Section 2: Interfaces, Abstract Classes, Generics and Try-Catch-Finally

Raymond Guo

Disclaimer: I'm a little rusty in 61B and actually relearning with y'all :). These are just my personal notes, so there may be errors in this document; if so, let me know! Also, this isn't affilianted with CSM or any Berkeley organization :).

Interfaces (Q1)

An interface is a Java abstract type that denotes what behaviors that a particular group of classes should always have. What differentiates an interface from a regular superclass is that you never actually instantiate the interface itself; instead, you instantiate classes that implement that interface.

For an example of a Java interface that is used commonly, check out the List interface!
https://docs.oracle.com/javase/8/docs/api/java/util/List.html

Here are the key takeaways you should get from interfaces:

- We say that a class `implements` an interface.
- An interface is like a blueprint for a group of classes. It specifies what methods each class should have, but not how each alss should implement those methods.
- If you want to specify the default behavior of a method in an interface, use the `default` keyword (this shouldn't come up too often).
- Interfaces are an example of Java's advantage in abstraction, polymorphism, and inheritance.

Abstract Classes (Q2)

I like to think of abstract classes as just regular classes, but with the ability to define *abstract methods*, which they don't have to provide the implementation for (similar to an interface). To define an abstract method, you just use the `abstract` keyword. Other than that, abstract classes pretty behave the same way as any other class.

Abstract Classses vs. Interfaces

Here is a list of differences we went in section today:

- *Type of methods*: Interface can have only abstract methods. Abstract class can have abstract and non-abstract methods. From Java 8, it can have default and static methods also.
- *Final Variables*: Variables declared in a Java interface are by default final. An abstract class may contain non-final variables.
- *Type of variables*: Abstract class can have final, non-final, static and non-static variables. Interface has only static and final variables.
- *Implementation*: Abstract class can provide the implementation of interface. Interface can’t provide the implementation of abstract class.
- *Inheritance vs Abstraction*: A Java interface can be implemented using keyword “implements” and abstract class can be extended using keyword “extends”.
- *Multiple implementation*: An interface can extend another Java interface only, an abstract class can extend another Java class and implement multiple Java interfaces.
- *Accessibility of Data Members*: Members of a Java interface are public by default. A Java abstract class can have class members like private, protected, etc.

(Source: https://www.geeksforgeeks.org/difference-between-abstract-class-and-interface-in-java/)

Generics (Q3)

Generics aare essentially "types" that we specify for a generic Java data structure, such as a LinkedList, HashSet, or many others. We use the `<>` syntax to specify the Generic type of a data structure; for example, `LinkedList<String> obList = new LinkedList<String>();` would be a linked list of strings. Only strings would be able be in this linked list.

A few quick notes:

- The line `LinkedList<Object> obList = new LinkedList<String>();` leads to a compile time error! This is a deviation from the typical subtyping rules that we have with static and dynamic types. They don't apply to generics! In order for you to compile this, it would have to be `LinkedList<Object> obList = new LinkedList<Object>();`
- The following code compiles! This is because a `String` is an `Object`, so Java sees no issue with this. In fact, obList can contain a `String`, followed by a `Dog`, followed by a `Circle`, since they're all `Objects`!

```
LinkedList<Object> obList = new LinkedList<Object>();
obList.add("this is a string but it's also an object!");
```

- *(I didn't go over this in section)* If you want to have a linked list of primitives (for example, ints), then you might see the following code. What's happening here is that `Integer` is a *wrapper class* for `int`; it's basically just an Object representation of the primitive so that it can be used as a Generic. When you add an `int` into `intList`, Java automatically converts the `int` into an `Integer`. This is called *autoboxing*.

```
int a = 0;
LinkedList<Integer> intList = new LinkedList<Integer>();
intList.add(a);
```

Try-Catch-Finally (Q4)

In section today, I went over the general flow to tracing a try-catch-finally block. Remember, the general flow is: `try -> catch -> finally -> return/throw`. To illustrate this, let's say we have the following code:

```
public class Main {

    public static void main(String[] args) {
        System.out.println(stuff());
    }

    public static String stuff() {
        try {
            System.out.println("try");
            doSomething();
        } catch (Exception e) {
            System.out.println("caught");
            return "return";
        } finally {
            System.out.println("finally");
        }
        return "return";
    }

    public static void doSomething() {
        System.out.println("doSomething");
    }
}
```

This would result in the following output:

```
try
doSomething
finally
return
```

Now let's look at the code below:

```
public class Main {
```

```
    public static void main(String[] args) {
        System.out.println(stuff());
    }

    public static String stuff() {
        try {
            System.out.println("try");
            doSomething();
        } catch (Exception e) {
            System.out.println("caught");
            return "return";
        } finally {
            System.out.println("finally");
        }
        return "return";
    }

    public static void doSomething() {
        System.out.println("doSomething");
        throw new NullPointerException("demo");
    }
}
```

The only difference between this code block and the one before is that deSomething now throws a `NullPointerException`. If this were to run, we would get the following output:

```
try
doSomething
caught
finally
return
```

Another two quick things that I went over in discussion:

- If you have two contiguous `catch` blocks, and the first `catch` block catches a more generic exception than the second, then this would be a compile time error. This is because Java would never reach the second `catch` block, since everything would be caught by the first `catch` block, thus leading to unreachable code.
- If you throw an Exception in a catch block, it is *not* caught by the same `try-catch-finally` block. This is why you might see nested `try-catch-finally` block; the outer block is designed to catch the exceptions that the inner one throws (for an example of this, see below).

Think you've mastered this material? Here's a slideshow made by Mudit Gupta reviewing an extremely complicated try-catch problem: http://bit.ly/try-catch-hard