Higher-Order Functional Programming with Java 8 Streams

Florian Biermann
fbie@itu.dk

IT University of Copenhagen & UCAS

2016-06-02

IT University
of Copenhagen

Today's Plan

We will be looking at (parallel) higher order functional programming in Java.

Today's Plan

We will be looking at (parallel) higher order functional programming in Java.

- Brief review of Java's anonymous inner classes.

Today's Plan

We will be looking at (parallel) higher order functional programming in Java.

- Brief review of Java's anonymous inner classes.
- Java 8 lambda expressions.

Today's Plan

We will be looking at (parallel) higher order functional programming in Java.

- Brief review of Java's anonymous inner classes.
- Java 8 lambda expressions.
- Java 8 functional interfaces.

Today's Plan

We will be looking at (parallel) higher order functional programming in Java.

- Brief review of Java's anonymous inner classes.
- Java 8 lambda expressions.
- Java 8 functional interfaces.
- Higher-order functional programming using lazy streams.

Java 7 Anonymous Inner Classes

In Java, you can instantiate new classes on the fly:

```
Runnable r =
   new Runnable() {
      public void run() {
         System.out.println("Inner class.");
      }};
r.run(); // "Inner class."
```

Java 7 Anonymous Inner Classes

In Java, you can instantiate new classes on the fly:

```
Runnable r =
   new Runnable() {
      public void run() {
         System.out.println("Inner class.");
      }};
r.run(); // "Inner class."
```

They can access global variables iff these are marked `final`:

```
final String s = "This is final!";
Runnable r = new Runnable() {
      public void run() {
         System.out.println(s);
      }};
r.run(); // "This is final!";
```

Runnable Is Just an Interface

```
public interface Runnable {
  public void run();
}
```

Runnable Is Just an Interface

```
public interface Runnable {
  public void run();
}
```

There are other useful interfaces, e.g. `Callable<V>` from `java.util.concurrent`:

```
public interface Callable<V> {
  public V call();
}
```

Runnable Is Just an Interface

```
public interface Runnable {
  public void run();
}
```

There are other useful interfaces, e.g. `Callable<V>` from `java.util.concurrent`:

```
public interface Callable<V> {
  public V call();
}

final String s = "Callable.";
Callable<String> c = new Callable<String>() {
    public String call() {
      return s;
    }};
System.out.println(c.call()); // "Callable."
```

Why Anonymous Inner Classes?

The Java answer to closures.

Why Anonymous Inner Classes?

The Java answer to closures.

- Whenever we want to start a thread, we pass it a new instance of `Runnable`.

Why Anonymous Inner Classes?

The Java answer to closures.

- Whenever we want to start a thread, we pass it a new instance of `Runnable`.
- Short-running tasks, only used once.

Why Anonymous Inner Classes?

The Java answer to closures.

- Whenever we want to start a thread, we pass it a new instance of `Runnable`.
- Short-running tasks, only used once.
- Methods in Java are no *first-class citizens*, but objects are.

Why Anonymous Inner Classes?

The Java answer to closures.

- Whenever we want to start a thread, we pass it a new instance of `Runnable`.
- Short-running tasks, only used once.
- Methods in Java are no *first-class citizens*, but objects are.
- Anonymous inner classes are a *work-around* for this problem.

Why Anonymous Inner Classes?

The Java answer to closures.

- Whenever we want to start a thread, we pass it a new instance of `Runnable`.
- Short-running tasks, only used once.
- Methods in Java are no *first-class citizens*, but objects are.
- Anonymous inner classes are a *work-around* for this problem.
- The syntax is ugly and hard to read, so Java 8 introduces *lambda expressions.*

Our First Java 8 Lambda Expression

Equal to instantiating an anonymous `Runnable`:

```
Runnable r =
  () -> { System.out.println("Lambda."); } ;
```

Our First Java 8 Lambda Expression

Equal to instantiating an anonymous `Runnable`:

```
Runnable r =
  () -> { System.out.println("Lambda."); } ;
```

No closing braces required if only one statement:

```
Runnable r =
  () -> System.out.println("Lambda.");
```

Our Second Java 8 Lambda Expression

Equal to instantiating an anonymous `Callable<String>`:

```
Callable<String> c =
   () -> { return "Lambda."; };
```

Our Second Java 8 Lambda Expression

Equal to instantiating an anonymous `Callable<String>`:

```
Callable<String> c =
  () -> { return "Lambda."; };
```

No `return` statement required if only one statement:

```
Callable<String> c = () -> "Lambda.";
```

