What's New in Java 8

Introduction to Lambda Expression in Java 8

José Paumard
blog.paumard.org
@JosePaumard

pluralsight
hardcore dev and IT training

What You Will Learn

- The most useful new parts of Java 8
- Lambda expressions
- The Stream API and Collectors
- And many bits and pieces
- Java FX
- Nashorn

Course Overview

- Java 8 lambda expressions and Interfaces
- Stream API and Collectors
- Date and Time API
- Strings, I/O and other bits and pieces
- Rich interfaces: Java FX
- Nashorn, a new Javascript engine for the JVM

Targeted Audience

- This is a Java course
- Basic knowledge of the main APIs
- Generics
- Collection API
- Java I/O

Module Outline

- Introduction to the « Lambda expressions »
- The lambda syntax
- Functional interfaces
- Method references
- Constructor references
- How to process data from the Collection API?

What Is a Lambda Expression for?

- A simple example

```
public interface FileFilter {

   boolean accept(File file) ;
}
```

What Is a Lambda Expression for?

- Let's implement this interface

```
public class JavaFileFilter implements FileFilter {

   public boolean accept(File file) {
      return file.getName().endsWith(".java");
   }
}
```

- And use it:

```
JavaFileFilter fileFilter = new JavaFileFilter();
File dir = new File("d:/tmp");
File[] javaFiles = dir.listFiles(fileFilter);
```

What Is a Lambda Expression for?

- Let's use an anonymous class

```
FileFilter fileFilter = new FileFilter() {

   @Override
   public boolean accept(File file) {
      return file.getName().endsWith(".java");
   }
};

File dir = new File("d:/tmp");
File[] javaFiles = dir.listFiles(fileFilter);
```

What Is a Lambda Expression for?

- The first answer is :

To make instances of anonymous classes easier to write and read!

A First Lambda Expression

- Let's use an anonymous class

```
FileFilter fileFilter = new FileFilter() {

   @Override
   public boolean accept(File file) {
      return file.getName().endsWith(".java");
   }
};
```

A First Lambda Expression

- Let's use an anonymous class

```
FileFilter fileFilter = new FileFilter() {

   @Override
   public boolean accept(File file) {
      return file.getName().endsWith(".java");
   }
};
```

We take the parameters

```
FileFilter filter = (File file)
```

A First Lambda Expression

- Let's use an anonymous class

```
FileFilter fileFilter = new FileFilter() {

   @Override
   public boolean accept(File file) {
      return file.getName().endsWith(".java");
   }
};
```

and then…

```
FileFilter filter = (File file) ->
```

A First Lambda Expression

- Let's use an anonymous class

```
FileFilter fileFilter = new FileFilter() {

   @Override
   public boolean accept(File file) {
      return file.getName().endsWith(".java");
   }
};
```

return this

```
FileFilter filter = (File file) -> file.getName().endsWith(".java");
```

A First Lambda Expression

- Let's use an anonymous class

```
FileFilter fileFilter = new FileFilter() {

   @Override
   public boolean accept(File file) {
      return file.getName().endsWith(".java");
   }
};
```

- This is a Java 8 lambda expression:

```
FileFilter filter = (File file) -> file.getName().endsWith(".java");
```

So What Is a Java 8 Lambda Expression?

- Answer: another way of writing instances of anonymous classes

- Live coding : FileFilter, Runnable, Comparator

Several Ways of Writing a Lambda Expression

- The simplest way:

```
FileFilter filter = (File file) -> file.getName().endsWith(".java");
```

- If I have more than one line of code:

```
Runnable r = () -> {
   for (int i = 0; i < 5; i++) {
      System.out.println("Hello world!");
   }
};
```

Several Ways of Writing a Lambda Expression

- If I have more than one argument:

```
Comparator<String> c =
   (String s1, String s2) ->
      Integer.compare(s1.length(), s2.length());
```

Three Questions About Lambdas

- What is the type of a lambda expression?
- Can a lambda be put in a variable?
- Is a lambda expression an object?

What Is the Type of a Lambda Expression?

- Answer: a functional interface
- What is a functional interface?

Functional Interface

- A functional interface is an interface with only one *abstract* method
- Example:

```
public interface Runnable {

   run();
};
```

```
public interface Comparator<T> {

   int compareTo(T t1, T t2);
};
```

```
public interface FileFilter {

   boolean accept(File pathname);
};
```

Functional Interface

- A functional interface is an interface with only one *abstract* method
- Methods from the Object class don't count:

```
public interface MyFunctionalInterface {

   someMethod();

   /**
    * Some more documentation
    */
   equals(Object o);
};
```

Functional Interface

- A functional interface can be annotated

```
@FunctionalInterface
public interface MyFunctionalInterface {

   someMethod();

   /**
    * Some more documentation
    */
   equals(Object o);
};
```

- It is just here for convenience, the compiler can tell me whether the interface is functional or not

Three Questions About Lambdas

- What is the type of a lambda expression?
  - Answer: a functional interface
- Can a lambda be put in a variable?
- Is a lambda expression an object?

Can I Put a Lambda Expression in a Variable?

- Answer is yes!

```
Comparator<String> c =
   (String s1, String s2) ->
      Integer.compare(s1.length(), s2.length());
```

- Consequences: a lambda can be taken as a method parameter, and can be returned by a method

Three Questions About Lambdas

- What is the type of a lambda expression?
  - Answer: a functional interface
- Can a lambda be put in a variable?
  - Answer: yes!
- Is a lambda expression an object?

Is a Lambda an Object?

- This question is tougher than it seems…

Is a Lambda an Object?

- Let's compare the following:

```
Comparator<String> c =
   (String s1, String s2) ->
      Integer.compare(s1.length(), s2.length());
```

```
Comparator<String> c =
   new Comparator<String>(String s1, String s2) {

      public boolean compareTo(String s1, String s2) {
         Integer.compare(s1.length(), s2.length());
      }
   };
```

Is a Lambda an Object?

- Let's compare the following:

```
Comparator<String> c =
   (String s1, String s2) ->
      Integer.compare(s1.length(), s2.length());
```

```
Comparator<String> c =
   new Comparator<String>(String s1, String s2) {

      public boolean compareTo(String s1, String s2) {
         Integer.compare(s1.length(), s2.length());
      }
   };
```

- A lambda expression is created without using « new »

Three Questions About Lambdas

- What is the type of a lambda expression?
  - Answer: a functional interface
- Can a lambda be put in a variable?
  - Answer: yes!
- Is a lambda expression an object?
  - The answer is complex, but no
  - Exact answer: a lambda is an object without an identity

Functional Interfaces Toolbox

- New package : java.util.function
- With a rich set of functional interfaces

Package java.util.function

- 4 categories:
- Supplier

```
@FunctionalInterface
public interface Supplier<T> {

   T get();
}
```

Package java.util.function

- 4 categories:
- Consumer / BiConsumer

```
@FunctionalInterface
public interface Consumer<T> {

   void accept(T t);
}
```

```
@FunctionalInterface
public interface BiConsumer<T, U> {

   void accept(T t, U u);
}
```

Package java.util.function

- 4 categories:
- Predicate / BiPredicate

```
@FunctionalInterface
public interface Predicate<T> {

   boolean test(T t);
}
```

```
@FunctionalInterface
public interface BiPredicate<T, U> {

   boolean test(T t, U u);
}
```

Package java.util.function

- 4 categories:
- Function / BiFunction

```
@FunctionalInterface
public interface Function<T, R> {

   R apply (T t);
}
```

```
@FunctionalInterface
public interface BiFunction<T, U, R> {

   R apply (T t, U u);
}
```

Package java.util.function

- 4 categories:
- Function / UnaryOperator

```
@FunctionalInterface
public interface Function<T, R> {

   R apply (T t);
}
```

```
@FunctionalInterface
public interface UnaryOperator<T> extends Function<T, T> {
}
```

Package java.util.function

- 4 categories:
- BiFunction / BinaryOperator

```
@FunctionalInterface
public interface Function<T, U, R> {

   R apply (T t, U u);
}
```

```
@FunctionalInterface
public interface BinaryOperator<T> extends BiFunction<T, T, T> {
}
```

More Lambda Expressions Syntax

- Most of the time, parameter types can be omitted

```
Comparator<String> c =
   (String s1, String s2) ->
      Integer.compare(s1.length(), s2.length());
```

- Becomes:

```
Comparator<String> c =
   (s1, s2) ->
      Integer.compare(s1.length(), s2.length());
```

Method References

- This lambda expression:

```
Function<String, String> f = s -> s.toLowerCase();
```

- Can be written like that:

```
Function<String , String> f = String::toLowerCase;
```

Method References

- This lambda expression:

```
Consumer<String> c = s -> System.out.println(s);
```

- Can be written like that:

```
Consumer<String> c = System.out::println;
```

Method References

- This lambda expression:

```
Comparator<Integer> c = (i1, i2) -> Integer.compare(i1, i2);
```

- Can be written like that:

```
Comparator<Integer> c = Integer::compare;
```

So What Do We Have so Far?

