Lab 6: Anonymous Inner Class and Reflection

Objectives

- Getting familiar with anonymous inner classes syntax and definition
- Getting familiar with the purpose of using such classes
- Get to know the Java reflection mechanism
- Full mark: 20 points

Source files

- Cuboid.java
- JavaDPExample.java

1 Introduction

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 only instance of a class, without creating subclasses or performing additional tasks such as method overloading.

Regular usage of inner class in java:

```
public class OuterClass {
    // ...
    public class InnerClass {
        // ...
    }
    public static void main(String[] args) {
        OuterClass outerObj = new OuterClass();
        OuterClass.InnerClass innerObj = outerObj.new InnerClass();
    }
}
```

And the usage of the anonymous inner class in Java:

```
new SomeSuperType().method();
```

Here, `SomeSuperType` can be an interface, and then this anonymous inner class implements that interface; or it can be a class, and then the anonymous inner class extends that class.

To understand how to define and use such kind of classes, it would be better to walk through an example.

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

```
public class OuterClass {
    public static void main(String[] args) {
        new SomeClass() {                       // This is not an instance of SomeClass,
            @Override public void method() {    // but an instance of the sub class of SomeClass
                System.out.println("Anonymous Hello World!");
            }
        }.method();
    }
}
```

In the code above, you can see that we have two classes, `SomeClass` and `OuterClass`. The `SomeClass` is pretty straightforward -- just has a simple method that prints a piece of text. In the `main` method an anonymous inner class is defined (`new SomeClass() { ... }`) and invoked (`new SomeClass() { ... }.method();`).

1.2 Purpose of Anonymous Inner Class

You have seen now that by creating an anonymous inner class, we create a sub class that extends some super class. But as we can do such thing in the regular style, what is the advantage of using anonymous inner class? Well:

1. First of all, it is faster to just write an anonymous inner class rather than create a new separate class.
2. Anonymous inner classes are especially useful when you need and only need one instance of a class, especially such instance is immutable (`final`).
3. We 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!

1.3 Anonymous Inner Classes Implementing Interfaces

The following example shows an anonymous inner class that implements an interface. It is similar with the previous example. Remember this is the typical usage of anonymous inner classes.

```
interface SomeInterface {
    public void method();
}

public class OuterClass {
    public static void main(String[] args) {
        new SomeInterface() {
            @Override public void method() {
                System.out.println("Hello World!");
            }
        }.method();
    }
}
```

In the example, we need a class that implements the `SomeInterface` and an instance of such class. What's more, we only need such instance only once. So we **created an instance of an anonymous inner class that implements** `SomeInterface`.

Note this is the only time in Java you can see such syntax: a class implementing an interface without using the keyword `implements`.

2 Java Reflection

Unlike Python, JavaScript, etc., Java is not a *dynamic programming language*. It relies heavily on the type of each class/variable. However, dynamically loading classes at runtime is important for object oriented programming. So, is it possible for Java to figure out attributes and methods of some random class? And is it possible to access these attributes, or invoke these methods? The answer is yes -- the reflection guarantees such requests. It can:

- recognize any class **at runtime**;
- construct an instance of any class **at runtime**;
- recognize the attributes and method **at runtime**;
- invoke methods of an instance **at runtime**.

Here is a quick example -- you have used the `SimpleWriter1L` in Lab 1 like this:

```
import components.simplewriter.SimpleWriter1L;
SimpleWriter1L out = new SimpleWriter1L();      // Initialize
out.println("Hello World!");                    // Invoke
out.close();
```

But to use reflection, you do it like this:

```
Class<?> simpleWriter1L = Class.forName("components.simplewriter.SimpleWriter1L");  // Load
Object out = simpleWriter1L.newInstance();                                          // Initialize
Method println = simpleWriter1L.getDeclaredMethod("println", String.class);         // Invoke
println.invoke(out, "Hello World!");
Method close = simpleWriter1L.getDeclaredMethod("close");
close.invoke(out);
```

It seems reflection is more complicated, but why do we need it? One of the answers can be found in Deliverable 2. So let's start do it.

3 Deliverable 1 -- Cuboid

Suppose we have an array of cuboids:

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

We want to sort a them: (1) by length; (2) by area; (3) by volume; and lastly (4) by length first and then area. We don't want to change the `Cuboid` class.

Here is the solution:

1. For each sorting procedure, sse the `java.util.Arrays.sort` method;
2. Supply each `java.util.Arrays.sort` method a class that implements `java.util.Comparator`;
3. Make each class anonymous.

Write and run your code.

DEMO this deliverable to the lab instructor (10 points).

4 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 the class `AnimalType` ... the code sucks!!! It has two massive problems:

1. The conditional structure (Line 29 to Line 34) -- this requires to be updated every time a new animal type enters the system (it also needs to be updated if an animal type leaves the code base -- yes, those todo's causing problems again). That is, the conditional structure is inflexible and needs to be removed.
2. `AnimalType` needs to know about the animal types -- crazy! It needs to know that mice are small -- insane!! And it needs to know how to create each animal (`new Lion()` for example) -- madness!!!

Rewrite `AnimalType`:

1. to avoid using a conditional structure;
2. so it knows NOTHING about the animals it handles. It may still need to accommodate a variable that stores different animal types, but the code in `AnimalType` must remain CONSTANT whenever animal types are changed. **HINT**: think *reflection*.

DEMO this deliverable to the lab instructor (10 points).

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*, ... ?