Java 8 training

Java 8 - new features and benefits overview, functional interfaces

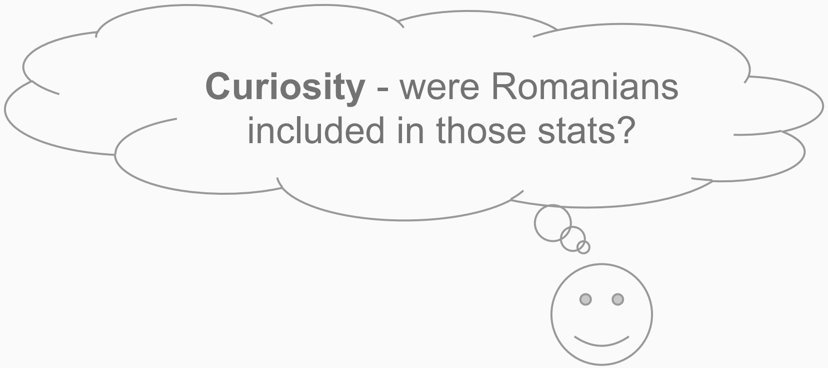
Overview

- **Java 8**:
  - New features overview
  - Features and benefits
- Functional interfaces - overview and hands-on

Java 8 stats (JM, Sept 2016 & JetBrains June 2018)

- 84% of developers are now using Java 8
  - October 2014 - 27%
  - May 2015 - 38%
  - August 2015 - 58%
  - March 2016 - 64%

- Features usage (% of developers):
  - Lambdas - 73%
  - Stream API - 70%
  - Optional - 51%



Why are we (RO) *not* using Java 8 (yet)?

- Legacy projects
- Lack of time (to learn / adapt / refactor)
- Others?

Your opinions?

Java 8 - new features

- **Functional interfaces** - single-method interfaces, implemented as lambda expressions
  - Main functional interfaces - `Predicate`, `Consumer`, `Function`, `Supplier`, **Bi***

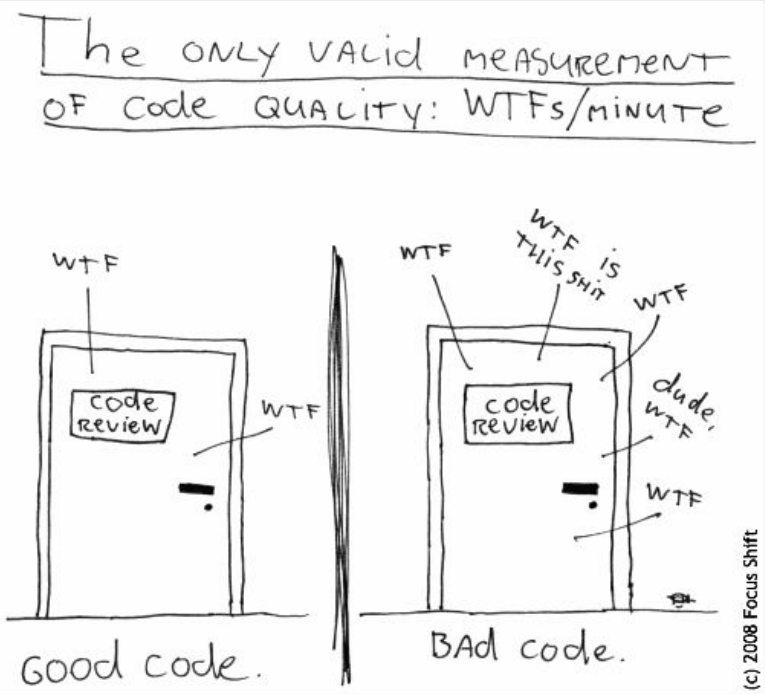
- **Lambda expressions** (λ) - functionalities passed as method arguments
- **Default methods** in interfaces
- **Method references (::)** - easy to read and use expressions for:
  - Constructors with a single / no parameters Class::new
  - Methods with a single / no parameters Class::method

- **`Optional`** class - wrapper for cleaner / improved `null` references handling
- **Streams API** for arrays and collections → simplified & chained processing
- New **Date and Time API**
- **`CompletableFuture`** - async processing stages

Benefits

- Cleaner and more concise code
- Streams:
  - Pipelined operations - monadic processing
  - Built-in map-reduce operations
- `Optional` - improved and cleaner handling for `null` references
  - Less encounters of our `NullPointerException` friend :)
- Improved async processing:
  - Streams processing can be parallelized
  - `CompletableFuture` - async processing stages
- Improved date and time handling

‘The only valid code quality metric’