- A new concept: the « lambda expression », with a new syntax
- A new interface concept: the « functional interface »
- Question: how can we use this to process data?

How Do We Process Data in Java?

- Where are our objects?
- Most of the time: in a Collection (or maybe a List, a Set or a Map)
- Can I process this data with lambdas?

```
List<Customer> list = ...;
list.forEach(customer -> System.out.println(customer));
```

- Or:

```
List<Customer> list = ...;
list.forEach(System.out::println);
```

Can I Process This Data with Lambdas?

- The good news is: yes!
- We can write:

```
List<Customer> list = ...;
list.forEach(System.out::println);
```

- But… where does this forEach method come from?
- Adding a forEach method on the Collection interface breaks the compatibility: all the implementations have to be refactored!

How to Add Methods to Iterable?

- Without breaking all the existing implementations?

```
public interface Iterable<E> {

   // the usual methods

   void forEach(Consumer<E> consumer);
}
```

- Refactoring these implementations is not an option

How to Add Methods to Iterable?

- If we cant put the implementation in ArrayList, then…

```
public interface Iterable<E> {

   // the usual methods

   default void forEach(Consumer<E> consumer) {

      for (E e : this) {
         consumer.accept(e);
      }
   }
}
```

Default Methods

- This is a new Java 8 concept
- It allows to change the old interfaces without breaking the existing implementations
- It also allows new patterns!

- And by the way…
- Static methods are also allowed in Java 8 interfaces!

Examples Of New Patterns

- Predicates

```
Predicate<String> p1 = s -> s.length() < 20;
Predicate<String> p2 = s -> s.length() > 10;

Predicate<String> p3 = p1.and(p2);
```

```
@FunctionalInterface
public interface Predicate<T> {

   boolean test(T t);

   default Predicate<T> and(Predicate<? super T> other) {
      Objects.requireNonNull(other);
      return (t) -> test(t) && other.test(t);
   }
}
```

Examples Of New Patterns

- Predicates

```
Predicate<String> id = Predicate.isEqual(target);
```

```
@FunctionalInterface
public interface Predicate<T> {

   boolean test(T t);

   static <T> Predicate<T> isEqual(Object targetRef) {
      return (null == targetRef)
             ? Objects::isNull
             : object -> targetRef.equals(object);
   }
}
```

Summary

- The new « lambda expression » syntax
- A lambda expression has a type : a functional interface
- Definition of a functional interface, examples
- Method and constructor references
- Iterable.forEach method
- Default and static methods in interfaces, examples

Streams & Collectors

New APIs for map / filter / reduce

José Paumard
blog.paumard.org
@JosePaumard

pluralsight
hardcore dev and IT training

Module Outline

- Introduction: map / filter / reduce
- What is a « Stream »?
- Patterns to build a Stream
- Operations on a Stream

Map / Filter / Reduce

- Example:
- Let's take a list a Person

```
List<Person> list = new ArrayList<>() ;
```

- Suppose we want to compute the

« average of the age of the people older than 20 »

Map / Filter / Reduce

- 1st step: mapping
- The mapping step takes a List<Person> and returns a List<Integer>
- The size of both lists is the same

Map / Filter / Reduce

- 2nd step: filtering
- The filtering step takes a List<Integer> and returns a List<Integer>
- But there some elements have been filtered out in the process

Map / Filter / Reduce

- 3rd step: average
- This is the reduction step, equivalent to the SQL aggregation

What Is a Stream?

- Technical answer: a typed interface

```
public interface Stream<T> extends BaseStream<T, Stream<T>> {

        // ...
}
```

- And a new concept!

What Is a Stream?

- What does it do?
- It gives ways to efficiently process large amounts of data… and also smaller ones

What Is a Stream?

- What does *efficiently* mean?
- Two things:
- In parallel, to leverage the computing power of multicore CPUs
- Pipelined, to avoid unnecessary intermediary computations

What Is a Stream?

- Why can't a Collection be a Stream?
- Because Stream is a new concept, and we dont want to change the way the Collection API works

What Is a Stream?

- So what is a Stream?
- An object on which one can define *operations*
- An object that does not hold any data
- An object that should not change the data it processes
- An object able to process data in « one pass »
- An object optimized from the algorithm point of view, and able to process data in parallel

How Can We Build a Stream?

- Many patterns!

```
List<Person> persons = ... ;

Stream<Person> stream = persons.stream();
```

A First Operation

- First operation: forEach()

```
List<Person> persons = ... ;

Stream<Person> stream = persons.stream();
stream.forEach(p -> System.out.println(p));
```

- Prints all the elements of the list
- It takes an instance of Consumer as an argument

A First Operation

- Interface Consumer<T>

```
@FunctionalInterface
public interface Consumer<T> {

    void accept(T t);
}
```

- Consumer<T> is a *functional interface*
- Can be implemented by a lambda expression

```
Consumer<T> c = p -> System.out.println(p);
```

```
Consumer<T> c = System.out::println; // Method reference
```

A First Operation

- In fact Consumer<T> is a bit more complex

```
@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); };
    }
}
```

- One can chain consumers!

A First Operation

- Let's chain consumers

```
List<String> list = new ArrayList<>();

Consumer<String> c1 = s -> list.add(s);
Consumer<String> c2 = s -> System.out.println(s);
```

A First Operation

- Let's chain consumers

```
List<String> list = new ArrayList<>();

Consumer<String> c1 = list::add;
Consumer<String> c2 = System.out::println;

Consumer<String> c3 = c1.andThen(c2);
```

A First Operation

- Only way to have several consumers on a single stream

```
List<String> result = new ArrayList<>();
List<Person> persons = ...;

Consumer<String> c1 = result::add;
Consumer<String> c2 = System.out::println;

persons.stream()
       .forEach(c1.andThen(c2));
```

- Because forEach() does not return anything

A Second Operation: Filter

- Example:

```
List<Person> list = ...;
Stream<Person> stream = list.stream();
Stream<Person> filtered =
    stream.filter(person -> person.getAge() > 20);
```

- Takes a predicate as a parameter:

```
Predicate<Person> p = person -> person.getAge() > 20;
```

A Second Operation: Filter

- Predicate interface, with default methods:

```
@FunctionalInterface
public interface Predicate<T> {

    boolean test(T t);

    default Predicate<T> and(Predicate<? super T> other) { ... }

    default Predicate<T> or(Predicate<? super T> other) { ... }

    default Predicate<T> negate() { ... }
}
```

A Second Operation: Filter

- Predicates combinations examples:

```
Predicate<Integer> p1 = i -> i > 20;
Predicate<Integer> p2 = i -> i < 30;
Predicate<Integer> p3 = i -> i == 0;

Predicate<Integer> p = p1.and(p2).or(p3); // (p1 AND p2) OR p3
Predicate<Integer> p = p3.or(p1).and(p2); // (p3 OR p1) AND p2
```

- Warning: method calls do not handle priorities

A Second Operation: Filter

- Predicate interface, with static method:

```
@FunctionalInterface
public interface Predicate<T> {

    boolean test(T t);

    // default methods

    static <T> Predicate<T> isEqual(Object o) { ... }
}
```

- Example:

```
Predicate<String> p = Predicate.isEqual("two") ;
```

A Second Operation: Filter

- Use case:

```
Predicate<String> p = Predicate.isEqual("two") ;

Stream<String> stream1 = Stream.of("one", "two", "three") ;

Stream<String> stream2 = stream1.filter(p) ;
```

