then itself
  - `andThen(after)`: applies itself first and then the „after” function

```
Function<String, Integer> toInteger = Integer::valueOf;

Function<String, String> backToString =
toInteger.andThen(String::valueOf);

backToString.apply("123");     // "123"
```

Supplier

- Describes a supplier object
- Functional method: `get()` – supplies the result

```
Supplier<Person> personSupplier = Person::new;
personSupplier.get();    // new Person
```

Consumer

- One-parameter operation without a result
- It has a side effect (the others do not have)
- Functional method: `accept(…)` – Executes the operation to its parameter
- Default method:
  - `andThen(after)`: Provides a Consumer by also executing the „after” after the execution of this

```
Consumer<Person> greeter = (p) -> System.out.println("Hello, " +
p.firstName);
```

```
greeter.accept(new Person("Luke", "Skywalker"));
```

Streams

See more:

http://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/

http://www.oracle.com/technetwork/articles/java/ma14-java-se-8-streams-2177646.html

http://www.slideshare.net/martyhall/lambda-expressions-and-streams-in-java-8-presented-by-marty-hall-to-google-inc

http://java67.blogspot.hu/2014/04/java-8-stream-api-examples-filter-map.html

What are streams?

- Streams are „monad”s
  - In functional programming *monad* means such a structure that represents calculations given as sequences of steps. A monad-structured type defines what it means to link or encapsulate functions.
- A stream represents a (infinite) list of elements and provides possible operations on these elements
- They are not Collections since they use no memory space.
- The operations may be intermediate or terminal
  - Fluent API: stream operations return streams
- Most of them can get a lambda expression as a parameter
  - This defines the exact work of the operation

Intermediate operations

- They return a stream, so they can be linked one after the other without using semicolon.
- E.g:
  - `filter`: provides a new stream by applying a filter on an old one
  - `map`: provides a new stream by mapping the elements off the old one
  - `sorted`: provides a new stream by sorting the elements of the old one
- Full list: http://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html
- They form an operation pipeline

```
List<String> myList = Arrays.asList("a1", "a2", "b1", "c2", "c1");
myList.stream()
   .filter(s -> s.startsWith("c"))
   .map(String::toUpperCase)
   .sorted()
   .forEach(System.out::println);
```

Terminal operations

- These are `void`, operations or they have a non stream return value
- E.g.: `forEach` or `reduce`

```
List<String> myList = Arrays.asList("a1", "a2", "b1", "c2", "c1");
myList.stream()
      .filter(s -> s.startsWith("c"))
      .map(String::toUpperCase)               //.map(s -> String.toUpperCase(s))
      .sorted()
      .forEach(System.out::println);        //.forEach(s -> System.out.println(s))
```

Result:

```
C1
C2
```

Desired properties of operations

- Non-interfering operation (no side effect): the operation does not modify the data source
  - E.g.: no lambda expression changes `myList` (on previous slide)

- Stateless operation: The execution order of operations is deterministic
  - E.g.: No lambda expression depends on such a variable (state) that may change during execution

Types of streams

- Streams can be created based on different data sources (mainly on Collections)
- New methods of `java.util.Collection`:
  - `stream()`: creates a sequential stream
  - `parallelStream()`: creates a parallel stream

Creating sequential streams

- **`stream()`**
  - Returns a stream of objects
- **`of()`**
  - Creates a stream of object references

```
Stream.of("a1", "a2", "a3")
          .findFirst()
          .ifPresent(System.out::println);
// a1
```

- **`range()`**
  - Initializes a stream of primitive type elements (`IntStream`, `LongStream`, `DoubleStream`, …)

```
IntStream.range(1, 4)
         .forEach(System.out::println);
// 1
// 2
// 3
```

Primitive streams

- Similar to ordinary streams except that…
  - They use specialized lambda expressions
    - E.g. `IntFunction` instead of `Function`, `IntPredicate` instead of `Predicate`…
  - They have more terminal aggregating operations
    - `sum()` and `average()`

```
Arrays.stream(new int[] {1, 2, 3})
    .map(n -> 2 * n + 1)
    .average()
    .ifPresent(System.out::println);  // 5.0
```