Our Second Java 8 Lambda Expression

Equal to instantiating an anonymous `Callable<String>`:

```
Callable<String> c =
  () -> { return "Lambda."; };
```

No `return` statement required if only one statement:

```
Callable<String> c = () -> "Lambda.";
```

Compare to Typed Racket, also no `return` statement:

```
(lambda () "Lambda.")
```

This Is Illegal

```
Callable<String> c =
  () -> {
    System.out.println("Callable.call()");
    "Lambda.";
  };
```

This Is Illegal

```
Callable<String> c =
   () -> {
     System.out.println("Callable.call()");
     "Lambda.";
   };
```

```
error: not a statement
        "Lambda.";};
        ^
```

This Is Illegal

```
Callable<String> c =
   () -> {
     System.out.println("Callable.call()");
     "Lambda.";
   };
```

```
error: not a statement
         "Lambda.";};
         ^
```

Rules:

This Is Illegal

```
Callable<String> c =
   () -> {
     System.out.println("Callable.call()");
     "Lambda.";
   };
```

```
error: not a statement
         "Lambda.";};
         ^
```

Rules:

- If you use more than one statement then you must use curly braces.

This Is Illegal

```
Callable<String> c =
   () -> {
     System.out.println("Callable.call()");
     "Lambda.";
   };
```

```
error: not a statement
         "Lambda.";};
         ^
```

Rules:

- If you use more than one statement then you must use curly braces.
- If you use curly braces then you *must* use `return` (except for `void` functions).

Java Inner Classes Summary

The important points until here:

Java Inner Classes Summary

The important points until here:

- Inner classes always implement an interface (or an abstract class).

Java Inner Classes Summary

The important points until here:

- Inner classes always implement an interface (or an abstract class).
- There are two important interfaces already defined in Java 7.

Java Inner Classes Summary

The important points until here:

- Inner classes always implement an interface (or an abstract class).
- There are two important interfaces already defined in Java 7.
- In Java 8, you can use *lambda expressions* instead of inner classes.

Java Inner Classes Summary

The important points until here:

- Inner classes always implement an interface (or an abstract class).
- There are two important interfaces already defined in Java 7.
- In Java 8, you can use *lambda expressions* instead of inner classes.
- But what about more interesting functions that also take parameter arguments?

Defining Your Own Functional Interfaces

A function of type `int` $\rightarrow$ `String`:

```
public interface StringToInt {
  public int apply(String s);
}
```

Defining Your Own Functional Interfaces

A function of type `int` $\rightarrow$ `String`:

```
public interface StringToInt {
  public int apply(String s);
}
```

Now, we can instantiate such a function using a lambda expression:

```
StringToInt parseToInt =
  s -> Integer.parseInt(s);
```

Defining Your Own Functional Interfaces

A function of type `int` $\rightarrow$ `String`:

```
public interface StringToInt {
  public int apply(String s);
}
```

Now, we can instantiate such a function using a lambda expression:

```
StringToInt parseToInt =
  s -> Integer.parseInt(s);
```

- Lambda expressions **have no types themselves!**

Defining Your Own Functional Interfaces

A function of type `int` $\rightarrow$ `String`:

```
public interface StringToInt {
  public int apply(String s);
}
```

Now, we can instantiate such a function using a lambda expression:

```
StringToInt parseToInt =
  s -> Integer.parseInt(s);
```

- Lambda expressions **have no types themselves!**
- Instead, they have a *target function type*.

Defining Your Own Functional Interfaces

A function of type `int` $\rightarrow$ `String`:

```
public interface StringToInt {
  public int apply(String s);
}
```

Now, we can instantiate such a function using a lambda expression:

```
StringToInt parseToInt =
  s -> Integer.parseInt(s);
```

- Lambda expressions **have no types themselves!**
- Instead, they have a *target function type*.
- Without target function type, the lambda expression does not work.

Defining Your Own Functional Interfaces

A function of type `int` $\rightarrow$ `String`:

```
public interface StringToInt {
  public int apply(String s);
}
```

Now, we can instantiate such a function using a lambda expression:

```
StringToInt parseToInt =
  s -> Integer.parseInt(s);
```

- Lambda expressions **have no types themselves!**
- Instead, they have a *target function type*.
- Without target function type, the lambda expression does not work.
- Here, one target function type is `StringToInt`.

Java 8 Functional Interfaces

No need to define your own interfaces all the time.