- The filter method returns a Stream
- This Stream is a new instance

A Second Operation: Filter

- Question: what do I have in this new Stream?
- Simple answer: the filtered data

- Really?
- We just said: « a stream does not hold any data »

A Second Operation: Filter

- Question: what do I have in this new Stream?
- Simple answer: ~~the filtered data~~ WRONG!
- The right answer is: nothing, since a Stream does not hold any data
- So, what does this code do?

```
List<Person> list = ... ;
Stream<Person> stream = list.stream();
Stream<Person> filtered =
        stream.filter(person -> person.getAge() > 20);
```

- Answer is: nothing

This call is only a declaration, no data is processed

A Second Operation: Filter

- The call to the filter method is *lazy*
- And all the methods of Stream that return another Stream are *lazy*
- Another way of saying it:

an operation on a Stream that returns a Stream
is called an intermediary operation

Back to the Consumer

- What does this code do?

```
List<String> result = new ArrayList<>();
List<Person> persons = ... ;

persons.stream()
       .peek(System.out::println)
       .filter(person -> person.getAge() > 20)
       .peek(result::add);
```

- Hint: the peek() method returns a Stream

Back to the Consumer

- What does this code do?

```
List<String> result = new ArrayList<>();
List<Person> persons = ... ;

persons.stream()
       .peek(System.out::println)
       .filter(person -> person.getAge() > 20)
       .peek(result::add);
```

- Answer: nothing!
- This code does not print anything
- The list « result » is empty

Summary

- The Stream API defines *intermediary operations*
- We saw 3 operations:
- forEach(Consumer) (not lazy)
- peek(Consumer) (lazy)
- filter(Predicate) (lazy)

Mapping Operation

- Example:

```
List<Person> list = ... ;
Stream<Person> stream = list.stream();
Stream<String> names =
        stream.map(person -> person.getName());
```

- map() returns a Stream, so it is an intermediary operation

Mapping Operation

- … with default methods to chain and compose mappings

```
@FunctionalInterface
public interface Function<T, R> {

    R apply(T t);

    default <V> Function<V, R> compose(Function<V, T> before);

    default <V> Function<T, V> andThen(Function<R, V> after);
}
```

- In fact this is the simplified version, beware the generics!

Mapping Operation

- compose() and andThen() methods with their exact signatures

```
@FunctionalInterface
public interface Function<T, R> {

    R apply(T t);

    default <V> Function<V, R> compose(
        Function<? super V, ? extends T> before);

    default <V> Function<T, V> andThen(
        Function<? super R, ? extends V> after);
}
```

Mapping Operation

- One static method: identity

```
@FunctionalInterface
public interface Function<T, R> {

    R apply(T t);

    // default methods

    static <T> Function<T, T> identity() {
        return t -> t;
    }
}
```

Flatmapping Operation

- Method flatMap()
- Signature:

```
<R> Stream<R> flatMap(Function<T, Stream<R>> flatMapper);
```

```
<R> Stream<R> map(Function<T, R> mapper);
```

- The flatMapper takes an element of type T, and returns an element of type Stream<R>

Flatmapping Operation

- Method flatMap()
- Signature:

```
<R> Stream<R> flatMap(Function<T, Stream<R>> flatMapper);
```

```
<R> Stream<R> map(Function<T, R> mapper);
```

- If the flatMap was a regular map, it would return a Stream<Stream<R>>
- Thus a « stream of streams »

Flatmapping Operation

- Method flatMap()
- Signature:

```
<R> Stream<R> flatMap(Function<T, Stream<R>> flatMapper);
```

```
<R> Stream<R> map(Function<T, R> mapper);
```

- If the flatMap was a regular map, it would return a Stream<Stream<R>>
- But it is a flatMap!
- Thus the « stream of streams » is flattened, and becomes a stream

Summary

- 3 categories of operations:
- forEach() and peek()
- filter()
- map() and flatMap()

Reduction

- And what about the reduction step?
- Two kinds of reduction in the Stream API
- 1st: aggregation = min, max, sum, etc…

Reduction

- How does it work?

```
List<Integer> ages = ... ;
Stream<Integer> stream = ages.stream();
Integer sum =
        stream.reduce(0, (age1, age2) -> age1 + age2);
```

- 1st argument: identity element of the reduction operation
- 2nd argument: reduction operation, of type BinaryOperator<T>

BinaryOperator

- A BinaryOperator is a special case of BiFunction

```
@FunctionalInterface
public interface BiFunction<T, U, R> {

    R apply(T t, U u);

    // plus default methods
}
```

```
@FunctionalInterface
public interface BinaryOperator<T>
extends BiFunction<T, T, T> {

    // T apply(T t1, T t2);

    // plus static methods
}
```

Identity Element

- The bifunction takes two arguments, so…
- What happens if the Stream is empty?
- What happens if the Stream has only one element?

- The reduction of an empty Stream is the identity element
- If the Stream has only one element, then the reduction is that element

Aggregations

- Examples:

```
Stream<Integer> stream = ...;
BinaryOperation<Integer> sum = (i1, i2) -> i1 + i2;
Integer id = 0; // identity element for the sum

int red = stream.reduce(id, sum);
```

```
Stream<Integer> stream = Stream.empty();

int red = stream.reduce(id, sum);
System.out.println(red);
```

- Will print:

```
> 0
```

Aggregations

- Examples:

```
Stream<Integer> stream = ...;
BinaryOperation<Integer> sum = (i1, i2) -> i1 + i2;
Integer id = 0; // identity element for the sum

int red = stream.reduce(id, sum);
```

```
Stream<Integer> stream = Stream.of(1);

int red = stream.reduce(id, sum);
System.out.println(red);
```

- Will print:

```
> 1
```

Aggregations

- Examples:

```
Stream<Integer> stream = ...;
BinaryOperation<Integer> sum = (i1, i2) -> i1 + i2;
Integer id = 0; // identity element for the sum

int red = stream.reduce(id, sum);
```

```
Stream<Integer> stream = Stream.of(1, 2, 3, 4);

int red = stream.reduce(id, sum);
System.out.println(red);
```

- Will print:

```
> 10
```

Aggregations: Corner Case

- Suppose the reduction is the max

```
BinaryOperation<Integer> max =
    (i1, i2) ->
    i1 > i2 ? i1 : i2;
```

- The problem is, there is no identity element for the max reduction
- So the max of an empty Stream is undefined…

Aggregations: Corner Case

- Then what is the return type of the this call?

```
List<Integer> ages = ... ;
Stream<Integer> stream = ages.stream();
... max =
        stream.max(Comparator.naturalOrder());
```

- If it is an int, then the default value is 0…

Aggregations: Corner Case

- Then what is the return type of the this call?

```
List<Integer> ages = ... ;
Stream<Integer> stream = ages.stream();
... max =
        stream.max(Comparator.naturalOrder());
```

- If it is an Integer, then the default value is null…

Optionals

- Then what is the return type of the this call?

```
List<Integer> ages = ... ;
Stream<Integer> stream = ages.stream();
Optional<Integer> max =
        stream.max(Comparator.naturalOrder());
```

- Optional means « there might be no result »

Optionals

- How to use an Optional?

```
Optional<String> opt = ... ;
if (opt.isPresent()) {
        String s = opt.get() ;
} else {
        ...
}
```

- The method isPresent() returns true if there is something in the optional
- The method get() returns the value held by this optional

Optionals

- How to use an Optional?

```
Optional<String> opt = ... ;
if (opt.isPresent()) {
        String s = opt.get() ;
} else {
        ...
}
```

- The method orElse() encapsulates both calls

```
String s = opt.orElse("") ; // defines a default value
```

Optionals

- How to use an Optional?

```
Optional<String> opt = ... ;
if (opt.isPresent()) {
        String s = opt.get() ;
} else {
        ...
}
```

- The method orElseThrow() defines a thrown exception

```
String s = opt.orElseThrow(MyException::new) ; // lazy construct.
```