- Conversion in both directions (`mapToInt`, `mapToLong`, `mapToDouble`)

```
Stream.of("a1", "a2", "a3")
.map(s -> s.substring(1))
.mapToInt(Integer::parseInt)
.max()
.ifPresent(System.out::println);
// 3
```

```
IntStream.range(1, 4)
.mapToObj(i -> "a" + i)
.forEach(System.out::println);

// a1
// a2
// a3
```

```
Stream.of(1.0, 2.0, 3.0)
.mapToInt(Double::intValue)
.mapToObj(i -> "a" + i)
.forEach(System.out::println);
// a1
// a2
// a3
```

Process order

- Intermediate operations use „lazy processing” meaning that they are processed only if there is a terminal operation
- Example that makes nothing:

```
Stream.of("d2", "a2", "b1", "b3", "c")
  .filter(s -> {
      System.out.println("filter: " + s);
      return true;
  });
```

- Plus a terminal operation:

```
Stream.of("d2", "a2", "b1", "b3", "c")
  .filter(s -> {
      System.out.println("filter: " + s);
      return true;
  })
  .forEach(s -> System.out.println("forEach: " + s));
```

- result:

```
filter: d2
forEach: d2
filter: a2
forEach: a2
filter: b1
forEach: b1
filter: b3
forEach: b3
filter: c
forEach: c
```

Execution is not horizontal, but vertical!

Execution is not horizontal, but vertical!

- This decreases the number of operations to be run on elements

```java
Stream.of("d2", "a2", "b1", "b3", "c")
  .map(s -> {
      System.out.println("map: " + s);
      return s.toUpperCase();
    })
  .anyMatch(s -> {
      System.out.println("anyMatch: " + s);
      return s.startsWith("A");
    });

// map: d2
// anyMatch: D2
// map: a2
// anyMatch: A2
```

`anyMatch` returns true as soon as its predicate can be applied to an input element (2nd element in the example).

As a result of vertical processing the `map` runs only twice.

Does the order matter?

```java
Stream.of("d2", "a2", "b1", "b3", "c")
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    })
    .filter(s -> {
        System.out.println("filter: " + s);
        return s.startsWith("A");
    })
  .forEach(s -> System.out.println("forEach: "+s));

// map:     d2
// filter:  D2
// map:     a2
// filter:  A2
// forEach: A2
// map:     b1
// filter:  B1
// map:     b3
// filter:  B3
// map:     c
// filter:  C
```

The order does matter!

```java
Stream.of("d2", "a2", "b1", "b3", "c")
    .filter(s -> {
        System.out.println("filter: " + s);
        return s.startsWith("a");
    })
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    })
   .forEach(s -> System.out.println("forEach: " + s));

// filter:  d2
// filter:  a2
// map:     a2
// forEach: A2
// filter:  b1
// filter:  b3
// filter:  c
```

sorted

- Stateful intermediate operation
  - In order to be able to sort a list of elements it has to store a state of elements

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .sorted((s1, s2) -> {
        System.out.printf("sort: %s; %s\n", s1, s2);
        return s1.compareTo(s2);
    })
    .filter(s -> {
        System.out.println("filter: " + s);
        return s.startsWith("a");
    })
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    })
    .forEach(s -> System.out.println("forEach: " + s));
```

```
// sort:    a2; d2
// sort:    b1; a2
// sort:    b1; d2
// sort:    b1; a2
// sort:    b3; b1
// sort:    b3; d2
// sort:    c; b3
// sort:    c; d2
// filter:  a2
// map:     a2
// forEach: A2
// filter:  b1
// filter:  b3
// filter:  c
// filter:  d2
```

Here `sorted` runs horizontally (for the full list)

Updated version:

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .filter(s -> {
        System.out.println("filter: " + s);
        return s.startsWith("a");
    })
    .sorted((s1, s2) -> {
        System.out.printf("sort: %s; %s\n", s1, s2);
        return s1.compareTo(s2);
    })
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    })
    .forEach(s -> System.out.println("forEach: " + s));
```

```
// filter:  d2
// filter:  a2
// filter:  b1
// filter:  b3
// filter:  c
// map:     a2
// forEach: A2
```

Here `sorted` is not called because `filter` filters to only one element.

