Nested Classes

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. Nested classes that are declared static are called *static nested classes*. Non-static nested classes are called *inner classes*

```
class OuterClass {
    ...
    static class StaticNestedClass {
        ...
    }
    class InnerClass {
        ...
    }
}
```

- A nested class is a member of its enclosing class.
- Non-static nested classes (inner classes) have access to other members of the enclosing class, even if they are declared private.
- Static nested classes do not have access to other members of the enclosing class.

Why Use Nested Classes?

- It is a way of logically grouping classes that are only used in one place
- It increases encapsulation
- It can lead to more readable and maintainable code

Static Nested Classes

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 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.

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();
```

There are two special kinds of inner classes: local classes and anonymous classes.

Local Classes

Local classes are classes that are defined in a *block*, which is a group of zero or more statements between balanced braces. You typically find local classes defined in the body of a method.

You can define a local class in a method body, a for loop, or an if clause etc.

A local class has access to the members of its enclosing class.

```
public class LocalClassTest {

private String outerClassVar = "I am from outer Class.";
private final String outerClassfinalVar = "I am Final Variable from outer Class.";

public static void main(String[] args) {
        LocalClassTest test = new LocalClassTest();
        test.tryLocalClassHere();
}

public void tryLocalClassHere(){
        String testVariable = "Simple Local Variable";
        final String finalLocalVar = "Final Local Variable ";
        /**
         * Local classes are classes that are defined in a block
         */
class LocalClass{
        public String localClsInstanceVariable = "local Class Instance Variable ;
```

```
        void localMethod(){

        System.out.println("I can access my own variables :: " + localClsInstanceVariable);
                /** Cannot touch non-final variables */
                //          testVariable = "Setting value inside local class";
        System.out.println("I can access final local variables Only " + finalLocalVar);
        System.out.println("I can access non-final instance variable of Outer Class " + outerClassVar);
        System.out.println("I can access final instance variable " + outerClassfinalVar);
        }
        /** Cannot declare main method in Local Class - as its static **/
        // public static void main(String[] args) {}
        }
        LocalClass local = new LocalClass();
        local.localMethod();
}
}
```

Anonymous Classes

Anonymous classes enable you to make your code more concise. They enable you to declare and instantiate a class at the same time. They are like local classes except that they do not have a name. Use them if you need to use a local class only once.

```
public class HelloWorldAnonymousClasses {

   interface HelloWorld {
      public void greet();
      public void greetSomeone(String someone);
   }

   public void sayHello() {

      class EnglishGreeting implements HelloWorld {
         String name = "world";
         public void greet() {
            greetSomeone("world");
         }
         public void greetSomeone(String someone) {
            name = someone;
            System.out.println("Hello " + name);
         }
      }
          HelloWorld englishGreeting = new EnglishGreeting();
          englishGreeting.greet();
          }

           public static void main(String... args) {
      HelloWorldAnonymousClasses myApp = new HelloWorldAnonymousClasses();
      myApp.sayHello();
   }
}
```

The anonymous class expression consists of the following:

- The new operator
- The name of an interface to implement or a class to extend. In this example, the anonymous class is implementing the interface HelloWorld.
- Parentheses that contain the arguments to a constructor, just like a normal class instance creation expression. **Note**: When you implement an interface, there is no constructor, so you use an empty pair of parentheses, as in this example.
- A body, which is a class declaration body. More specifically, in the body, method declarations are allowed but statements are not.