Reductions

- Available reductions:
  - max(), min()
  - count()
- Boolean reductions
  - allMatch(), noneMatch(), anyMatch()
- Reductions that return an optional
  - findFirst(), findAny()

Reductions

- Reductions are *terminal operations*
- They trigger the processing of the data

Terminal Operation

- Example:

```
List<Person> persons = ...;

Optional<Integer> minAge =
persons.map(person -> person.getAge())   // Stream<Integer>
       .filter(age -> age > 20)          // Stream<Integer>
       .min(Comparator.naturalOrder());  // terminal operation
```

Terminal Operation

- Example, optimization:

```
List<Person> persons = ... ;

persons.map(person -> person.getLastName())
       .allMatch(length < 20);              // terminal op.
```

- The map / filter / reduce operations are evaluated in one pass over the data

Summary

- Reduction seen as an aggregation
- Intermediary / terminal operation
- Optional: needed because default values cant be always defined

Collectors

- There is another type of reduction
- Called « mutable » reduction
- Instead of aggregating elements, this reduction put them in a « container »

Collecting in a String

- Example:

```
List<Person> persons = ... ;

String result =
persons.stream()
       .filter(person -> person.getAge() > 20)
       .map(Person::getLastName)
       .collect(
          Collectors.joining(", ")
       );
```

- Result is a String with all the names of the people in persons, older than 20, separated by a comma

Collecting in a List

- Example:

```
List<Person> persons = ... ;

List<String> result =
persons.stream()
       .filter(person -> person.getAge() > 20)
       .map(Person::getLastName)
       .collect(
          Collectors.toList()
       );
```

- Result is a List of String with all the names of the people in persons, older than 20

Collecting in a Map

- Example:

```
List<Person> persons = ... ;

Map<Integer, List<Person>> result =
persons.stream()
       .filter(person -> person.getAge() > 20)
       .collect(
          Collectors.groupingBy(Person::getAge)
       );
```

- Result is a Map containing the people of persons, older than 20
  - The keys are the ages of the people
  - The values are the lists of the people of that age

Collecting in a Map

- Example:

```
List<Person> persons = ... ;

Map<Integer, List<Person>> result =
persons.stream()
       .filter(person -> person.getAge() > 20)
       .collect(
          Collectors.groupingBy(Person::getAge)
       );
```

- It is possible to « post-process » the values, with a *downstream collector*

Collecting in a Map

- Example:

```
List<Person> persons = ... ;

Map<Integer, Long> result =
persons.stream()
       .filter(person -> person.getAge() > 20)
       .collect(
          Collectors.groupingBy(Person::getAge),
          Collectors.counting() // the downstream collector
       );
```

- Collectors.counting() just counts the number of people of each age

So What Is a Stream?

- **An object that allows one to define processings on data**
  - There is no limit on the amount of data that can be processed
- **Those processings are typically map / filter / reduce operations**

- **Those processings are optimized :**
- **First, we define all the operations**
- **Then, the operations are triggered**

So What Is a Stream?

- **Last remark:**
- **A Stream cannot be « reused »**
- **Once it has been used ot process a set of data, it cannot be used again to process another set**

Summary

- **Quick explanation of the map / filter / reduce**
- **What is a Stream**
- **The difference between *intermediary* and *final* operations**
- **The « consuming » operations: forEach() and peek()**
- **The « mapping » operations: map() and flatMap()**
- **The « filter » operation: filter()**
- **The « reduction » operations:**
  - Aggregations: reduce(), max(), min(), …
  - Mutable reductions: collect, Collectors

Java 8 Date and Time API

The Java 8 Date and Time API

José Paumard
blog.paumard.org
@JosePaumard

Module Outline

- Why do we need a new Date API in Java 8?
- The new Date API from Java 8: 7 concepts
- Instant and Duration
- LocalDate, Period
- TemporalAdjusters
- LocalTime
- ZonedTime
- Date formatters

The Date API in Java 7

- One class : java.util.Date (and java.sql.Date) [JDK 1.0]
- And one pattern

```
Date date = new Date(); // just now !
```

The Date API in Java 7

- How can I create a date for the 2014 / 2 / 10?
- I must use the Calendar class

```
Calendar cal = Calendar.getInstance(); // just now !
cal.set(2014, 1, 10);                  // january is 0

Date feb10th = cal.getTime();
```

- How can I add 7 days to feb10th?

```
cal.add(Calendar.DAY_OF_MONTH, 7);

Date oneWeekLater = cal.getTime(); // one week later
```

The Date API in Java 7

- The Date class is *mutable*: what does it mean?
- Here is an example

```
public class Customer {

   private Date creationDate;

   public Date getCreationDate() {
      return this.creationDate;
   }
}
```

The Date API in Java 7

- Some other code could do that

```
Customer customer = new Customer();

Date d = customer.getCreationDate();
d.setTime(0L);
```

- Thus modifying the *value* of the date of creation of the customer object

- How can I prevent that?

The Date API in Java 7

- Use a defensive copy!

```
public class Customer {

   private Date creationDate ;

   public Date getCreationDate() {
      return new Date(this.creationDate.getTime()) ;
   }
}
```

- Overheads: new object to create on each call, overhead for the garbage collector
- Having a mutable Date class has a cost!

The Date API in Java 8

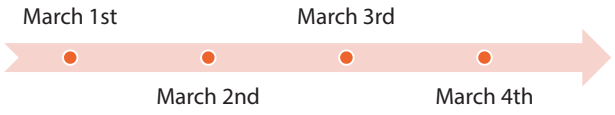
- New API, package is java.time
- New key concepts
- Interoperation with the legacy API

1st Concept: Instant

- And Instant is a point on the time line

1st Concept: Instant

- And Instant is a point on the time line



- The precision is the nanosecond!

1st Concept: Instant

- And Instant is a point on the time line

- Instant 0 is the January the 1st, 1970 at midnight GMT
- Instant.*MIN* is 1 billion years ago
- Instant.*MAX* is Dec. 31 of the year 1,000,000,000
- Instant.*now*() is the current instant

1st Concept: Instant

- And Instant is a point on the time line

- Precision is the nanosecond

1st Concept: Instant

- An Instant is immutable

1st Concept: Instant

- An Instant is immutable
- How can I use Instant?

```
Instant start = Instant.now();

// some long computations

Instant end = Instant.now();
```

- New concept: Duration

```
Duration elapsed = Duration.between(start, end);
long millis = elapsed.toMillis();
```

2nd Concept: Duration

- A Duration is the amount of time between two Instant
- Methods :
- toNanos(), toMillis(), toSeconds(), toMinutes(), toHours(), toDays()
- minusNanos(), …
- plusNanos(), …
- And also :
- multipliedBy(), dividedBy(), negated()
- isZero(), isNegative()

Many Cases Are Not Covered

- There are many cases where a date is not an « instant »
- Ex: « Shakespeare was born Apr. 23rd, 1564 »
- Ex: « Let us meet a 1pm and have lunch together! »

3rd Concept: LocalDate

- We need another concept for those « dates »
- New concept: LocalDate
- How to create a LocalDate?

```
LocalDate now = LocalDate.now();
LocalDate dateOfBirth =
   LocalDate.of(1564, Month.APRIL, 23);
```

4th Concept: Period

- A Period is the amount of time between two LocalDate
- Same concept as Duration, same kind of methods
- When was Shakespeare born?

```
Period p = dateOfBirth.until(now);
System.out.println("# years = " + p.getYears());
```

```
long days = dateOfBirth.until(now, ChronoUnit.DAYS);
System.out.println("# days = " + days);
```

5th Concept: DateAdjuster

- Useful to add (or substract) an amount of time to an Instant or a LocalDate
- Use the method with()

```
LocalDate now = LocalDate.now();
LocalDate nextSunday =
   now.with(TemporalAdjusters.next(DayOfWeek.SUNDAY));
```

TemporalAdjusters

- 14 static methods to adjust an Instant or a LocalDate
- firstDayOfMonth(), lastDayOfMonth()
- firstDayOfYear(), lastDayOfYear()
- firstDayOfNextMonth(), firstDayOfNextYear()

TemporalAdjusters

- 14 static methods to adjust an Instant or a LocalDate
- firstInMonth(DayOfWeek.MONDAY)
- lastInMonth(DayOfWeek.TUESDAY)
- dayOfWeekInMonth(2, DayOfWeek.THURSDAY)

