Functional Style Programming in JAVA 8

Daniel Kocsis
2016. December

A brief history of Java

Java SE 5 - 2004

- <Generics>
- @Annotations
- Autoboxing/unboxing
- ENUMERATIONS
- Varargs…
- `for each` loop
- Static import
- New Concurrent utilities

Java SE 6 - 2006

- Scripting language support
- Performance improvements
- JAX-WS (Java API for XML web services [SOAP])
- JDBC 4.0
- Java Compiler API
- JAXB 2.0 and StAX
- Pluggable annotations
- New GC algorithms

Java SE 7 - 2011

- JVM support for dynamic languages
- Compressed 64-bit pointers
- Language changes
  - Strings in switch
  - Automatic resource management in try-statement
  - The diamond operator
  - Simplified varargs method declaration
  - Binary integer literals
  - Underscores in numeric literals
  - Improved exception handling (a.k.a multi catch)
- ForkJoin Framework
- NIO 2.0 having support for multiple file systems, file metadata and symbolic links
- WatchService
- Timsort is used to sort collections and arrays of objects instead of merge sort
- APIs for the graphics features
- Support for new network protocols, including SCTP and Sockets Direct Protocol

Java SE 8 - 2014

- **Lambda expression support in APIs**
- **Functional interface and default methods**
- **Optionals**
- Nashorn – JavaScript runtime which allows developers to embed JavaScript code within applications
- Annotation on Java Types
- Unsigned Integer Arithmetic
- Repeating annotations
- New Date and Time API
- Statically-linked JNI libraries
- Launch JavaFX applications from jar files
- Remove the permanent generation from GC

Java SE 9 - 2017?

- Language Changes
  - Private Interface (Default) Methods
  - Try-With-Resources on Effectively Final Variables
  - Diamond Operator for Anonymous Classes
  - SaveVarargs on Private Methods
  - No More Deprecation Warnings for Imports
- APIs
  - OS Processes
  - Stack Walking
  - **Reactive Streams**
  - HTTP/2
  - Extensions to Existing APIs, e.g.
    - Optional, Stream, and Collectors
    - DateTime API
    - Matcher
    - Atomic…
    - Array utilities
- And many more, see the complete list here or a good summary here
- You can download the early access release of the JDK9 from here

So what is this functional programming thing?

“In computer science, functional programming is a programming paradigm - a style of building the structure and elements of computer programs - that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data.”

- Wikipedia

Common Concepts - not necessarily all apply to every FP language!

- Focus on what is to be computed rather than how to compute it
- Based on Lambda Calculus (typed or untyped)
- First-class(Pure) and higher-order functions
  - Currying and partial application
- Pure functions
  - Immutability
- Referential transparency
- Recursion
- Strict(eager) vs Non-Strict(lazy) evaluation
- Type system (mostly statically typed and strongly typed)
- Special (immutable) data types

Other swear words from the FP world

- Closures
- Lambdas
- Functors
- Applicatives
- Monads
- Pattern matching (language level)

What is a Closure?

It has a scientific definition of course :) - see this or this Wiki article

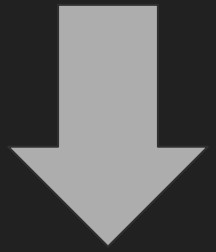
In programming:

“A closure is a persistent scope which holds on to local variables even after the code execution has moved out of that block”

Closure = Function pointer + Stack frame

What is a lambda expression?

$$f(x) = e$$



$$f = \lambda x.e$$

What is a Lambda expression for programmers?

