CSC 335 – assertThrows(), Interfaces, and Anonymous Functions

Dr. Jonathan Misurda
jmisurda@cs.arizona.edu

A Puzzle

- We wanted to test whether our code throws the proper exception using JUnit
- We have lots of assertions in JUnit 5, including assertThrows()
  - But it's a bit different than the usual (expected, actual) parameter list
- Why can't we just do:

```
assertThrows(MastermindIllegalColorException.class,
          controllerInstance.isCorrect("xxxx");
```

A Puzzle Wrapped in an Engima

- Also, what is that whole `.class` thing anyway?
- What does assertThrows want to do?
  1. Call some code that potentially throws an exception
  2. Catch that exception if it happens
  3. Determine if the caught exception is the one we expected (and not one of some different type)

Step 1: Run Some Code

- When we call a method, the order of operations dictates that the arguments are evaluated first, and then they are passed by value to the method:

```
System.out.println(2+3);
```

- You know this will print 5, because the value of 2+3 is determined and then 5 is passed to `System.out.println()` to display

Oops!

- So if we do:

```
assertThrows(MastermindIllegalColorException.class,
          controllerInstance.isCorrect("xxxx"));
```

- The code will first call:
  `controllerInstance.isCorrect("xxxx")`
- This will raise the exception in our test method, not in the assertThrows method!

Passing Code to a Method

- What we really want is for the assertThrows() method to take some arbitrary code as a parameter
- How can we do that in Java?
  - We put code in methods, and methods in classes

For Example

```
class WrapperForException {
	public void wrapperMethod(
		MastermindController controllerInstance,
		String guess)
		throws MastermindIllegalColorException {

		controllerInstance.isCorrect(guess);
	}
}
```

- Then we could pass an instance of this class and assertThrows could call the method and everything would work

But…

- How would assertThrows know what our method is called?
- This is one way we can use Java's notion of an **interface:**
  - An **interface** is a list of methods that a class promises to implement
- This is related to the IsA relationship, but it is more of a "CanDo" relationship

Implementing Interfaces

- While you `extend` a class, you `implement` an interface
- When you implement an interface, you are responsible for providing an implementation for **all** of the methods in the interface
  - If you don't, your class is now `abstract`, and we don't want to talk about that right now

Interface

```
interface Testable {
	//Not quite generic enough, but let's pretend
	public void methodThatGeneratesAnException() throws
		Exception;
}
```

- Methods in interfaces do not typically contain an implementation
- Java eventually changes this rule (`default` methods in Java 8+) to solve a particular problem with interfaces:
  - If you ever add to an interface a new method, you have to go and add that method to everyone that implements it

Implementing an Interface

```
class WrapperForException implements Testable {
	public void methodThatGeneratesAnException()
		throws Exception
	{
		MastermindController controllerInstance = …;
		controllerInstance.isCorrect("xxxx");
	}
}
```

Polymorphism via Interfaces

- Just as we can refer to a derived class via a base class reference (Liskov substitution), we can refer to an object that implements an interface via an interface reference:

```
Testable t = new WrapperForException();
t.methodThatGeneratesAnException();
```

- The compiler knows this will always work, because we were required to implement that method in our class, or the class wouldn't have compiled

assertThrows

- So now assertThrows could take a parameter of type Testable and call the method it knows must be there, and test our code
- But this is a lot of work:
  - Required JUnit to provide a particular interface
  - We had to write a class that implemented that interface
  - And we'd have to make a separate class for every single test we'd want to run

Another way?

- While we have been focusing on OOPLs, specifically Java, there are other categories of programming languages
  - Procedural languages like C, BASIC, FORTRAN
  - Declarative languages like SQL, Prolog
  - Functional languages like LISP, Scheme, F#, ML
- Functional languages have something useful we might borrow:
  - The "anonymous" function

Functional Languages

- Functional Languages are based on the mathematical idea of a function
  - Take inputs, produce outputs, have no **side effects**
- A **side effect** is when our code does anything other than produce a return value
  - Change a class field, write to a file, etc.
- Functional languages take their inspiration from the "lambda calculus"
  - A mathematical model of computation that uses function calls and composition

Anonymous Functions

- A function can sometimes be allowed to have no name
  - In the lambda calculus, all function applications are denoted with the Greek letter lambda: λ
- Thus they are also often called "lambdas"

Java Lambdas

- Java 8 added support for lambda functions as a means by which we can store a (simple) computation to a variable or pass it to a method as a parameter
  - Saved or passed to be performed later
- The syntax is:
  `(parameter list) -> expr`
- or
  `(parameter list) -> { block; }`
- You can omit the data type of the parameters in a lambda expression.
- You can omit the parentheses if there is only one parameter.

assertThrows() with Lambdas

```
assertThrows(MastermindIllegalColorException.class,
       () -> { return controllerInstance.isCorrect("xxxx"); }
);
```

- The () means we take no parameters (void)
- The -> means we're defining the body of the lambda
- The { } make the body of the lambda method
- And that will all be assigned to a parameter, that assertThrows can call inside of a try/catch block

Example

```
public class Calculator {
    interface IntegerMath {
        int operation(int a, int b);
    }
    public int operateBinary(int a, int b, IntegerMath op) {
        return op.operation(a, b);
    }
    public static void main(String... args) {
        Calculator myApp = new Calculator();
        IntegerMath addition = (a, b) -> a + b;
        IntegerMath subtraction = (a, b) -> a - b;
        System.out.println("40 + 2 = " +
        myApp.operateBinary(40, 2, addition));
        System.out.println("20 - 10 = " +
        myApp.operateBinary(20, 10, subtraction));
    }
}
```

Functional Interfaces

- Hey, I thought we didn't need interfaces anymore!
- An interface with only one method is called a **functional interface**
- Java implements lambdas in terms of these functional interfaces
- So really we have two identical ways to do the same thing, but the lambda syntax is more convenient

Step 3: Determine the Exception Type

- We can ask an object what type it is via two different mechanisms
- The keyword `instanceof` asks if an object is an "instance of" a class or interface
  - But this is true for superclasses, and thus is not strict equality
  - A FileNotFoundException is an instanceof an IOException

getClass()

- `java.lang.Object.getClass()` method returns the runtime class of an object.
- All objects inherit this method since they all extend Object
- So we can now ask what Class (type) of exception we got without using instanceof:

```
Integer i = new Integer(5);
System.out.println("" + i);
```

Prints: `java.lang.Integer`

`getClass()` on a class?

- `getClass()` is an instance method
- We don't have an instance of our exception to pass to `assertThrows()`
- Java provides a ClassName.class syntax to get the Class of a class without needing an instance first

All together now…

```
assertThrows(
      MastermindIllegalColorException.class,
      () -> {
            controllerInstance.isCorrect("xxxx");
      }
);
```