TemporalAdjusters

- 14 static methods to adjust an Instant or a LocalDate
- next(DayOfWeek.SUNDAY)
- nextOrSame(DayOfWeek.SUNDAY)
- previous(DayOfWeek.SUNDAY)
- previousOrSame(DayOfWeek.SUNDAY)

6th Concept: LocalTime

- A LocalTime is a time of day
- Ex: 10:20
- Pattern

```
LocalTime now = LocalTime.now();
LocalTime time = LocalTime.of(10, 20) ; // 10:20
```

- Plus a set of methods to manipulate the time

```
LocalTime bedTime = LocalTime.of(23, 0);
LocalTime wakeUpTime = bedTime.plusHours(8); // 7:00
```

7th Concept: Zoned Time

- There are Time Zones all over the earth
- Java uses the IANA database (https://www.iana.org/time-zones)
- The zones are available from

```
Set<String> allZonesIds = ZoneId.getAvailableZoneIds();

String ukTZ = ZoneId.of("Europe/London");
```

7th Concept: Zoned Time

- How to create a zoned time

```
System.out.println(
   ZonedDateTime.of(
      1564, Month.APRIL.getValue(), 23, // year / month / day
      10, 0, 0, 0,                      // h / mn / s / nanos
      ZoneId.of("Europe/London"))
); // prints 1564-04-23T10:00-00:01:15[Europe/London]
```

7th Concept: Zoned Time

- ZonedDateTime exposes a set of methods to compute other zoned times : plus, minus, with, etc…

```
ZonedDateTime currentMeeting =
   ZonedDateTime.of(
      LocalDate.of(2014, Month.MARCH, 12), // LocalDate
      LocalTime.of(9, 30),                 // LocalTime
      ZoneId.of("Europe/London")
);

ZonedDateTime nextMeeting =
   currentMeeting.plus(Period.ofMonth(1));
```

- And to change the time zone:

```
ZonedDateTime nextMeetingUS =
   nextMeeting.withZoneSameInstant(ZoneId.of("US/Central"));
```

How to Format a Date

- The new date API proposes a new formatter: DateTimeFormatter
- The DateTimeFormatter proposes a set of predefined formatters, available as constants

```
ZonedDateTime nextMeetingUS =
   nextMeeting.withZoneSameInstant(ZoneId.of("US/Central"));

System.out.println(
   DateTimeFormatter.ISO_DATE_TIME.format(nextMeetingUS)
);
// prints 2014-04-12T03:30:00-05:00[US/Central]

System.out.println(
   DateTimeFormatter.RFC_1123_DATE_TIME.format(nextMeetingUS)
);
// prints Sat, 12 Apr 2014 03:30:00 -0500
```

Bridges Between the APIs

- How to interoperate with the legacy Date API?
- Instant & Date:

```
Date date = Date.from(instant);     // legacy -> new API
Instant instant = date.toInstant(); // API -> legacy
```

- Instant & TimeStamp:

```
TimeStamp time = TimeStamp.from(instant); // legacy -> new API
Instant instant = time.toInstant();       // API -> legacy
```

- LocalDate & Date :

```
Date date = Date.from(localDate);         // legacy -> new API
LocalDate localDate = date.toLocalDate(); // API -> legacy
```

- LocalTime & Time

```
Time time = Time.from(localTime);         // legacy -> new API
LocalTime localTime = time.toLocalTime(); // API -> legacy
```

Summary

- The new Date API from Java 8 fixes the issues of Java 7
- The new concepts of « date » in Java
- The new concepts of « duration » in Java
- How to compute a new date from a given date
- How to deal with time zones
- How to format date following the established standards

Strings, I/O and Other Bits & Pieces

Many Useful Little Things

José Paumard
blog.paumard.org
@JosePaumard

pluralsight
hardcore dev and IT training

Module Outline

- Java 8 is not only about Lambdas and Streams
- The String class
- The Java I/O package
- Collection interface
- Comparators
- Numbers
- Maps
- Annotations

Strings in Java 8

Creating a Stream on a String

- A new method on the String class

```
String s = "Hello world!";
Stream stream = s.chars(); // creates a Stream on the
                           // letters of s
stream.map(String::toUpperCase)
      .forEach(System.out::print);
```

- Will print:

```
> HELLO WORLD!
```

The StringJoiner

- Concatenation of Strings is not that simple!

```
String s1 = "Hello";
String s2 = "world";

String s = s1 + " " + s2; // it works!
```

- Some people will tell you:
- « it's not efficient, and should not be used! »
- « because of the multiple creations / deletions of intermediary strings »

The StringJoiner

- Concatenating Strings is not that simple!

```
StringBuffer sb1 = new StringBuffer();
sb1.append("Hello");
sb1.append(" ").append("world"); // can be chained
String s = sb1.toString();
```

- Better but StringBuffer is synchronized

The StringJoiner

- Concatenating Strings is not that simple!

```
// The JDK 5 way
StringBuilder sb1 = new StringBuilder();
sb1.append("Hello");
sb1.append(" ").append("world"); // can be chained
String s = sb1.toString();
```

- Better!
- In fact, this is the way the JDK7 compiles String concatenations

The StringJoiner

- Much simpler in JDK 8 with the StringJoiner!
- A StringJoiner is built with a separator

```
// The JDK 8 way
StringJoiner sj = new StringJoiner(", ");
sj.add("one").add("two").add("three");
String s = sj.toString();
System.out.println(s);
```

- Will print:

```
> one, two, three
```

The StringJoiner

- Much simpler in JDK 8 with the StringJoiner!
- A StringJoiner can also be built with a separator, a prefix and a postfix

```
// The JDK 8 way
StringJoiner sj = new StringJoiner(", ", "{", "}");
// we leave the joiner empty
String s = sj.toString();
System.out.println(s);
```

- Will print:

```
> {}
```

The StringJoiner

- Much simpler in JDK 8 with the StringJoiner!
- A StringJoiner can also be built with a separator, a prefix and a postfix

```
// The JDK 8 way
StringJoiner sj = new StringJoiner(", ", "{", "}");
sj.add("one");
String s = sj.toString();
System.out.println(s);
```

- Will print:

```
> {one}
```

The StringJoiner

- Much simpler in JDK 8 with the StringJoiner!
- A StringJoiner can also be built with a separator, a prefix and a postfix

```
// The JDK 8 way
StringJoiner sj = new StringJoiner(", ", "{", "}");
sj.add("one").add("two").add("three");
String s = sj.toString();
System.out.println(s);
```

- Will print:

```
> {one, two, three}
```

The StringJoiner

- The StringJoiner can be used from the String class

```
// From the String class, with a vararg
String s = String.join(", ", "one", "two", "three");
System.out.println(s);
```

- Will print:

```
> one, two, three
```

The StringJoiner

- The StringJoiner can be used from the String class

```
// From the String class, with an Iterable
String [] tab = {"one", "two", "three"};
String s = String.join(", ", tab);
System.out.println(s);
```

- Will print:

```
> one, two, three
```

Java I/O enhancements

Reading Text Files

- A lines() method has been added on the BufferedReader class

```
// Java 7 : try with resources
try (BufferedReader reader =
        new BufferedReader(
        new FileReader(
        new File("d:/tmp/debug.log")));) {

   Stream<String> stream = reader.lines();
   stream.filter(line -> line.contains("ERROR"))
         .findFirst()
         .ifPresent(System.out::println);

} catch (IOException ioe) {
   // handle the exception
}
```

Reading Text Files

- Method File.lines(path)

```
// Java 7 : try with resources and use of Paths
Path path = Paths.get("d:", "tmp", "debug.log");
try (Stream<String> stream = Files.lines(path)) {

   stream.filter(line -> line.contains("ERROR"))
         .findFirst()
         .ifPresent(System.out::println);

} catch (IOException ioe) {
   // handle the exception
}
```

- Stream implements AutoCloseable, and will close the underlying file

Reading Directory Entries

- Method File.list(path)

```
// Java 7 : try with resources and use of Paths
Path path = Paths.get("c:", "windows");
try (Stream<Path> stream = Files.list(path)) {

   stream.filter(path -> path.toFile().isDirectory())
         .forEach(System.out::println);

} catch (IOException ioe) {
   // handle the exception
}
```

