# Workshop
# ***Hands-on with Java 8***

*by*

*Nanne Baars / Urs Peter*
*nbaars@xebia.com / upeter@xebia.com*


# New Java 8 Language Features covered in this Workshop

- JSR 335
  - Lambda Expressions and Virtual Extension Methods
- JSR 310
  - Date and Time API

# Lambda's in a Nutshell

```
//Before Java8:
File file = new File("/tmp");
file.listFiles(new FileFilter() {
        @Override
        public boolean accept(File pathname) {
              return pathname.isDirectory();
         }
 });
```

Anonymous inner classes are history

```
//Lambda Expression
file.listFiles((File f) -> f.isDirectory());

//Lambda with Type Inference (omit input type)
file.listFiles(f -> f.isDirectory());

//Method reference syntax
file.listFiles(File::isDirectory);
```

Use **Lamda's** or **Method References** to create instances of **Functional Interfaces**

# @FunctionalInterfaces in package java.util.function:

A Bunch of new **Functional Interfaces** were added to Java 8:

```
Supplier<String> supplier = () -> "JFall Rocks";
supplier.get();                                  //Returns: JFall Rocks
```

```
Consumer<Object> consumer = o -> System.out.println(o);
consumer.accept("Java 8 Rocks");          //Returns: Java 8 Rocks
```

```
Function<Integer, Integer> function= i -> i * i;
function.apply(4);                               //Returns: 16
```

```
Predicate<File> predicate = f -> f.isDirectory();
predicate.test(new File("/tmp"));          //Returns: true
```

# Smooth Collections with Streams

```
List<Integer> input = Arrays.asList(1,2,3);
```

```
List<Integer> result = new ArrayList<>();
for(Integer i:input) {
    if(i % 2 == 0) {
            result.add(i);
    }
}
Collections.sort(result);
```

Say goodbye to For-loops…

… say hello to **Streams**:

```
input.stream()
        .filter(i -> i % 2 == 0)
        .sorted()
        .collect(Collectors.toList());
```

# Streams in a Nutshell

```
List<Integer> input = Arrays.asList(1,2,3);

input.stream()
        .filter(i -> i % 2 == 0)
        .sorted()
        .collect(Collectors.toList());
```

**stream()** is a new method available on **java.util.Collection**

**java.util.stream.Stream** offers a variety of methods to do data transformations with Lambda's. E.g. filter, map, sort etc.

Use **collect()** to convert a Stream to a Collection type

**java.util.stream.Collectors** offers a variety of useful reduction operations

# Some Cool Methods on Stream: Can you figure out the results?

```
List<Integer> input = Arrays.asList(1,2,3);

input.stream().filter(i -> i % 2 != 0).collect(Collectors.toList());

input.stream().map(i -> i + 1).collect(Collectors.toList());

input.stream().forEach(System.out::print);

input.stream().reduce((i, j) -> i + j);

input.stream().collect(Collectors.groupingBy(i -> i % 2));

input.stream().allMatch(i -> i > 4);

input.stream().anyMatch(i -> i == 100);
```

# Some Cool Methods on Stream: Can you figure out the results?

```
List<Integer> input = Arrays.asList(1,2,3);

input.stream().filter(i -> i % 2 != 0).collect(Collectors.toList());
- result: List(1,3)
input.stream().map(i -> i + 1).collect(Collectors.toList());
- result: List(2,3,4)
input.stream().forEach(System.out::print);
- result: "123"
input.stream().reduce((i, j) -> i + j);
- result: 6
input.stream().collect(Collectors.groupingBy(i -> i % 2));
- result: Map(0 -> List(2), 1 -> List(1,3))
input.stream().allMatch(i -> i > 4);
- result: true
input.stream().anyMatch(i -> i == 100);
- result: false
```

# Default Methods: Interfaces with default Implementations

```
public interface Collection<E> extends Iterable<E> {
    default Stream<E> stream() {
        return StreamSupport.stream(spliterator(), false);
    }
}
```

Interfaces methods with the **default** keyword…

…can contain an **implementation**.

The class that implements this interface *does not have to* (but can) implement default methods.

**Defaults Methods** are great for **backwards compatibility** of interfaces and **multiple inheritance**!

```
public class MyCollection<E> implements Collection<E> {
...
}
```

# Highest time for: java.time

```
import java.time.*;
LocalDate today = LocalDate.now();
LocalDateTime now= LocalDateTime.now();
now.format(DateTimeFormatter.BASIC_ISO_DATE);
```

LocalDate/Time are *immutable* objects with a *thread-safe* TimeFormatter

```
LocalDate aDate =
        MonthDay.of(Month.DECEMBER, 31).atYear(Year.now()).plusDays(1);
```

Java.time offers a DSL to create LocalDate/Time

```
Period p = Period.between(aDate, LocalDate.now());
long millis = ChronoUnit.MILLIS.between(now, now);
```

Helper classes to calculate with Dates and Time

# LABS

- com.xebia.java8_1.lambdas
  - Introduction to Lambda syntax
- com.xebia.java8_2.functions
  - Introduction to Functional Interfaces in package java.util.function
- com.xebia.java8_3.collections
  - Using java.util.stream.Streams
- com.xebia.java8_4.functionalpatterns
  - Leverage your code with functional patterns
- com.xebia.java8_5.defaultmethods
  - Apply multiple inheritance with default methods
- com.xebia.java8_6.datetime
  - Introduction to java.time
- com.xebia.java8_7.infinite_list
  - Advanced lab for working with infinite lists

# Bootstrap

- Copy contents of memory stick to your machine
- Follow instruction on Readme
- If you have no IDE with Java8 IntelliJ is provided for Windows, Mac & Linux
- No internet connection is needed

- Have fun!!!