- For a little bit of science see this or the related Wikipedia article
- In computer programming, an anonymous function (function literal, **lambda**) is a function definition that is not bound to an identifier. Anonymous functions are often:
  - arguments being passed to higher-order functions, or
  - used for constructing the result of a higher-order function that needs to return a function.
- Simply put:
  - ***Lambda expressions (in programming languages) are an expression whose value is a function***


PARENTAL
ADVISORY
EXPLICIT CONTENT

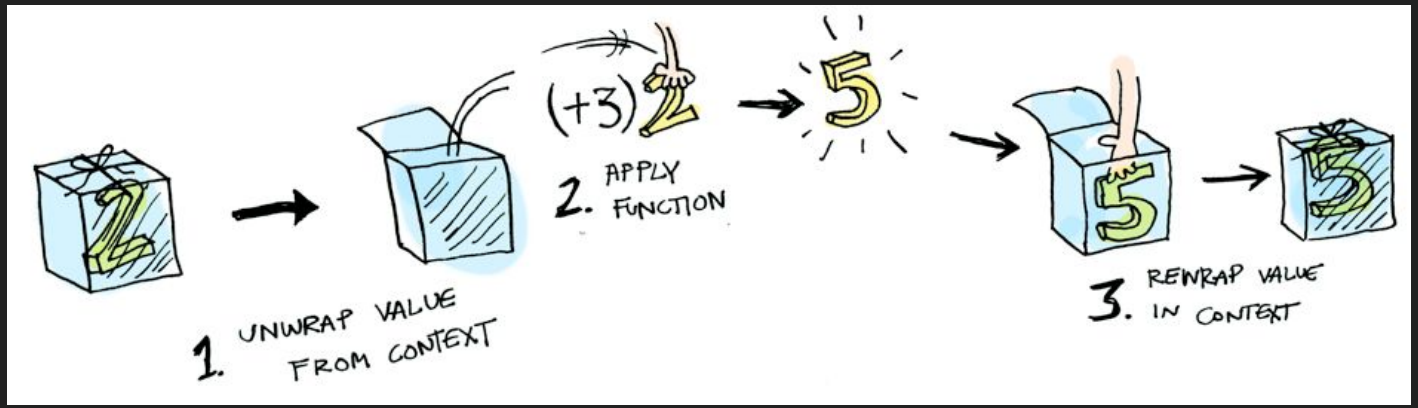
What is a Functor?

Problem: We have a simple value, but it is wrapped into a context, now how to apply a function on it?

Solution: Functor!

A Functor is any data type that defines how map applies to it

Simply put: **Functor is something that knows how to apply a function to a simple value wrapped in a context**



http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html

Functors in JAVA

```
public interface Functor<T, F> {
  public <R> Functor<R, F> map(Function<T, R> function);
}

class Identity<T> implements Functor<T,Identity<?>> {

    private final T value;

    Identity(T value) { this.value = value; }

    public <R> Identity<R> map(Function<T,R> f) {
        final R result = f.apply(value);
        return new Identity<>(result);
    }

}
```

```
Identity<byte[]> idBytes = new Identity<>(customer)
        .map(Customer::getAddress)
        .map(Address::street)
        .map((String s) -> s.substring(0, 3))
        .map(String::toLowerCase)
        .map(String::getBytes);
```

