



Meme Credit: reddit user u/HappyGoblin

Lab 5: Optionals and Lambdas

CS 2112 Fall 2021

September 27 / 29, 2021

Dynamic Typing

In dynamically typed languages, no type-checking is performed for the programmer.

```
1 function subtract(a, b) {  
2     return a - b;  
3 }  
4  
5 subtract("1", 1);
```

While this has its advantages, it can lead to difficult-to-debug errors, and thus has been losing popularity in recent years.

Type System

A statically typed language can enforce correct types. Thus, this entire class of error is impossible to reproduce in a typed language, as the code would never have compiled.

```
1 int subtract(int a, int b) {  
2     return a - b;  
3 }  
4 subtract("1", 1); // TYPE ERROR  
5  
6 String concat(String a, int b) {  
7     return a + b;  
8 }  
9 concat("1", 1); // ok
```

Null

Except...

```
1 String concat(String a, int b) {  
2     return a + b;  
3 }  
4  
5 concat(null, 1);
```

Null is a magic value that overrides all type-checking and can be passed in as any type, despite not being an object of that class. As such, `NullPointerException`s are frequent and difficult to debug.

Regret

“I call it my billion-dollar mistake.
It was the invention of the null reference in 1965.”

- Tony Hoare, inventor of null (and QuickSort)

Fundamental Theorem of Software Engineering

All problems in computer science can be solved by another level of indirection (abstraction)

Optional

- ▶ Contains a value, or is empty
- ▶ Makes you check when you access the value

Optional<T> API

```
1 public boolean isEmpty() { ... }  
2 public T get() { ... }
```

Unfortunately, `get()` can throw an unchecked exception, which kind of defeats the point.

Creating Our Own Abstraction

We'll build our own class, called `Maybe<T>`.

TODO list:

- ▶ Class Signature
- ▶ Instance Variables
- ▶ Class Invariant
- ▶ Constructors

Basic Setup

Implement the following three methods:

```
1  boolean isPresent()  
2  
3  T get() throws AbsentInformationException  
4  
5  T orElse(T other)
```

You may choose another checked exception for `get()` to throw if you wish.

Why “Lambda”?

Anonymous functions are also called lambdas.

Name originates from Alonzo Church's “Lambda Calculus” (1930s)

Wrote functions as $(\lambda x.M)$ with x as the parameter, M as the expression

Portions of this section's slides are adapted from CS 2110 by Prof. David Gries

Motivation

Imagine having two methods that share almost all their code, save for a single operation in the middle:

```
1 void method1() {  
2     // Lots of code  
3     int c = a + b;  
4     // More code  
5 }
```

```
1 void method2() {  
2     // The same code  
3     int c = a * b;  
4     // More code  
5 }
```

If the only thing that differed were the values, obviously we'd factor the same code into one method, and pass the different values in as a parameter. But here, the difference is in code. How could we pass in the operation we wish to perform as a parameter?

Assert Throws

Or consider the `AssertThrows` method from JUnit. How does the method call the code that's supposed to throw an exception?

```
1 try {  
2     // ??? What goes here?  
3     fail();  
4 } catch (Throwable t) {  
5     // Test passes if t has correct type  
6 }
```

We know how to pass primitives and objects to a method, but how could we go about passing an entire function as a value?

Example

The expression on the right below is equivalent to the method in the class on the left.

```
1 class Test {
2     int sum(int a, int b) {
3         return a + b;
4     }
5 }
```

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

More Examples

```
1 // without parameters
2 () -> System.out.println("Hello, world")
3
4 // with only 1 parameter
5 a -> a
6
7 // with explicit types
8 (int id, String name) -> name + id
9
10 // with a code block
11 (a, b) -> { if (a < b) return a; else return b; }
```


Implementation

Imagine trying to figure out how to pass a function as a value before lambdas existed in Java.