`java.util.function` contains *many* different functional interfaces:

Java 8 Functional Interfaces

No need to define your own interfaces all the time.

`java.util.function` contains *many* different functional interfaces:

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

Java 8 Functional Interfaces

No need to define your own interfaces all the time.
`java.util.function` contains *many* different functional interfaces:

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

Predicate<String> isEmpty =
  s -> s.isEmpty();
```

Java 8 Functional Interfaces

No need to define your own interfaces all the time.
`java.util.function` contains *many* different functional interfaces:

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

Predicate<String> isEmpty =
  s -> s.isEmpty();

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

Java 8 Functional Interfaces

No need to define your own interfaces all the time.
`java.util.function` contains *many* different functional interfaces:

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

Predicate<String> isEmpty =
  s -> s.isEmpty();

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

Supplier<String> genString =
  () -> "Nihao.";
```

Primitive Data Type Interfaces

To avoid *value boxing* (`int` $\rightarrow$ `Integer`), there are special implementations for primitive data types:

Primitive Data Type Interfaces

To avoid *value boxing* (`int` $\rightarrow$ `Integer`), there are special implementations for primitive data types:

```
ToIntFunction<String> parseToInt =
  s -> Integer.parseInt(s);
```

Primitive Data Type Interfaces

To avoid *value boxing* (`int` → `Integer`), there are special implementations for primitive data types:

```
ToIntFunction<String> parseToInt =
  s -> Integer.parseInt(s);

LongBinaryOperator longMul =
  (x, y) -> x * y;
```

Primitive Data Type Interfaces

To avoid *value boxing* (`int` → `Integer`), there are special implementations for primitive data types:

```
ToIntFunction<String> parseToInt =
  s -> Integer.parseInt(s);

LongBinaryOperator longMul =
  (x, y) -> x * y;

DoubleToIntFunction round =
  x -> (int)(x + 0.5d);
```

Primitive Data Type Interfaces

To avoid *value boxing* (`int` → `Integer`), there are special implementations for primitive data types:

```
ToIntFunction<String> parseToInt =
  s -> Integer.parseInt(s);

LongBinaryOperator longMul =
  (x, y) -> x * y;

DoubleToIntFunction round =
  x -> (int)(x + 0.5d);

IntPredicate isEven =
  x -> x % 2 == 0;
```

Function References

This is redundant syntax:

```
ToIntFunction<String> parseToInt =
  s -> Integer.parseInt(s);
```

Function References

This is redundant syntax:

```
ToIntFunction<String> parseToInt =
  s -> Integer.parseInt(s);
```

We call a function that has the same type as the target function type, `String` $\rightarrow$ `int`. Instead, we can use a *function reference*:

Function References

This is redundant syntax:

```
ToIntFunction<String> parseToInt =
  s -> Integer.parseInt(s);
```

We call a function that has the same type as the target function type, `String` $\rightarrow$ `int`. Instead, we can use a *function reference*:

```
ToIntFunction<String> parseToInt =
  Integer::parseInt;
```

This makes the code much more concise!

Functional Interfaces Summary

The important points until here:

Functional Interfaces Summary

The important points until here:

- In Java 8, you can use *lambda expressions* instead of inner classes.

Functional Interfaces Summary

The important points until here:

- In Java 8, you can use *lambda expressions* instead of inner classes.
- You can define your own *target function types* by defining interfaces.

Functional Interfaces Summary

The important points until here:

- In Java 8, you can use *lambda expressions* instead of inner classes.
- You can define your own *target function types* by defining interfaces.
- Better: use pre-defined functional interfaces from `java.util.function.*`

Functional Interfaces Summary

The important points until here:

- In Java 8, you can use *lambda expressions* instead of inner classes.
- You can define your own *target function types* by defining interfaces.
- Better: use pre-defined functional interfaces from `java.util.function.*`
- If you work on primitive data types, use specialized interfaces!

Functional Interfaces Summary

The important points until here:

- In Java 8, you can use *lambda expressions* instead of inner classes.
- You can define your own *target function types* by defining interfaces.
- Better: use pre-defined functional interfaces from `java.util.function.*`
- If you work on primitive data types, use specialized interfaces!
- So what can we use these for?