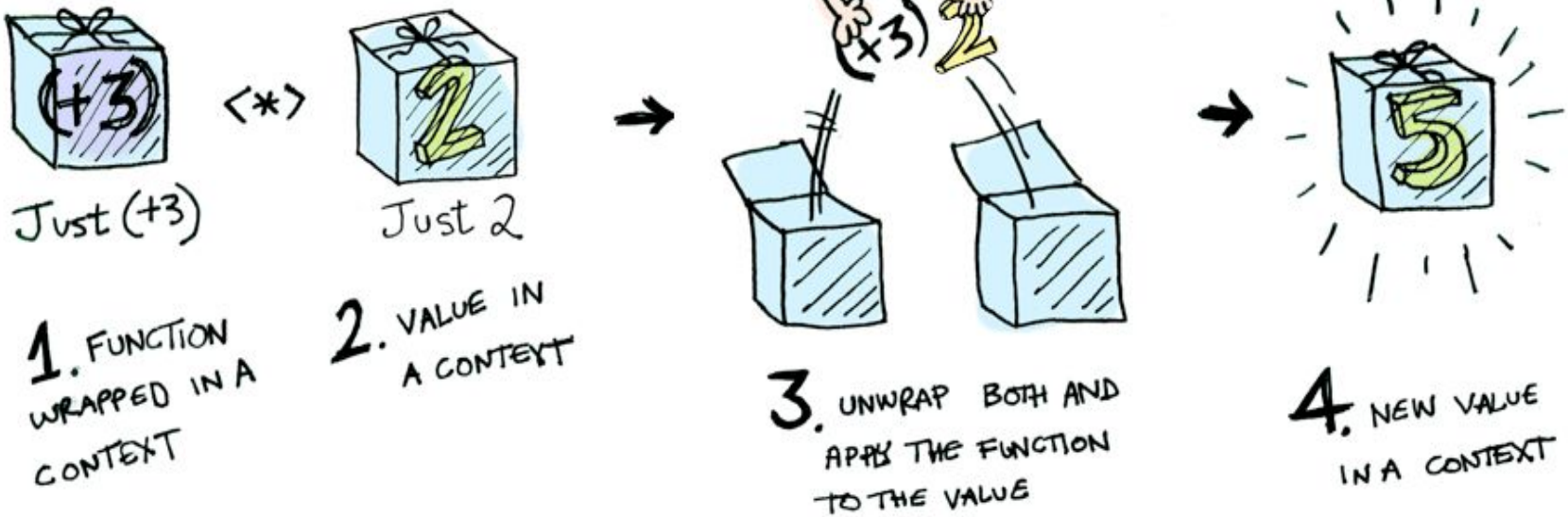
What is an Applicative?

Problem: We have a value wrapped in context and functions are wrapped in a context too! How to apply these functions to the value?

Solution: Applicative!

Applicatives apply takes a functor that has a function in it and another functor and extracts that function from the first functor and then maps it over the second one.

Simply put: Applicatives apply a wrapped function to a wrapped value


(+3)
<*>
2
Just (+3)
Just 2
1. FUNCTION WRAPPED IN A CONTEXT
2. VALUE IN A CONTEXT
(+3)
2
3. UNWRAP BOTH AND APPLY THE FUNCTION TO THE VALUE
5
4. NEW VALUE IN A CONTEXT

Applicative in JAVA

```
interface Applicative<AP, T>  extends Functor<T> {

  public <S, U> Applicative<AP, U>  apply(Applicative<AP, S> applicative);

  public <S> Applicative<AP, S>  unit(S value);

}
```

What is a Monad?

For a scientific definition please see this Wiki

Simply put: Monads apply a function that returns a wrapped value to a wrapped value.

A monad is a parameterized type M<T> with functions:

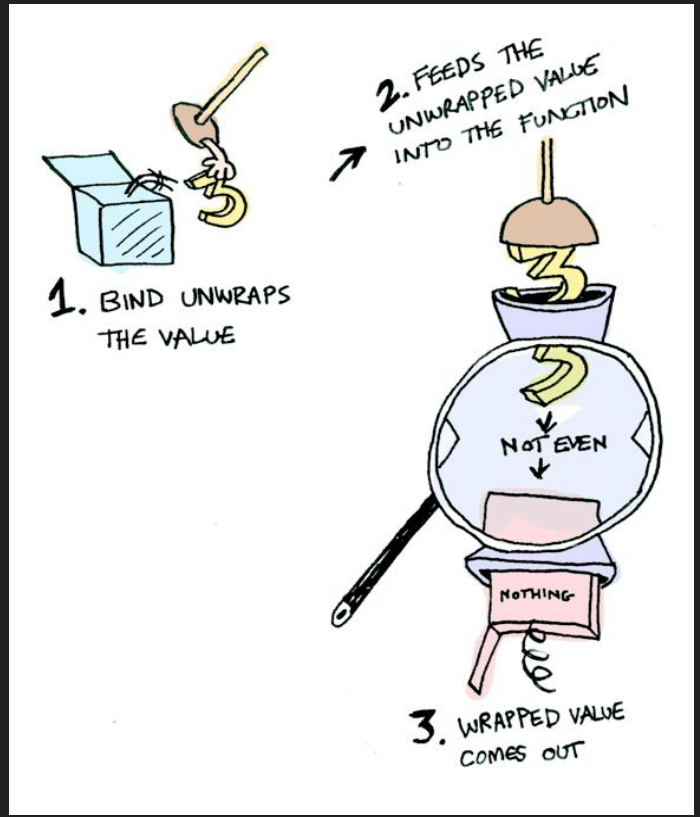
"unit": `T -> M<T>`

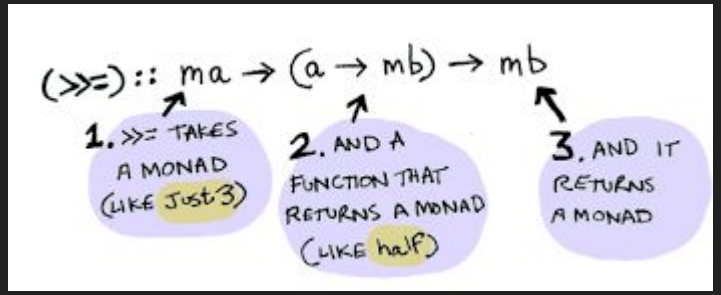
"bind": `M<T> bind T -> M<U> = M<U>`

Monad laws:

- Left identity
- Right identity
- Associativity

https://medium.com/coding-with-clarity/understanding-the-optional-monad-in-java-8-e3000d85ffd2#.4qbaa8r5t




(>>=) :: ma → (a → mb) → mb
1. >>= TAKES A MONAD (LIKE Just 3)
2. AND A FUNCTION THAT RETURNS A MONAD (LIKE half)
3. AND IT RETURNS A MONAD

Monad in JAVA

```
public interface Monad<T> extends Functor<T> {

  /**
   * in Java Monads use the naming flatMap instead of bind
   */
  public <S> Monad<S> flatMap(Function<T, Monad<S>> function);

  public <S> Monad<S> unit(S value);

}
```

Why do FP needs these concepts?

1. We want to program only using functions. ("functional programming (FP)" after all).
2. But how can we form an ordered sequence of functions (i.e. a program) using no more than functions?
   a. Solution: compose functions. `f(x), g(x) -> f(g(x,y))`
3. But some functions might fail, but we have no "exceptions"in FP
   a. Solution: Let's allow functions to return two kind of things: `g : Real,Real -> Real | Nothing`
4. But functions should (to be simpler) return only one thing.
   a. Solution: let's create a new type of data to be returned, a "boxing type" so we can have
      `g : Real,Real -> Maybe Real`
5. What happens now to `f(g(x,y))`? We don't want to change every function we could connect with g to consume a Maybe Real.
   a. Solution: let's have a special function to "connect"/"compose"/"link" functions. That way, we can, behind the scenes, adapt the output of one function to feed the following one.
6. In our case: `g >>= f` (connect/compose g to f).
   a. we want >>= to get g's output, inspect it and, in case it is Nothing just don't call f and return Nothing;
   b. or on the contrary, extract the boxed Real and feed f with it.
   c. also note that >>= must be written only once per "boxing type" (different box, different adapting algorithm).
7. Many other problems arise which can be solved using this same pattern:
   a. Use a "box" to codify/store different meanings/values, and have functions like g that return those "boxed values".
   b. Have a composer/linker g >>= f to help connecting g's output to f's input, so we don't have to change any f at all.

Interesting reads

http://nyelvek.inf.elte.hu/leirasok/Haskell/ (only in hungarian)

http://stackoverflow.com/questions/36636/what-is-a-closure

https://wiki.haskell.org/Functional_programming

https://dzone.com/articles/introduction-functional

http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html

http://stackoverflow.com/a/36878651

http://stackoverflow.com/a/8357604

http://stackoverflow.com/questions/3161112/functional-programming-and-type-systems

JAVA 8 SE

Functional Interfaces

```
@FunctionalInterface
public interface MyInterface {

  public boolean myMethod();

}
```

What makes a functional interface?