- Visits the first level entries

Reading Directory Entries

- To visit the whole subtree use the Files.walk(path) method

```
// Java 7 : try with resources and use of Paths
Path path = Paths.get("c:", "windows");
try (Stream<Path> stream = Files.walk(path)) {

   stream.filter(path -> path.toFile().isDirectory())
         .forEach(System.out::println);

} catch (IOException ioe) {
   // handle the exception
}
```

Reading Directory Entries

- To visit the whole subtree use the Files.walk(path) method

```
// Java 7 : try with resources and use of Paths
Path path = Paths.get("c:", "windows");
try (Stream<Path> stream = Files.walk(path, 2)) {

   stream.filter(path -> path.toFile().isDirectory())
         .forEach(System.out::println);

} catch (IOException ioe) {
   // handle the exception
}
```

- One can limit the depth of the exploration

New Methods on the Collection API

- Of course, the most important : stream() and parallelStream()
- Also : spliterator()

New Method on Iterable

- Method forEach()

```
// Unfortunately not for arrays
List<String> strings =
   Arrays.asList("one", "two", "three");

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

New Methods on Collection

- Method removeIf(), returns a boolean

```
// removes an element on a predicate
Collection<String> strings =
   Arrays.asList("one", "two", "three", "four");

// will not work if list is unmodifiable
Collection<String> list = new ArrayList<>(strings);

// returns true if the list has been modified
boolean b = list.removeIf(s -> s.length() > 4);

System.out.println(
   list.stream().collect(Collectors.joining(", ")));
```

New Methods on Collection

- Method removeIf(), returns a boolean

```
// removes an element on a predicate
Collection<String> strings =
   Arrays.asList("one", "two", "three", "four");

// will not work if list is unmodifiable
Collection<String> list = new ArrayList<>(strings);

// returns true if the list has been modified
boolean b = list.removeIf(s -> s.length() > 4);

System.out.println(
   list.stream().collect(Collectors.joining(", ")));
```

- Will print:

```
> one, two, four
```

New Methods on List

- Method replaceAll()

```
// removes an element on a predicate
List<String> strings =
   Arrays.asList("one", "two", "three", "four");

// will not work if list is unmodifiable
List<String> list = new ArrayList<>(strings);

// doesnt return anything
list.replaceAll(String::toUpperCase);

System.out.println(
   list.stream().collect(Collectors.joining(", ")));
```

New Methods on List

- Method replaceAll()

```
// removes an element on a predicate
List<String> strings =
   Arrays.asList("one", "two", "three", "four");

// will not work if list is unmodifiable
List<String> list = new ArrayList<>(strings);

// doesnt return anything
list.replaceAll(String::toUpperCase);

System.out.println(
   list.stream().collect(Collectors.joining(", ")));
```

- Will print:

```
> ONE, TWO, THREE, FOUR
```

New Methods on List

- Method sort()

```
// removes an element on a predicate
List<String> strings =
   Arrays.asList("one", "two", "three", "four");

// will not work if list is unmodifiable
List<String> list = new ArrayList<>(strings);

// doesnt return anything
list.sort(Comparator.naturalOrder());

System.out.println(
   list.stream().collect(Collectors.joining(", ")));
```

New Methods on List

- Method sort()

```
// removes an element on a predicate
List<String> strings =
   Arrays.asList("one", "two", "three", "four");

// will not work if list is unmodifiable
List<String> list = new ArrayList<>(strings);

// doesnt return anything
list.sort(Comparator.naturalOrder());

System.out.println(
   list.stream().collect(Collectors.joining(", ")));
```

- Will print

```
> four, one, three, two
```

Comparators

New Way to Write a Comparator

- The JDK 7 way:

```
// comparison using the last name
Comparator<Person> compareLastName =
   new Comparator<Person>() {

      @Override
      public int compare(Person p1, Person p2) {
         return p1.getLastName().compareTo(p2.getLastName());
      }
  };
```

- It would also need to check if p1 or p2 is null

New Way to Write a Comparator

- The JDK 7 way:

```
// comparison using the last name then the first name
Comparator<Person> compareLastNameThenFirstName =
   new Comparator<Person>() {

      @Override
      public int compare(Person p1, Person p2) {
         int lastNameComparison =
            p1.getLastName().compareTo(p2.getLastName());
         return lastNameComparison == 0 ?
            p2.getFirstName().compareTo(p2.getFirstName());
            lastNameComparison;
      }
  };
```

- Same remark!

New Way to Write a Comparator

- The JDK 8 way:

```
// comparison using the last name
Comparator<Person> compareLastName =
   Comparator.comparing(Person::getLastName);
```

New Way to Write a Comparator

- The JDK 8 way:

```
// comparison using the last name
Comparator<Person> compareLastName =
   Comparator.comparing(Person::getLastName);
```

- comparing() is a static method of the interface Comparator

New Way to Write a Comparator

- The JDK 8 way:

```
// comparison using the last name and then the first name
Comparator<Person> compareLastNameThenFirstName =
   Comparator.comparing(Person::getLastName)
             .thenComparing(Person::getFirstName);
```

- thenComparing() is a default method of the interface Comparator

Other Useful Utilities

- How to reverse a given comparator?

```
// reverses a comparator
Comparator<Person> comp = ...;

Comparator<Person> reversedComp = comp.reversed();
```

Other Useful Utilities

- The natural comparator

```
// compares comparable objects
Comparator<String> c = Comparator.naturalOrder();
```

Other Useful Utilities

- The natural comparator

```
// compares comparable objects
Comparator<String> c = Comparator.naturalOrder();
```

- The reversed natural comparator

```
// compares comparable objects in the reverse order
Comparator<String> c = Comparator.reversedOrder();
```

Other Useful Utilities

- And what about null values?

```
// considers null values lesser than non-null values
Comparator<String> c =
   Comparator.nullsFirst(Comparator.naturalOrder());
```

Other Useful Utilities

- And what about null values?

```
// considers null values lesser than non-null values
Comparator<String> c =
   Comparator.nullsFirst(Comparator.naturalOrder());
```

- And of course…

```
// considers null values greater than non-null values
Comparator<String> c =
   Comparator.nullsLast(Comparator.naturalOrder());
```

Numbers

A Few Points on Numbers

- Primitive types: byte, short, char, int, long, double, float and boolean
- They all got a wrapper type

New Methods on the Number Types

- New useful methods: sum, max, min

```
long max = Long.max(1L, 2L);
```

- Useful to create reduction operations

```
BinaryOperator<Long> sum = (l1, l2) -> l1 + l2;
                         = (l1, l2) -> Long.sum(l1, l2);
                         = Long::sum;
```

A Few Points on Numbers

- Hash code computation

```
// JDK 7
long l = 31415926535897932381;
int hash =  new Long(l).hashCode(); // -1985256439
```

A Few Points on Numbers

- Hash code computation

```
// JDK 7
long l = 31415926535897932381;
int hash =  new Long(l).hashCode(); // -1985256439
```

- Costly boxing / unboxing to compute this hash code

A Few Points on Numbers

- Hash code computation

```
// JDK 7
long l = 31415926535897932381;
int hash =  new Long(l).hashCode(); // -1985256439
```

- Costly boxing / unboxing to compute this hash code

```
// JDK 8
long l = 31415926535897932381;
int hash =  Long.hashCode(l); // - 1985256439
```

- This method is available on the 8 wrapper types

Maps

New Methods on Map

- Method forEach()

```
Map<String, Person> map = ...;
map.forEach((key, person) ->
   System.out.println(key + " " + person);
```

- Takes a BiConsumer as a parameter

New Methods on Map

- Method get()

```
Map<String, Person> map = ...;


Person p = map.get(key); // p can be null!
```

New Methods on Map

- Method get()

```
Map<String, Person> map = ...;

Person defaultPerson = Person.DEFAULT_PERSON;
Person p = map.getOrDefault(key, defaultPerson); // JDK 8
```

- Returns the default value passed as a parameter if there is no value in the map

New Methods on Map

- Method put()

