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 »

Module Outline

- Introduction to the « Lambda expressions »
- The lambda syntax

Module Outline

- Introduction to the « Lambda expressions »
- The lambda syntax
- Functional interfaces

Module Outline

- Introduction to the « Lambda expressions »
- The lambda syntax
- Functional interfaces
- Method references
- Constructor references

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") ;
   }
}
```

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

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

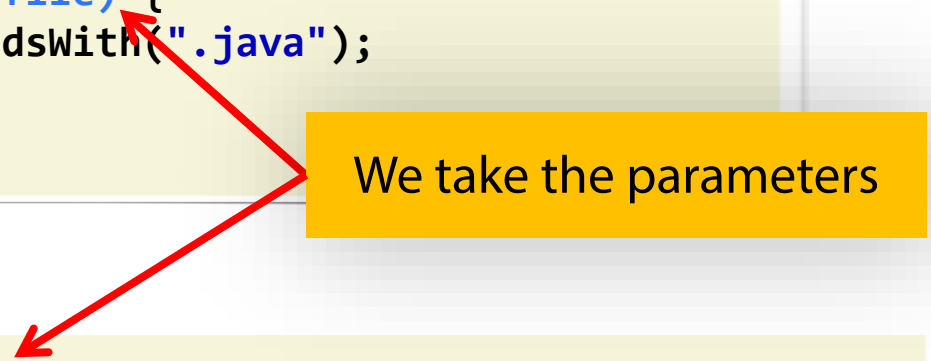
- Let's use an anonymous class

```
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");
    }
};
```




```
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");
   }
};
```

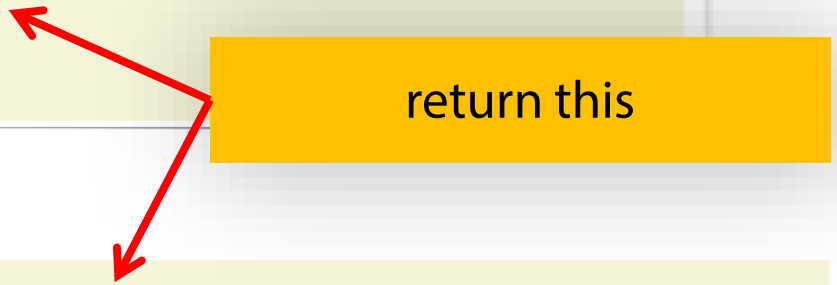
and then…

```
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");
    }
};
```




```
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");
```

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?

Three Questions About Lambdas

- What is the type of a lambda expression?

- Can a lambda be put in a variable?

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 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

Functional Interface

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

```
public interface Runnable {

   run();
};
```

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);
};
```

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());
```

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

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

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

   void accept(T t);
}
```

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

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

   boolean test(T t);
}
```

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

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

   R apply (T t);
}
```

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

So What Do We Have so Far?

- A new concept: the « lambda expression », with a new syntax
- A new interface concept: the « functional interface »

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?

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)

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));
```

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!

Can I Process This Data with Lambdas?

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

```
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?

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);
}
```

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

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!

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…

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;
```

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);
```

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);
```

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