

Module Outline

- **Java 8 is not only about Lambdas and Streams**

Module Outline

- **Java 8 is not only about Lambdas and Streams**
- **The String class**

Module Outline

- **Java 8 is not only about Lambdas and Streams**
- **The String class**
- **The Java I/O package**

Module Outline

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

Module Outline

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

Module Outline

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

Module Outline

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

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!";
```

Creating a Stream on a String

- A new method on the String class

```
String s = "Hello world!";  
IntStream stream = s.chars(); // creates a Stream on  
                               // the letters of s
```

Creating a Stream on a String

- A new method on the String class

```
String s = "Hello world!";  
IntStream stream = s.chars(); // creates a Stream on  
                               // the letters of s  
stream.mapToObj(letter -> (char)letter)  
       .map(Character::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!
```

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");
```

The StringJoiner

- Concatenating Strings is not that simple!

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


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();
```

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!

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");
```

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);
```

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
}
```

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
}
```

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



Collection API

New Methods on the Collection API

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

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());
        }
    };
};
```

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 = 3141592653589793238L;  
int hash = new Long(l).hashCode(); // -1985256439
```

A Few Points on Numbers

- Hash code computation

```
// JDK 7  
long l = 3141592653589793238L;  
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 = 3141592653589793238L;  
int hash = new Long(l).hashCode(); // -1985256439
```

- Costly boxing / unboxing to compute this hash code

```
// JDK 8  
long l = 3141592653589793238L;  
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 bimap

```
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?

Annotations

- How does it work?
- The wrapping annotation is automatically added for us

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