Java 8 Streams

Streams in Java 8 are lazy collections of values:

```
Stream<Student> students =
  Stream.of(new Student("A", 89),
            new Student("B", 44),
            new Student("C", 62));
```

Java 8 Streams

Streams in Java 8 are lazy collections of values:

```
Stream<Student> students =
  Stream.of(new Student("A", 89),
            new Student("B", 44),
            new Student("C", 62));
```

We can map over streams:

```
students.map(s -> s.improveGrade(10));
```

Java 8 Streams

Streams in Java 8 are lazy collections of values:

```
Stream<Student> students =
  Stream.of(new Student("A", 89),
            new Student("B", 44),
            new Student("C", 62));
```

We can map over streams:

```
students.map(s -> s.improveGrade(10));
```

- Functional: `map` produces a new `Stream<Student>` instance!

Java 8 Streams

Streams in Java 8 are lazy collections of values:

```
Stream<Student> students =
  Stream.of(new Student("A", 89),
            new Student("B", 44),
            new Student("C", 62));
```

We can map over streams:

```
students.map(s -> s.improveGrade(10));
```

- Functional: `map` produces a new `Stream<Student>` instance!
- Lazy: the computation is not executed but saved for later.

Java 8 Streams

Streams in Java 8 are lazy collections of values:

```
Stream<Student> students =
  Stream.of(new Student("A", 89),
            new Student("B", 44),
            new Student("C", 62));
```

We can map over streams:

```
students.map(s -> s.improveGrade(10));
```

- Functional: `map` produces a new `Stream<Student>` instance!
- Lazy: the computation is not executed but saved for later.
- Requires a *terminal operation* to trigger computation.

Java 8 Streams

Streams in Java 8 are lazy collections of values:

```
Stream<Student> students =
  Stream.of(new Student("A", 89),
            new Student("B", 44),
            new Student("C", 62));
```

We can map over streams:

```
students.map(s -> s.improveGrade(10));
```

- Functional: `map` produces a new `Stream<Student>` instance!
- Lazy: the computation is not executed but saved for later.
- Requires a *terminal operation* to trigger computation.

```
students
  .map(s -> s.improveGrade(10))
  .forEach(System.out::println);
```

What is Laziness?

Deferring computation until value is requested.

```
public class Lazy<T> implements Supplier<T> {
  private final Supplier<T> f;
  private volatile T t;

  public Lazy(Supplier<T> f) {
    this.f = f;
  }

  public T get() {
    if (t == null)
      t = f.get();
    return t;
  }}
```

Laziness in Action

```
Lazy<String> s = new Lazy<String>(() -> {
    System.out.println("Calling get()");
    return "someString";
  }); // Not yet computed!

String s1 = s.get(); // "Calling get()"
String s2 = s.get(); // Nothing printed.
```

How Does Laziness Help?

Laziness allows the `Stream` implementation to *fuse* successive applications of `map`. This is an in-place mapping:

```
for (int i = 0; i < students.length; ++i)
  students[i] = f(students[i]);
for (int i = 0; i < students.length; ++i)
  students[i] = g(students[i]);
```

How Does Laziness Help?

Laziness allows the `Stream` implementation to *fuse* successive applications of `map`. This is an in-place mapping:

```
for (int i = 0; i < students.length; ++i)
  students[i] = f(students[i]);
for (int i = 0; i < students.length; ++i)
  students[i] = g(students[i]);
```

This is what loop fusion does:

```
for (int i = 0; i < students.length; ++i)
  students[i] = g(f(students[i]));
```

Computing Number of Primes

The Java 7 way:

```
long count = 0;
for (long p = 0; p < n; ++p)
  if (isPrime(p)) ++count;
```

Computing Number of Primes

The Java 7 way:

```
long count = 0;
for (long p = 0; p < n; ++p)
  if (isPrime(p)) ++count;
```

Java 8 Streams:

```
LongStream.range(0, n)
  .filter(p -> isPrime(p))
  .count();
```

Computing Number of Primes

The Java 7 way:

```
long count = 0;
for (long p = 0; p < n; ++p)
  if (isPrime(p)) ++count;
```

Java 8 Streams:

```
LongStream.range(0, n)
  .filter(p -> isPrime(p))
  .count();
```

Parallel streams:

```
LongStream.range(0, n)
  .parallel()
  .filter(p -> isPrime(p))
  .count();
```

Java Streams Overview

Java streams are a great tool:

- Very well implemented interface.
- Easy to use when you are used to functional programming.
- Easy to parallelize (`.parallel()`).

Java Streams Overview

Java streams are a great tool:

- Very well implemented interface.
- Easy to use when you are used to functional programming.
- Easy to parallelize (`.parallel()`).

But the downsides...

- Sometimes, performance is unpredictable because of laziness.
- OBS: Functions with side-effects will break!