If we want to call a method, we have to call it on an object. If we accept an object as a parameter, we should declare an interface to ensure the object has the method we require.

```
1 interface F1 {  
2     Integer m(String s);  
3 }  
4 void doSomething(F1 v1) {  
5     System.out.println(v1.m("34"));  
6 }
```

Implementation

To use this method, we'd then need to make a new class that implements the interface, instantiate the class, and pass that object in as the parameter.

```
1 interface F1 { Integer m(String s); }
2 void doSomething(F1 v1) {
3     System.out.println(v1.m("34"));
4 }
5 class C implements F1 {
6     public Integer m(String s) {
7         return Integer.valueOf(s);
8     }
9 }
10 doSomething(new C());
```

Implementation

An anonymous function is actually equivalent to doing all of that. The anonymous function itself will create a new class that implements the interface and its method:

```
1 interface F1 { Integer m(String s); }
2 void doSomething(F1 v1) {
3     System.out.println(v1.m("34"));
4 }
5
6 // equivalent to last slide
7 doSomething(s -> Integer.valueOf(s));
```

Demo

You can actually see this by creating your own interface, and then calling `toString()` on some lambdas, as the default `toString()` is to print the class name and memory location:

```
1 interface F1 { Integer m(String s); }
2 F1 v1 = s -> Integer.valueOf(s);
3 v1.toString(); // $Lambda$13/0x000008@6cc4
4 F1 v2 = s -> Integer.valueOf(s);
5 v2.toString(); // $Lambda$14/0x000004@643b
```

Functional Interface

An interface that has exactly one abstract method in it can be annotated with `@FunctionalInterface`

```
1 @FunctionalInterface
2 interface F1 {
3     Integer m(String s);
4 }
```

Functional Interface Example

An example of a commonly used functional interface.

```
1  int[] arr = {1,2,3,4};  
2  arr = Arrays.stream(arr).filter(a -> a%2 == 0).toArray();  
3  // arr now contains {2,4}
```

Syntactic Sugar

An anonymous function that takes the same parameters as an existing method can be used directly with double colon notation

```
1 s -> Integer.valueOf(s)
2 // equivalent to
3 Integer::valueOf
```

Built-In Interfaces

Java's standard library provides a large number of built-in functional interfaces that you may find useful. They are under the `java.util.function` package.

Practice

Fill in the blank:

```
1  /** Remove empty strings from str.*/  
2  public void practice(List<String> str) {  
3      str.removeIf(      TODO      );  
4  }
```

The answer is `s -> s.isEmpty()`

ifPresent()

Write the method `ifPresent()` which calls a lambda on the value if it exists.

You may find the interface `Consumer<E>` useful. It represents a function from `E` to `void`.

```
1 void ifPresent(Consumer< ??? > action);
```

What should the action be parameterized on? It is a lambda that must be able to take something the value (of type `T`) can be cast to, meaning `T` or a supertype of `T`. Thus, the answer is `Consumer<? super T>`.

filter()

Write the method `filter()` which returns a `Maybe` with the value if the value exists and matches a given condition.

You may find the interface `Predicate <E>` useful. It represents a function from `E` to `boolean`.

```
1 Maybe<T> filter(Predicate< ??? > predicate);
```

What should the predicate be parameterized on? It is a lambda that must be able to take something the value (of type `T`) can be cast to, meaning `T` or a supertype of `T`. Thus, the answer is `Predicate<? super T>`.

map()

Write the method `map()` which calls a lambda on the value if it exists and returns a `Maybe` with the result.

You may find the interface `Function<E, F>` useful. It represents a function from `E` to `F`.

```
1 <U> Maybe<U> map(Function< ??? , ??? > mapper);
```

What should the mapper be parameterized on? It is a lambda that must be able to take something the value (of type `T`) can be cast to, meaning `T` or a supertype of `T`. It must return something that can be cast to `U`, either `U` or a subtype of `U`. Thus, the answer is `Function<? super T, ? extends U>`.

toString()

Now implement `toString()`, which returns the value of calling `toString()` on the value if it is present, or a special string (you get to pick) if it is not.

This can be done in **just one line** using `map()` and `orElse()`. Hopefully this shows how powerful lambdas can be, helping you keep your code succinct and legible.

Additional Exercise

Implement any or all of the following methods you may find useful.

- ▶ `int hashCode()`
- ▶ `Maybe<T> or(Supplier<? extends Maybe<? extends T>>)`
- ▶ `T orElseGet(Supplier<? extends T>)`
- ▶ `<U> Maybe<U> flatMap(
 Function<? super T, ? extends Maybe<? extends U>>)`

Congratulations! You've built your own abstraction that will hopefully make your future code cleaner and simpler.