- **It must be an interface with one single public abstract method!**
  - **Important!** It can have other default methods, the limitation only applies to the number of public abstract methods
  - If an interface declares an abstract method overriding one of the public methods of java.lang.Object, that also does not count toward the interface's abstract method count
- This is not a new concept at all, these used to be called Single Abstract Method type (SAM)
- The use of the annotation gives compile time error if you violate the first rule
- There are tons of existing interfaces have already fulfill these requirements, luckily these became valid functional interfaces from Java 8 (e.g.: Runnable)

Built-in functional interfaces: `java.util.function`

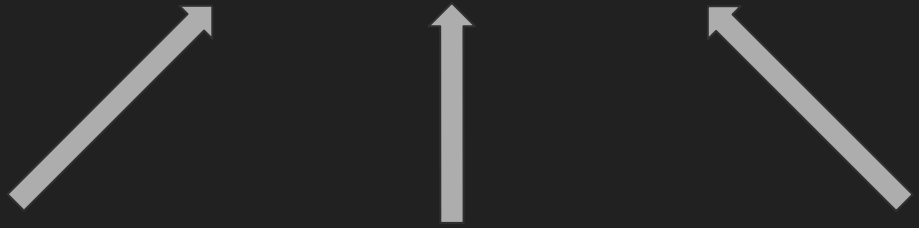
- The most important ones:
  - `Predicate<T>`: boolean test(T t);
  - `Consumer<T>`: void accept(T t);
  - `Function<T, R>`: R apply(T t);
  - `Supplier<T>`: T get();
  - `UnaryOperator<T>`: it is a Function<T, T>
  - `BinaryOperator<T>`: it is a BiFunction<T, T, T>

Lambdas - Closures*

*over final or effectively final variables

This is a so called lambda expression

```
(int x, int y) -> {return x == y;}
```



Argument list

Arrow Token

Block

In Java 7, you would write this...

```
void checkRows(Collection<Row> rows, RowChecker rowChecker) {
      for (Row row : rows) {
            if(!rowChecker.isValidRow(row)) {
                  throw new RuntimeException("Bad row");
            }
      }
}

void doSomething() {
      …
      checkRows(
            rows,
            new RowChecker() {

                  @Override
                  public boolean isValidRow(Row row) {
                        return row != null;
                  }

            });
}
```

But from Java SE 8, you can write this...

```
void doSomething() {
    …
    checkRows(rows, (Row row) -> {return row != null;});
    checkRows(rows, r -> r != null);
    …
}
```

Things to know about lambdas

Syntactic rules

1. A lambda expression can have zero, one or more parameters.
2. The type of the parameters can be explicitly declared or it can be inferred from the context.
3. Multiple parameters are enclosed in mandatory parentheses and separated by commas. Empty parentheses are used to represent an empty set of parameters.
4. When there is a single parameter, if its type is inferred, it is not mandatory to use parentheses. e.g. a -> return a*a.
5. The body of the lambda expressions can contain zero, one or more statements
6. If body of lambda expression has single statement curly brackets are not mandatory and the return type of the anonymous function is the same as that of the body expression. When there is more than one statement in body than these must be enclosed in curly brackets.

Only final or effectively final variables can be in their context. Which means Lambdas only close over values, but not variables! So lambdas are **Closures (with restrictions)!**