```
Map<String, Person> map = ...;

map.put(key, person); // will erase an existing person
```

New Methods on Map

- Method put()

```
Map<String, Person> map = ...;

map.put(key, person);
map.putIfAbsent(key, person); // JDK8
```

- Will not erase an existing person

New Methods on Map

- Method replace()

```
Map<String, Person> map = ...;

map.replace(key, person);
```

- Replaces an existing person

New Methods on Map

- Method replace()

```
Map<String, Person> map = ...;

map.replace(key, person);
map.replace(key, oldPerson, newPerson);
```

- Replaces oldPerson by newPerson

New Methods on Map

- Method replace()

```
Map<String, Person> map = ...;

map.replace(key, person);
map.replace(key, oldPerson, newPerson);

map.replaceAll((key, oldPerson) -> newPerson);
```

- Applies the remapping function to all the existing key / person pairs

New Methods on Map

- Method remove()

```
Map<String, Person> map = ...;
map.remove(key);
```

New Methods on Map

- Method remove()

```
Map<String, Person> map = ...;
map.remove(key);          // JDK 7
map.remove(key, person); // JDK 8
```

- Removes a key / person value

New Methods on Map

- Method compute(), computeIfPresent() , computeIfAbsent()

```
Map<String, Person> map = ...;

map.compute(key, person, (key, oldPerson) -> newPerson);
```

- Returns the computed value

New Methods on Map

- Method compute(), computeIfPresent() , computeIfAbsent()

```
Map<String, Person> map = ...;

map.computeIfPresent(key, person, (key, oldPerson) -> newPerson);
```

- Returns the computed value

New Methods on Map

- Method compute(), computeIfPresent() , computeIfAbsent()

```
Map<String, Person> map = ...;

map.computeIfAbsent(key, key -> newPerson);
```

- Returns the computed value

New Methods on Map

- Method compute(), computeIfPresent() , computeIfAbsent()

```
Map<String, Person> map = ...;

map.computeIfAbsent(key, key -> newPerson);
```

- Returns the computed value
- Useful to create bimaps

```
Map<String, Map<Integer, Person>> bimap = ...;
Person p = ...;


bimap.computeIfAbsent(key1, key -> new HashMap<>()).put(key2, p);
```

New Methods on Map

- Method merge()

```
Map<String, Person> map = ...;

map.merge(key, person, (key, person) -> newPerson);
```

- Associates a key not present in the map, or associated to a null value, to a new value

Annotations

Annotations

- Java 8 brings the concept of « multiple annotations »
- Suppose we want to test this case with several parameters
- Java 7 solution: wrap the annotation

```
@TestCases({
   @TestCase(param=1, expected=false),
   @TestCase(param=2, expected=true)
})
public boolean even(int param) {
   return param % 2 == 0;
}
```

Annotations

- Java 8 brings the concept of « multiple annotations »
- Suppose we want to test this case with several parameters
- Java 7 solution: wrap the annotation

```
@TestCases({
   @TestCase(param=1, expected=false),
   @TestCase(param=2, expected=true)
})
public boolean even(int param) {
   return param % 2 == 0;
}
```

- Because an annotation cannot be applied twice on the same element

Annotations

- Java 8 brings the concept of « multiple annotations »
- Suppose we want to test this case with several parameters
- Java 8 solution

```
@TestCase(param=1, expected=false)
@TestCase(param=2, expected=true)
public boolean even(int param) {
   return param % 2 == 0;
}
```

Annotations

- Java 8 brings the concept of « multiple annotations »
- Suppose we want to test this case with several parameters
- Java 8 solution

```
@TestCase(param=1, expected=false)
@TestCase(param=2, expected=true)
public boolean even(int param) {
   return param % 2 == 0;
}
```

- Annotations become « repeatable »

Annotations

- How does it work?
- The wrapping annotation is automatically added for us
- First, create the annotations as usual

Annotations

- How does it work?
- The wrapping annotation is automatically added for us
- First, create the annotations as usual

```
@interface TestCase {
   int param();
   boolean expected();
}
```

```
@interface TestCases {
   TestCase[] value();
}
```

Annotations

- How does it work?
- The wrapping annotation is automatically added for us
- First, create the annotations as usual
- Then add the @Repeatable annotation on the wrapped annotation

```
@Repeatable(TestCases.class)
@interface TestCase {
   int param();
   boolean expected();
}
```

```
@interface TestCases {
   TestCase[] value();
}
```

Type Annotations

- Java 8 allows annotations to be put on types
- Example 1: to declare that a variable should not be null

```
private @NonNull List<Person> persons = ... ;
```

Type Annotations

- Java 8 allows annotations to be put on types
- Example 1: to declare that a variable should not be null

```
private @NonNull List<Person> persons = ... ;
```

- Example 2: to declare that a list should not be null, and should not contain null values

```
private @NonNull List<@NonNull Person> persons = ... ;
```

Summary

- The String class, StringJoiner
- Easy ways to create streams on text files
- Simple ways to visit directories
- New methods on Iterable, Collection and List
- New patterns to create Comparator
- Useful methods on the number wrapper classes
- New methods on Map
- How to use and create repeatable annotations

Introduction to Java FX 8

A New Framework to Design GUI in Java

José Paumard
blog.paumard.org
@JosePaumard

pluralsight
hardcore dev and IT training

Module Outline

- A first and simple example: scene, stage
- Layout
- Designing a GUI using the JavaFX API
- Using an FXML file
- Dependency injection in a JavaFX controller
- Catching events in callbacks

A Simple Example

- Create a class that extends Application, and overrides start()
- Call the launch() method on that class

A Simple Example

- Create a class that extends Application, and overrides start()
- Call the launch() method on that class

```
import javafx.application.Application;

public class FirstApplication extends Application {

   public void start(Stage stage) {
      // callback
   }

   public static void main(String... args) {
      launch();
   }
}
```

A Simple Example

- Then add some content

```
public void start(Stage stage) {
   // a simple UI
   Label message = new Label("Hello world!");
   message.setFont(new Font(100));

   stage.setScene(new Scene(message));
   stage.setTitle("Hello");
   stage.show();
}
```

A Few Key Concepts

- Stage: « top-level window »
- A Stage can be a top-level window
- A Stage can be a rectangular area in the case of an applet
- A Stage can be the full screen itself

A Few Key Concepts

- On our example:
- The Label is added to a Scene
- The Scene is added to the Stage
- And we call the show() method on the stage

A Layout Example

- Layout: can hold several components

```
public void start(Stage stage) {
   // javacontrol.Label
   Label message1 = new Label("Hello world!");
   message1.setFont(new Font(100));

   Label message2 = new Label("Hello world!");
   message2.setFont(new Font(100));
   // java.scene.layout.VBox
   VBox vbox = new VBox(message1, message2);

   stage.setScene(new Scene(vbox));
   stage.setTitle("Hello");
   stage.show();
}
```

A Login Window Example – XML Version

- Can also be designed in a XML file

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.geometry.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>
<?import javafx.scene.paint.*?>

<GridPane hgap="10" vgap="10">

   <!-- content of the grid pane -->

</GridPane>
```

A Login Window Example – XML Version

- Can also be designed in a XML file

```
<GridPane hgap="10" vgap="10">

   <padding>
      <Insets bottom="10.0" top="10.0"
              left="10.0" right="10.0" />
   </padding>

   <children>
      <!-- children components -->
   </children>
</GridPane>
```

A Login Window Example – XML Version

- Can also be designed in a XML file

```
<children>
   <Label text="User name:" />
   <TextField />
</children>
```

A Login Window Example – XML Version

- Can also be designed in a XML file

```
<children>
   <Label text="User name:" />
   <TextField id="username"/>
</children>
```

A Login Window Example – XML Version

- Can also be designed in a XML file

```
<children>
   <Label text="User name:"
          GridPane.columnIndex="0" GridPane.rowIndex="0" />
   <TextField id="username"
          GridPane.columnIndex="1" GridPane.rowIndex="0" />
</children>
```

A Login Window Example – XML Version

- Can also be designed in a XML file

```
<children>
   <Label text="User name:"
          GridPane.columnIndex="0" GridPane.rowIndex="0"
          GridPane.halignment="RIGHT" />
   <TextField id="username"
          GridPane.columnIndex="1" GridPane.rowIndex="0" />
</children>
```