Reuse of streams

- By default it is not possible
  - By calling a terminal operation the stream closes

```
Stream<String> stream =
Stream.of("d2", "a2", "b1", "b3", "c")
      .filter(s -> s.startsWith("a"));
stream.anyMatch(s -> true); // ok
stream.noneMatch(s -> true); // exception (IllegalStateException)
```

- Solution: for all terminal operations we want to call create a separate stream chain
  - Every `get()` creates a new stream

```
Supplier<Stream<String>> streamSupplier =
    () -> Stream.of("d2", "a2", "b1", "b3", "c").filter(s -> s.startsWith("a"));

streamSupplier.get().anyMatch(s -> true);   // ok
streamSupplier.get().noneMatch(s -> true);  // ok
```

collect

- Terminal operation with which we can create different forms of the elements of the stream
  - E.g. `List`, `Set`, `Map`, etc.
- It gets a `Collector` that has four operations:
  - A supplier
  - An accumulator
  - A combiner
  - A finisher
- There are many built-in `Collector`-s

collect examples

```java
class Person {
    String name;
    int age;

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

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

```java
List<Person> persons =
    Arrays.asList(
        new Person("Max", 18),
        new Person("Peter", 23),
        new Person("Pamela", 23),
        new Person("David", 12));
```

collect examples

```
List<Person> filtered =
    persons
        .stream()
        .filter(p -> p.name.startsWith("P"))
        .collect(Collectors.toList());

System.out.println(filtered);
// [Peter, Pamela]
```

```
IntSummaryStatistics ageSummary =
persons.stream().collect(
Collectors.summarizingInt(p -> p.age));

System.out.println(ageSummary);
// IntSummaryStatistics{count=4, sum=76,
min=12, average=19.000000, max=23}
```

```
Map<Integer, List<Person>> personsByAge = persons
    .stream()
    .collect(Collectors.groupingBy(p -> p.age));

personsByAge
    .forEach((age, p) -> System.out.format("age %s: %s\n", age, p));

// age 18: [Max]
// age 23: [Peter, Pamela]
// age 12: [David]
```

How to create a Collector

- Let’s link the names of all persons by a pipe (|) character

```
Collector<Person, StringJoiner, String> personNameCollector =
    Collector.of(() -> new StringJoiner(" | "), // supplier
                 (j, p) -> j.add(p.name.toUpperCase()), // accumulator
                 (j1, j2) -> j1.merge(j2), // combiner
                 StringJoiner::toString); // finisher

String names = persons.stream().collect(personNameCollector);

System.out.println(names); // MAX | PETER | PAMELA | DAVID
```

reduce

- Reduces all the elements of a stream to only one element
- It has three versions
  - Reduces a stream of elements to exactly one element
    - E.g.: Who is the oldest?
  - Reduces on a base of a unity element and a BinaryOperator accumulator
  - Reduces on a base of a unity element, a BiFunction accumulator and a BinaryOperator typed combiner function
    - It is often more simple to simply combine a map and reduce

reduce examples

Who is the oldest?

```
persons
    .stream()
    .reduce((p1, p2) -> p1.age > p2.age ? p1 : p2)
    .ifPresent(System.out::println);    // Pamela
```

Let’s create a Person object, that aggregates the names and ages of elements of the stream.

```
Person result =
    persons
        .stream()
        .reduce(new Person("", 0), (p1, p2) -> {
            p1.age += p2.age;
            p1.name += p2.name;
            return p1;
        });

System.out.format("name=%s; age=%s", result.name, result.age);
// name=MaxPeterPamelaDavid; age=76}
```

peek

- Intermediate operation

```
System.out.println("sum: " +
        IntStream.range(1, 11)
                .map(i -> i + 5)
                .peek(i -> System.out.print(i + " "))
                .reduce(0, (i, j) -> i + j)
);
// 6 7 8 9 10 11 12 13 14 15 sum: 105
```

Make it parallel!