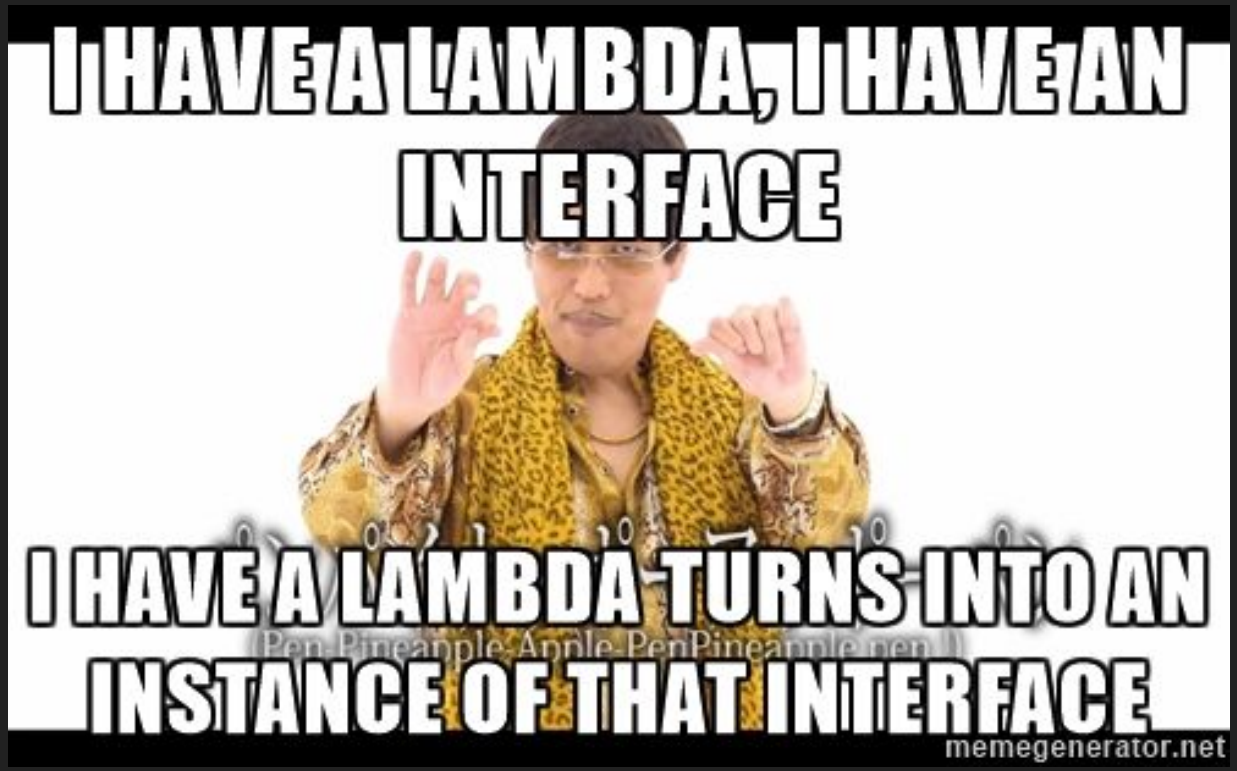
These are all valid lambdas

```
(int a) -> {return a + 1;}

(int a) -> a + 1

a -> a + 1

() -> {System.out.println("lambda");}

() -> System.out.println("lambda")

() -> true
```

```
(true) -> {}

(int a, int b) -> a + b

(a, b) -> {return a + b;}

(int y) -> {return;}
```

“Lambda expressions rely on the notion of ***deferred execution***. It means that code is specified now but runs later. Even though the execution is deferred, the compiler will still validate that the code syntax is properly formed.”

...but how lambdas
and functional
interfaces are related
to each other?


I HAVE A LAMBDA, I HAVE AN INTERFACE
I HAVE A LAMBDA TURNS INTO AN INSTANCE OF THAT INTERFACE
memegenerator.net

- Lambda expressions are anonymous functions with no name and they are passed (mostly) to other functions as parameters
- In Java method parameters always have a type and this type information is looked for to determine which method needs to be called in case of method overloading or method calling
- Every lambda expression also must be convertible to some type to be accepted as method parameters

That type in which lambda expressions are converted, are always of functional interface type.

```
Consumer<String> consumer =
   (String logMsg) ->
        System.out.println(logMsg);

consumer.accept(“Hello World!”);
```

“In simple words, a lambda expression is an instance of a functional interface. But a lambda expression itself does not contain the information about which functional interface it is implementing; that information is deduced from the context in which it is used.”

by http://www.lambdafaq.org/what-is-the-type-of-a-lambda-expression

Method references

- These are special method and constructor references converted to functional interfaces as well
- They are useful when your lambda does nothing but call another method or construct an object
- Method references acts like shortcuts to write really simple lambdas

There are 4 different type of method references

Reference to a static method	ContainingClass::staticMethodName
Reference to an instance method of a particular object	containingObject::instanceMethodName
Reference to an instance method of an arbitrary object of a particular type	ContainingType::methodName
Reference to a constructor	ClassName::new

Optional

What is wrong with Java null?

