Meme Credit: reddit user u/HappyGoblin

Lab 5: Optionals and Lambdas

CS 2112 Fall 2020

October 5 / 7, 2020

Dynamic Typing

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

```
function subtract(a, b) {
    return a - b;
}

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.

```
int subtract(int a, int b) {
    return a - b;
}
subtract("1", 1); // TYPE ERROR

String concat(String a, int b) {
    return a + b;
}
concat("1", 1); // ok
```

Null

Except...

```
String concat(String a, int b) {
        return a + b;
}

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, Null Pointer Exceptions 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

```java
public boolean isEmpty() { ... }
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:

```java
boolean isPresent()

T get() throws AbsentInformationException

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:

```
void method1() {
    // Lots of code
    int c = a + b;
    // More code
}
```

```
void method2() {
    // The same code
    int c = a * b;
    // More code
}
```

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 with 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?

```
try {
    // ??? What goes here?
    fail();
} catch (Throwable t) {
    // Test passes if t has correct type
}
```

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.

```java
class Test {
  int sum(int a, int b) {
    return a + b;
  }
}
```

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

More Examples

```java
// without parameters
() -> System.out.println("Hello, world")

// with only 1 parameter
a -> a

// with explicit types
(int id, String name) -> name + id

// with a code block
(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.

```
interface F1 {
    Integer m(String s);
}
void doSomething(F1 v1) {
    System.out.println(v1.m("34"));
}
```

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.

```java
interface F1 { Integer m(String s); }
void doSomething(F1 v1) {
    System.out.println(v1.m("34"));
}
class C implements F1 {
    public Integer m(String s) {
        return Integer.valueOf(s);
    }
}
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:

```java
interface F1 { Integer m(String s); }
void doSomething(F1 v1) {
    System.out.println(v1.m("34"));
}

// equivalent to last slide
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:

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

Functional Interface

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

```
@FunctionalInterface
interface F1 {
    Integer m(String s);
}
```

Syntactic Sugar

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

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

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

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`.

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

```
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`.

```
<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.