```
Integer ageSum = persons
    .parallelStream()
    .reduce(0,
        (sum, p) -> {
            System.out.format("accumulator: sum=%s; person=%s\n", sum, p);
            return sum += p.age;
        },
        (sum1, sum2) -> {
            System.out.format("combiner: sum1=%s; sum2=%s\n", sum1, sum2);
            return sum1 + sum2;
        });

// accumulator: sum=0; person=Pamela
// accumulator: sum=0; person=David
// accumulator: sum=0; person=Max
// accumulator: sum=0; person=Peter
// combiner: sum1=18; sum2=23
// combiner: sum1=23; sum2=12
// combiner: sum1=41; sum2=35
```

Parallel streams

- The running performance may increase
- A so called `ForkJoinPool` is used that forks to threads

```
ForkJoinPool commonPool = ForkJoinPool.commonPool();
System.out.println(commonPool.getParallelism());    // 3
```

- It can be reached in two ways:
  - Through the `parallelStream()` method of collections
  - By the `parallel()` intermediate operation that parallelizes a sequential stream

Example

```
Arrays.asList("a1", "a2", "b1", "c2", "c1")
    .parallelStream()
    .filter(s -> {
        System.out.format("filter: %s [%s]\n",
         s, Thread.currentThread().getName());
        return true;
    })
    .map(s -> {
        System.out.format("map: %s [%s]\n",
         s, Thread.currentThread().getName());
        return s.toUpperCase();
    })
    .forEach(
        s -> System.out.format("forEach: %s [%s]\n",
        s, Thread.currentThread().getName()));
```

```
filter: b1 [main]
map: b1 [main]
filter: a2 [ForkJoinPool.commonPool-worker-1]
filter: a1 [ForkJoinPool.commonPool-worker-2]
forEach: B1 [main]
filter: c1 [ForkJoinPool.commonPool-worker-3]
map: c1 [ForkJoinPool.commonPool-worker-3]
forEach: C1 [ForkJoinPool.commonPool-worker-3]
filter: c2 [main]
map: c2 [main]
forEach: C2 [main]
map: a1 [ForkJoinPool.commonPool-worker-2]
forEach: A1 [ForkJoinPool.commonPool-worker-2]
map: a2 [ForkJoinPool.commonPool-worker-1]
forEach: A2 [ForkJoinPool.commonPool-worker-1]
```

Aggregated operations and iterators

- Aggregated operations may look similar to iterators (e.g. `forEach`)
- Differences are:
    - They use inner iteration
        - They do not have a method like next()
        - Their use defines on which collection we use them but the JDK tells how
    - They process elements of a stream
        - And not the elements of the collection itself. This is the very reason why we call them stream operations.
    - We describe the behavior by parameters.
        - Most aggregated operations are paramterizable so they are configurable

Other new features

java.util.Optional

- A container object that may hold a null value or may not
- Since Java 8
- Important methods:
  - static empty(): returns an empty Optional instance
  - static of(value): returns an Optional that has a given value
  - static ofNullable(value): If value is not null it is returned as an optional else it returns an empty optional.
  - get(): If in the Optional we have a value it returns it else it throws NoSuchElementException
  - ifPresent(): Returns true if there is an element in the optional. (Else false.)
  - orElse(other): If there is a value it is returned else the Other object.

What is Optional good for?

- We force the caller to handle the case if an object does not exist
    - So we will have less NPE-s (NullPointerException)
- All methods that may not return a value should have Optional as a return type.

How to use?

```
public Optional<Student> findStudentByNeptunId(String neptunId) {
 …
}

Optional<Student> optional = findStudentByNeptunId(neptunId);

if (optional.isPresent()) {
    Student st = optional.get();
    /* use the Student object */
}
else {
    /* handle the case if there is no such element */
}
```

java.util.StringJoiner

- Creates a character sequence separated by given character sequence between a given optional prefix and suffix

```
int[] t = new int[] {1,2,3,4,5,6,7,8,9,10};
StringJoiner sj = new StringJoiner(", ", "[", "]");
for (int i : t)
  sj.add(i + ""); // String.valueOf(i);
System.out.println(sj); // [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

- **Collectors.joining**

```
System.out.println(Arrays.stream(t)
    .mapToObj(i -> i + "") // .mapToObj(String::valueOf)
    .collect(Collectors.joining(", ", "[", "]"))); // [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Streams and lambda practice

- https://github.com/JavaCodeKata/stream-lambda
- https://github.com/AdoptOpenJDK/lambda-tutorial
- http://technologyconversations.com/2014/10/16/java-tutorial-through-katas/
  - https://github.com/vfarcic/java-8-exercises