Prepared by Linan Qiu <lq2137@columbia.edu> adapted from Java Tutorials on Nested Classes

Nested Classes

This is an elaboratio of the section that I glossed over in the previous chapter (I guess I found more time this week!). Let's go through nested classes.

What are Nested Classes

The following section is practically lifted from https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html because I find it hard to write a better one.

The Java programming language allows you to define a class within another class. Such a class is called a nested class and is illustrated here:

```
class OuterClass {
  // ...

  class NestedClass {
    // ...
  }
}
```

Nested classes are divided into two categories: **static and non-static**.

```
class OuterClass {
  private int someVariable;

  class NonStaticNestedClass {
    // ...
  }

  static class StaticNestedClass {
    // ...
  }
}
```

Non-static nested classes have access to other members of the enclosing class, even if they are declared private. In this example, it will have access to `someVariable`. Static nested classes do not have access to other members of the enclosing class. Hence, `StaticNestedClass` will not have access to `someVariable`. A nested class can also be declared `private public protected`.

Why do we use nested classes?

- It is a way of logically grouping classes that are only used in one place: If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together. Nesting such "helper classes" makes their package more streamlined.
- It increases encapsulation: Consider two top-level classes, `A` and `B`, where `B` needs access to members of `A` that would otherwise be declared private. By hiding class `B` within class `A`, `A`'s members can be declared private and `B` can access them. In addition, `B` itself can be hidden from the outside world. You will find this useful when we talk about `AwsmLinkedList` in the next chapter.
- It can lead to more readable and maintainable code: Nesting small classes within top-level classes places the code closer to where it is used.

Static Nested Classes

As with class methods and variables, a static nested class is associated with its outer class. And like static class methods, a static nested class cannot refer directly to instance variables or methods defined in its enclosing class: it can use them only through an object reference.

A static nested class interacts with the instance members of its outer class (and other classes) just like any other top-level class. In effect, a static nested class is behaviorally a top-level class that has been nested in another top-level class for packaging convenience.

Static nested classes are accessed using the enclosing class name:

```
OuterClass.StaticNestedClass
```

For example, to create an object for the static nested class, use this syntax:

```
OuterClass.StaticNestedClass nestedObject = new OuterClass.StaticNestedClass();
```

Inner Classes

As with instance methods and variables, an inner class is associated with an instance of its enclosing class and has direct access to that object's methods and fields. Also, because an inner class is associated with an instance, it cannot define any static members itself.

Objects that are instances of an inner class exist within an instance of the outer class. Consider the following classes:

```
class OuterClass {
  // ...
```

```
  class NestedClass {
    // ...
  }
}
```

An instance of `InnerClass` can exist only within an instance of `OuterClass` and has direct access to the methods and fields of its enclosing instance.

To instantiate an inner class, you must first instantiate the outer class. Then, create the inner object within the outer object with this syntax:

```
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

Refactoring `AwsmArrayList` and `AwsmArrayListIterator`

When I copied `AwsmArrayListIterator` into `AwsmArrayList`, I was faced with a design choice: do I make the `AwsmArrayListIterator` a **static** or **non-static** class? Well, remember the train jumping analogy? `AwsmArrayListIterator` is specific to an instance of a `AwsmArrayList` and would need to access its data. Hence, we should make the iterator **non-static**.

And that's exactly what we did in the previous chapter!