- In Java `null` is a special type, has no name and only literal `null` can be associated with it
- It is really hard to know if you need to be prepared for `null` return values when calling methods, and if you forget it, boom: `NullPointerException`
- There are best practices, but still, NPE happens really often and requires a decent amount of planning ahead (*and javadocs*) to avoid them

"*I call it my billion-dollar mistake.*" – Sir C. A. R. Hoare

What is `Optional`?

- A container object which may contain a non-null value or nothing
- **It is never said that optional “contain null”!**
- If a value is present, `isPresent()` will return true and `get()` will return the value
- It has additional useful methods , like `orElse()` or `orElseGet()`
- **It is not Serializable!**
- There is a special empty Optional: `Optional.empty()`
- If you call `get()` on an Optional return false when call `isPresent()` then `get()` throws a `NoSuchElementException`

What is `Optional` trying to solve?

- `Optional` is an attempt to reduce the number of null pointer exceptions in Java systems
- By using `Optional`, user is forced to think about the exceptional case
- The biggest advantage of `Optional` is its **idiot-proof-ness**

What is `Optional` not trying to solve?

`Optional` is not meant to be a mechanism to avoid all types of null pointers

The mandatory input parameters of methods and constructors will still have to be tested

Do not use `Optional`

- in the domain model layer (it’s not serializable)
- in DTOs (it’s not serializable)
- in input parameters of methods
- in constructor parameters

Do you remember Functors and Monads?

Java `Optional<T>` is a Monad (and also a Functor)!

Why?

- Parameterized type: `M<T> => Optional<T>`
- Unit: `T -> M<T> => Optional.of()`
- Bind: `M<T> bind T -> M<U> = M<U> => Optional.flatMap()`

```
public interface Optional<T> {

    // If a value is present, apply the provided mapping function to it, and if the result
    // is non-null, return an Optional describing the result
    public <U> Optional<U>  map(Function<T, U> mapper);

    // If a value is present, apply the provided Optional-bearing mapping function to it,
    // return that result, otherwise return an empty Optional.
    public <U> Optional<U>  flatMap(Function<T, Optional<U>> mapper);

    // many other methods...

}
```

So if it is a Functor, then use it like that!

Good

```
optional.map(s -> s.concat("other"));
```

Bad

```
if(optional.isPresent()) {
      String string = optional.get();
      return Optional.of(string.concat("other"));
}
else {
      return Optional.empty();
}
```

`Optional<T>` should be used as the return type of functions that might not return a value

Good

```
public Optional<String> getUserName() {
    return Optional.ofNullable(_name);
}
```

Bad

```
public String getUserName() {
    return _name;
}
```

You can easily find yourself in a situation having:

`Optional<Optional[...]<T>>`

```
...
public Optional<String> tryFindSimilar(String s) {...}

Optional<Optional<String>> bad = opt.map(this::tryFindSimilar);

Optional<String> similar = opt.flatMap(this::tryFindSimilar);
...
```

Summary

- F**unctional interfaces** are interfaces with only one public abstract method and an optional @FunctionalInterface annotation
- **Lambda expressions** are nameless functions converted to functional interfaces and with a special syntax: (T param) -> {commands;}
- Java 8 contains basic functional interfaces in the `java.util.function` package
- **Method References** are special method calls acts like lambdas, but converted to functional interfaces
- `Optional<T>` is a new wrapper class which is useful to prevent NPE to happen
- More details about its proper usage: https://medium.com/@afcastano/monads-for-java-developers-part-1-the-optional-monad-aa6e797b8a6e#.5eyxscu3r