ECE 522 – Software Construction, Verification and Evolution

Objectives:

- Getting familiar with anonymous inner classes syntax and definition
- Getting familiar with the purpose of using such classes

1. Introduction

Welcome to the sixth ECE 522 lab. This lab contains 2 deliverables, which you should demo to the lab instructor before the end lab session.

1.1 Anonymous Inner Classes

Anonymous classes in Java are more accurately known as anonymous ***inner*** classes. **Inner** class means that a class is defined inside another class. An anonymous inner class is an inner class that is declared without using a class name at all. That is, of course is why it's called an ***anonymous*** class.
The ***anonymous inner classes*** are very useful in some situations. For example, consider a situation where you need to create the instance of an object without creating the subclass of a class and also performing additional tasks such as method overloading.
Normal inner class syntax in java:

```
public class OuterClass {
      // ...
      public class InnerClass {
            // ....
      }
      // ...
}
```

In this case, the inner class can be created as follow:

```
OuterClass outerObj = new OuterClass();
OuterClass.InnerClass innerObj = outerObj.new InnerClass();
```

Anonymous inner class syntax in Java:

```
new SuperType() {
    // inner class methods and data
};
```

Here, `SuperType` can be an interface; then, the inner class implements that interface. Or SuperType can be a class; then, the inner class extends that class.

1.2 Anonymous Inner Class Example

To understand how to define and use these types of classes, it would be better to walk through them by an example.

```
class HelloWorld  {
    public void print() {
        System.out.println("Hello World!");
    }
}

class PrintAll {
    /* This creates an anonymous inner class: */
    HelloWorld HW = new HelloWorld () {
        public void print() {
            System.out.println("anonymous HelloWorld ");
        }
    };
}
```

In the code above, you can see that we have two classes, one called `HelloWorld` and another called `PrintAll`. The `HelloWorld` class is pretty straightforward. There's just a simple method called `"print"` that prints the text "Hello World!" when is called.
But another class called by `PrintAll` shown with white shading is an anonymous inner class.
It might look like we are creating an instance of the `HelloWorld` class called `HW`, but what's actually happening is that an instance of an anonymous class is being created. By using anonymous inner class, we are actually providing a ***new*** method definition for a method named `"print"` with no need to create a separate class, having it extend the `HelloWorld`.

1.3 Purpose of Anonymous Inner Class

You have seen now that by creating an anonymous inner class, we can override one or more methods of a superclass. In above example, the superclass is the `HelloWorld` class, and the method being overridden is the `print`.
But, we could have easily done the same thing by just creating a separate class, having it extend `HelloWorld`, and then just override the `print` method. So, what is the need to create an anonymous inner class when we could have done the same thing using a normal, separate class? Well,

1. The main thing is that it is quicker to just create an anonymous inner class rather than create a new separate class.
2. Anonymous inner classes are especially useful when you only need to override a small amount of functionality (like just one method) in a superclass, and don't want to deal with the overhead of creating an entire class for something so simple.
3. We tend to want to avoid creating extra classes without good reasons. The more classes which exist, the more difficult it is to find the one you want!

1.4 Anonymous Inner Classes Implementing Interfaces

Following example shows an anonymous inner class, which implements an interface. In

this code, we have an interface called `HelloWorld` that has just one method declaration inside it. Moreover, we again have another class called `PrintAll`.
Inside the `PrintAll`, the code in red is actually **creating an instance of an anonymous inner class that implements the `HelloWorld` interface.** The class that is being instantiated (where `"HW2"` is the instance variable) is anonymous because it has no name, and the anonymous class is actually implementing the `HelloWorld` interface. This means that we are simultaneously creating an anonymous class that implements the `HelloWorld` interface and also creating an instance of that anonymous class.
NOTE: this is the only time in Java when you will see the syntax like what we showed above – where it looks like we are actually creating an instance of an interface class directly without even using the `"implements"` keyword.

```
interface HelloWorld {
    public void print();
}

class PrintAll {
    HelloWorld HW2 = new HelloWorld () {
        public void print() {
            System.out.println("interface anonymous class implementer");
        }
    };
}
```

2. Deliverable 1 – Cuboid

Suppose we are given an array named `Cuboids` of type `Cuboid[]` as follows:

```
Cuboid[] Cuboids = new Cuboid[5];
```

Assume the elements are initialized to proper object references.
We want to sort a sequence of cuboids (1) by length (2) by area (3) by volume and lastly (4) by length and area without changing the `Cuboid` class.
Here are the solutions:

1. Supply the comparator classes for the `Cuboid` type, each implementing `Compararable` interface but providing different comparison logic.
2. Use an overloaded function following version of the `sort` method of the `Arrays` class.
3. Use anonymous inner classes.

Compile and run your code.
DEMO this deliverable to the lab instructor.

3. Deliverable 2 – Last Call: Fix the Design Pattern

In lab 3, you were presented with a design pattern. While the pattern represented a great idea, our implementation of that idea stunk! Big time!!
Specifically if you look at class `AnimalType` … the code sucks!!!
`AnimalType` has two massive problems.

1. The conditional structure – this requires to be updated every time a new animal enters the system (it also needs updating if an animal leaves the code base – yes, those todo's causing problems again). That is, the conditional structure is inflexible and needs to be removed.

Rewrite class `AnimalType` to avoid using a conditional structure. it still needs to accommodate a variable number of different animals and this number will change over time, but the code must remain CONSTANT; that is, the code will not change.

2. `AnimalType` needs to know about the animals – crazy! It needs to know that mice are small – insane!! And is needs to know how to create each animal ("`new lion()`" for example) – madness!!!

Rewrite class `AnimalType` so it knows NOTHING about the animals it handles.
DEMO this deliverable to the lab instructor.

Once, you have fixed those two issues our design pattern is pretty good. You can consider this new solution as industrial strength code! Hence, if you managed this – you are good to go outside into the real world and code!!
But don't I need to know about lambda functions, streams, bloom filters, concurrent hash map, marshalling… :-)

Good Luck