Java 8 **improves the code quality,** through:

- *Functional programming* → lambda expressions
    - Inline methods become ‘first class citizens’
    - Less code, less bugs
- *Streams*:
    - Chained and optional processing stages
    - Out of the box conversions → collectors
- Built in and standardized ‘map reduce’ flows
- Out of the box *parallelization*, where applicable
- Improved `null` references handling → `Optional` class

Method and constructor references

- Simplified form for calling **methods with a single / no parameter**
- Represented through the ‘::’ syntax
- Used for:
  - Methods `Class::method`
  - `this` `this::method`
  - Constructors `Class::new`

Examples:

- `Integer::parseInt` // a `String` param is implied for the ‘parseInt’ method
- `System.out::println` // a `String` is implied for the parameter
- `ArrayList::new` // creating a new `ArrayList`

Default methods in interfaces

- Interfaces can have `'default'` methods → non-abstract methods, aka 'extension methods'
- Example:

```
public interface ProductProcessor() {
    void processProduct(Product product);

    default void lazyProcessing(Product product) {
        product.process();
    }
}
```

Default methods in interfaces - reasoning

- Extending the functionality of the existing JDK interfaces - adding support for functional interfaces-based methods
    - `Collection`
    - `List`
    - `Map`
    - ...
- No need for an explicit implementation in each implementing class
    - Can be implemented, if / when needed
- The only reasonable mean for adding logic in functional interfaces

Imperative vs functional programming

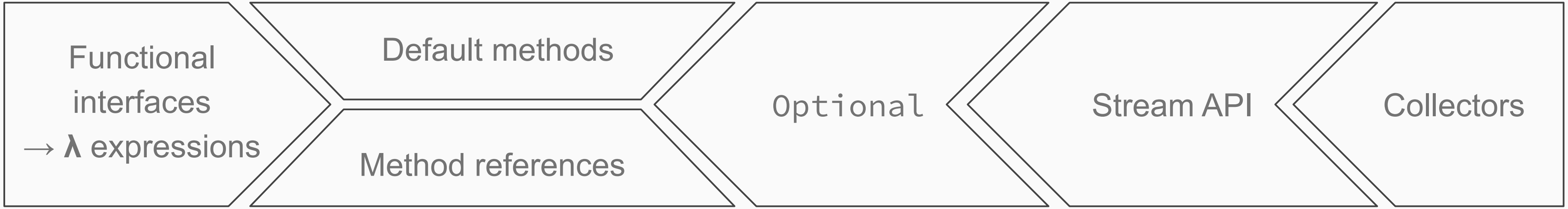
- *Imperative* → telling the computer *how* to run a program (comm. sequence)
  - More difficult to maintain and read
  - More difficult to parallelize
- *Functional* → telling the computer *what* to do, not how
  - The runtime machine will determine *how* to run them
  - More scalable programs
  - More succinct and maintainable

→ Allows the developers to *focus more on what*

Imperative vs functional - recommendation

- Do *not* consider functional programming a ‘golden hammer’
  - There are contexts where imperative programming is better suited
- The ‘best solution’: a mix of the two paradigms - use:
  - *Functional* programming where its idioms are well suited
  - *Imperative* programming where:
    - Is easier to understand and debug
      - Especially for developers with less functional programming experience
    - Will be easier to maintain / extend → open / closed principle

Java 8 - functional programming enablers



Functional interfaces

- **Single abstract method interfaces**, annotated with @**FunctionalInterface** (opt)
  - Core enabler for functional programming - using (inline) lambda expressions
  - Any interface with a single abstract method can be inferred as functional interface

- Main functional interfaces:
  - **`Predicate`** - boolean valued function of one argument
    - `boolean test(T t);` // tests a condition on the object of type T, returns true/false
  - **`Consumer`** - accepts a single input argument, does not return anything
    - `void accept(T t);` // processes an object of type T
  - **`Function`** - accepts one argument, returns a result
    - `R apply(T t);` // processes an object of type T, returns an object of type R
  - **`Supplier`** - supplier of results
    - `T get();` // no requirement for a new / distinct result

Main functional interfaces - contracts

Predicate<T>	• boolean test(T t)
Consumer<T>	• void accept(T t)
Function<T, R>	• R apply(T t)
Supplier<T>	• T get()

Functional interfaces - default method usage

- Functional interfaces - used by a lot of JDK 8 classes & interfaces:
    - `Optional`
    - `Stream`
    - Collections hierarchy - `Collection, Map, List, Set`
    - ... many others...
- Their abstract method can be invoked:
    - Implicitly (/ auto) - by JDK methods which use a functional interface as parameter
        - The most frequent usage mode
    - Explicitly - when / if needed