A Login Window Example – XML Version

- Can also be designed in a XML file

```
<children>
   <Label text="User name:"
          GridPane.columnIndex="0" GridPane.rowIndex="0"
          GridPane.halignment="RIGHT" />
   <TextField id="username"
          GridPane.columnIndex="1" GridPane.rowIndex="0" />
   <Label text="Password:"
          GridPane.columnIndex="0" GridPane.rowIndex="0"
          GridPane.halignment="RIGHT" />
   <PasswordField id="password"
          GridPane.columnIndex="1" GridPane.rowIndex="0" />
</children>
```

A Login Window Example – XML Version

- Can also be designed in a XML file

```
<children>
   <!-- labels + textfields -->
   <HBox >
      <children>
         <Button text="Ok" />
         <Button text="Cancel" />
      </children>
   </HBox>
</children>
```

A Login Window Example – XML Version

- Defining the ID attributes

```
<GridPane hgap="10" vgap="10"
          xmlns:fx="http://javafx.com/fxml"
          fx:controller="org.paumard.javafx.MyController">

   <children>
      <Label text="User name:" />
      <TextField fx:id="username"/>
   </children>

   <HBox >
      <children>
         <Button text="Ok" onAction="#okAction"/>
      </children>
   </HBox>
   <!-- rest of the UI -->

</GridPane>
```

A Login Window Example – XML Version

- The controller class

```
public class MyController implements Initializable {

    @FXML
    private TextField username;

    @Override
    public void initialize(URL url, ResourceBundle bundle) {
    }

    public void okAction(ActionEvent event) {

        System.out.println("Clicked ok");
        System.out.println("user name = " +
                           username.getText());
    }
}
```

A Login Window Example – XML Version

- The Application class

```
public class MyApplication extends Application {
   @Override
   public void start(Stage stage) {
      try {
         FXMLLoader loader = new
            FXMLLoader(getClass().getResource("ihm.fxml"));

         Parent root = loader.load();
         stage.setScene(new Scene(root));
         stage.show();
      } catch (IOException ioe) {
         // ...
      }
   }

   public static void main(String... args) {
      launch();
   }
}
```

And There Is More

- Supports CSS for customizing the look and feel of the GUI
- A rich animation API for moving, scaling, rotating etc… components
- Support for touch interfaces
- Works on many types of displays
- Compatible with Swing (to a certain extent)

Summary

- Quick overview of Java FX 8
- How to create basic interfaces
- Building an interface with the API or FXML
- Dependency injection on GUI components
- Callbacks on simple events

Nashorn: a JavaScript Engine on the JVM

A JavaScript Engine for the JVM

José Paumard
blog.paumard.org
@JosePaumard

pluralsight
hardcore dev and IT training

Module Outline

- REPL: Java in JavaScript
- ScriptEngine: Java in JavaScript
- JavaScript and JavaFX

A JavaScript REPL

What Is a REPL?

- REPL = Read, Eval, Print Loop
- It looks like a shell, ie with a prompt
- And enables one to type in JavaScript interactively
- jjs is the REPL executable
- It is located in $JAVA_HOME/bin, in the same place as javac or java

What Is a REPL?

- REPL = Read, Eval, Print Loop
- It looks like a shell, ie with a prompt
- And enables one to type in JavaScript interactively

What Is a REPL?

- REPL = Read, Eval, Print Loop
- It looks like a shell, ie with a prompt
- And enables one to type in JavaScript interactively

```
jjs> 'Hello world!'.length()
12
jjs>
```

What Is a REPL?

- REPL = Read, Eval, Print Loop
- It looks like a shell, ie with a prompt
- And enables one to type in JavaScript interactively

```
jjs> function fibo(n) { return n <= 1 ? n : n + fibo(n - 1) }
function fibo(n) { return n <= 1 ? n : n + fibo(n - 1) }
jjs>fibo(100)
5050
jjs>
```

What Is a REPL?

- REPL = Read, Eval, Print Loop
- It looks like a shell, ie with a prompt
- And enables one to type in JavaScript interactively

```
jjs> function fibo(n) { return n <= 1 ? n : n + fibo(n - 1) }
function fibo(n) { return n <= 1 ? n : n + fibo(n - 1) }
jjs>fibo(100)
5050
jjs>function fact(n) { return n <= 1 ? n : n*fact(n - 1) }
function fact(n) { return n <= 1 ? n : n*fact(n - 1) }
jjs>fact(5)
120
jjs>
```

What Is a REPL?

- One can create Java objects and interact with them

```
jjs>var s = new java.lang.String("Hello")
jjs>s
Hello
jjs>
```

What Is a REPL?

- One can create Java objects and interact with them

```
jjs>var s = new java.lang.String("Hello")
jjs>s
Hello
jjs>s.toUpperCase()
HELLO
jjs>
```

The REPL

- The Nashorn REPL allows to interactively type in an execute Java and JavaScript

Running JavaScript in Java code

Running JavaScript in a Java Application

- Java has been supporting script engines since Java 6 (2006)
- Many languages are available: Groovy and JRuby
- One needs to get a script engine by its name

```
ScriptEngineManager manager = new ScriptEngineManager();

ScriptEngine engine = manager.getEngineByName("nashorn");
```

- This object is used to interact with the JavaScript interpreter

```
Object result = engine.eval("/* JavaScript code here */");
```

- One can also pass JavaScript code through a file

```
Object result = engine.eval(Files.newBufferedReader(path));
```

How to Pass Objects to JavaScript

- Two ways of passing Java objects to the JavaScript engine
- Suppose we want to pass the Stage object (from JavaFX)
- 1st solution:

```
public void start(Stage stage) {
   engine.put("stage", stage);
   engine.eval(script); // script is my JavaScript code
}
```

- In this case the stage variable is available in the JavaScript « global scope »

How to Pass Objects to JavaScript

- Two ways of passing Java objects to the JavaScript engine
- Suppose we want to pass the Stage object (from JavaFX)
- 2nd solution: we want to scope our variable

```
Bindings scope = engine.createBindings();
scope.put("stage", stage);
engine.eval(script, scope);
```

- In this case the stage variable is only available in the scope defined by the scope object

Invoking Getters and Setters

- The JavaFX stage object has a property named title
- In Java a property = a getter and a setter
- In JavaScript one can write this:

```
stage.setTitle('This is JavaScript!')
```

- But also this:

```
stage.title = 'This is JavaScript!'
```

Invoking Getters and Setters

- The JavaFX stage object has a property named title
- In Java a property = a getter and a setter
- In JavaScript one can write this:

```
stage.setTitle('This is JavaScript!')
```

- But also this:

```
stage.title = 'This is JavaScript!'
```

- And also this:

```
stage['title'] = 'This is JavaScript!'
```

Nashorn and JavaFX

Lauching a JavaFX Application Through Nashorn

- One can use jjs to launch a JavaFX application

```
$ jjs -fx myJavaFXApp.js
```

- Nashorn will make the stage object available through $STAGE

```
var message =
   new javafx.scene.control.Label("This is JavaScript!");
message.font =
   new javafx.scene.text.Font(100);
$STAGE.scene = new javafx.scene.Scene(message);
$STAGE.title = "Hello World!";
```

Summary

- **Quick overview of Java / JavaScript integration using Nashorn**
- **How to type in JavaScript code through the REPL jjs**
  - How to use Java objects and classes in the JavaScript code
- **How to evaluate JavaScript code in a Java application**
  - How to pass Java objects in the JavaScript code
- **How to create a JavaFX application using JavaScript**

Course Summary

- **Lamba expressions**
  - Anonymous class, functional interfaces, method references, collection API
- **Streams & Collectors**
  - Map / filter / reduce, patterns to build a stream, operations on a Stream
- **Java Date & Time API**
  - Instance / Duration, LocalDate / Period, LocalTime, Zoned Time
- **Strings, I/O, and other Bits & Pieces**
  - Strings, I/O, Collection, Comparators, Numbers, Maps, Annotations
- **Java FX**
- **Nashorn and JavaScript**