Functional interfaces -> lambda expressions

- **Lambda expressions** - inline implementations of functional interfaces
- Syntax: parameter name -> body
- Optional characteristics:
  - Parameter types
  - Parentheses around parameters → mandatory for >1 parameters
  - Curly braces and / or `return` statement → statement lambda expressions
- No parameters → empty parentheses () must be provided

```
stringsList.removeIf(string -> string.isEmpty());
```

// removes the empty strings

Lambda expressions

- Can use multiple named parameters:

```
Map<String, String> mapEntries = new HashMap<>(); // + add items
mapEntries.forEach((key, value) -> list(key + “,“ + value));
```

- Can use method references, where applicable:

```
List<String> items = new ArrayList<>();    // + add operations
items.forEach(System.out::println);        // inferred argument
```

Predicate

- Boolean returning method → ‘`test(Type type)`’ method

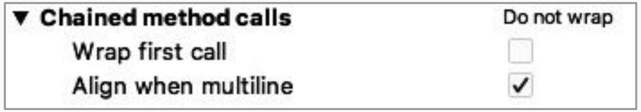
```
Predicate<Integer> isEven = value -> value % 2 == 0;
```

- Chaining logical operators - `.and(), .or(), .negate()`

```
Predicate<Integer> biggerThan10 = value -> value > 10;

boolean isEvenAndBiggerThan10 = isEven.and(biggerThan10)
                                      .test(12);
```

Hands-on - s01e02 - using `Predicates`



Consumer

- Operations applied on **a single input argument**
- Applied by the `'void accept(Type t)'` method
- Example:

```
Consumer<String> display = it -> System.out.println(it);
display.accept("A Consumer example");
```

Hands-on - s01e03 - using Consumers

Consumer - default methods

- `.andThen(Consumer c)` - chaining another `Consumer` of the *same* type

Example:

```
Consumer<Integer> preProcessor = number -> preProcess(number);
Consumer<Integer> processor = number -> process(number);
preProcessor.andThen(processor).accept(25);
```

Function

- **Converts** an input T into an output R → ‘`R apply(T t)`’
- Example:

```
Function<Integer, String> display = it -> "It is: " + it;
System.out.println(display.apply(5));
```

- Allows **chaining other (same type) functions** → ‘andThen’ and ‘compose’
  - ‘`andThen`’ - applied *after* the caller
  - ‘`compose`’ - applied *before* the caller

Hands-on - s01e04 - using Functions

Existing inferred functional interfaces

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

```
interface Callable<T> {
    T call();
}
```

```
interface Comparable<T> {
    int compareTo(T t);
}
```

```
interface Comparator<T> {
    int compare(T o1, T o2);
}
```

Comparable and Comparator as functional interfaces

- Comparable & Comparator interfaces:

```
interface Comparable<T> {
    int compareTo(T t);
}
```

```
interface Comparator<T> {
    int compare(T o1, T o2);
}
```

- Can be inferred as functional interfaces:

```
Comparable<Product> comparable = prod -> prod.getPrice();

Comparator<Product> comparator = (first, second) ->
                                    first.getId() - second.getId();
```

Supplier

- **Creates an object of a given type** (hence called 'supplier') → '`T get()`'
- Example:

```
Supplier<String> newStringSupplier = String::new;
Supplier<String> stringSupplier = () -> "some";
```

// ( ) = method with no parameters

Hands-on - s01e05 - using Suppliers

Bi* functional interfaces

Processors of multiple types:

- `BiPredicate   → boolean test(T t, U u)`
- `BiFunction    → R apply(T t, U u)`
- `BiConsumer    → void accept(T t, U u)`

Where:

- T = the 1st type
- U = the 2nd type

Hands-on - s01e06 - using Bi* functional interfaces

UnaryOperator, BinaryOperator

- **UnaryOperator** → Function applied on the same type: Function<T, T>

```
UnaryOperator<Integer> square = input -> input * input;
```

- **BinaryOperator** → BiFunction applied on a type: BiFunction<T, T, T>

```
BinaryOperator<Double> squareRoot = (a, b) -> Math.sqrt(a * b);
```

Hands-on - s01e07 - using unary & binary operators

Typed functional interfaces

`Predicate`, `Consumer`, `Function`, `Supplier`, `UnaryOperator` and `BinaryOperator` for:

- `Double`
- `Int`
- `Long`

Hands-on - s01e08 - using the typed functional interfaces

Q & A session

- You ask, I answer
- I ask, you answer