

# CST8221 – Java Application Programming

## Hybrid Activity #1

### Nested and Inner Classes

#### **Terminology**

The Java programming language allows you to define a class within another class. A **nested** class is a class defined inside the definition of another class (similar to class methods, which are always defined inside a class). The class which contains a declaration of another class or classes is sometimes called “outer” or “enclosing” class. The term **top-level** class applies to a class which declaration is not part of another class (that is, the class is the top level of containment). Nested classes are divided into two categories: static and non-static. Nested classes that are declared static are simply called static nested classes. Non-static nested classes are called **inner** classes.

There are three types of inner classes:

- **Member inner classes.** They are defined inside another class, at the same level as fields and methods. They are members of the enclosing class and they have full access to the fields and the methods of the enclosing class regardless of their access specifiers.
- **Local inner classes.** They are defined inside a block of code. They are local to the enclosing method or block similar to the local variables.
- **Anonymous inner classes.** They are inner classes that do not have names. They are used in a manner similar to anonymous array – to be passed as arguments to methods.

An inner class can have a **private** or **protected** access specifier in addition to public and package specifiers (top-level classes can be designated as **public** or **package** only). Inner classes may not contain class methods or class fields (static fields or methods). Local inner classes cannot be declared with access specifiers. They can be used only within the method as any other local variable. Local inner classes have access to the enclosing class fields. They also have access to any local variable (including the arguments) which is marked as **final**, but not to the other local variables. Anonymous inner classes cannot have constructors because they do not have names. Initializer blocks can be used to initialize their field to values different from the default initializers. The syntax of the initializer block is the same as the one used to initialize blank final fields: *{initialization statements}*. The instantiation of an anonymous inner class has the following syntax.

**new optional\_class\_ or\_ interface\_name() {class body}**

where the *optional\_class\_ or\_ interface\_name* is the name of either a class that is extended or an interface that is implemented by the anonymous class being defined and instantiated.

Inner classes as any class in Java are compiled into separate class files. The name of an inner class file always has the name of the top-level class containing the inner class definition followed by \$ followed by the name of the inner class. Example: **A\$B.class**.

#### **The Nature of Things**

Inner classes were introduced later in Java (since Java 1.1) when the event model of Java had been changed. You can write a sound Java applications without using inner classes, but using them can lead to more elegant and easier to maintain solutions. They are now integral part of many classes in the Java library. Also, they are extensively used in event handling code of the Java GUI application. The impact of the inner classes on your code depends very much on how you use

them. They are not as critical as inheritance, without which Object-Oriented programming is severely limited.

An anonymous inner class is a very convenient way to implement an interface when the interface defines only one method. This type of interfaces are called Single Abstract Method (SAM) interfaces. Since Java 8 there is a better way to implement SAM interfaces – using **lambda expressions**. This situation is very common when you program graphical user interface (GUI). If the interface define more than one abstract method inner classes are very useful for creating so called adapter classes. Adapter Classes implement a programming design pattern intended to convert the method(s) of a class or interface into a different method(s) that the class or the interface does not implement. You can define adapter classes that are implemented as inner classes, in the places where they are used. The important thing to remember is that the resulting adapter classes also have access to the fields and methods of the enclosing class. Another useful feature is that inner classes that implement one-method interfaces can be passed as arguments. You may wonder whether inner classes are worth the complications they bring to the code. The answer is “Yes.” If you use inner classes in a straightforward manner and for the purposes for which they were intended and designed, you can reap significant benefits from minimal efforts. Something more, as I have already pointed out inner classes are used extensively in the Java libraries (remember the Data Structures course) and professional applications.

## References

Textbook 1 – Chapter 14, Textbook 2 – Chapter 15.

Links:

<http://docs.oracle.com/javase/tutorial/java/javaOO/nested.html>


<http://docs.oracle.com/javase/tutorial/uiswing/events/generalrules.html>

Google search: Java inner class tutorial

## Syntax, Semantic Dissection, Code Example, and Program Code Dissection

Let us take a look at the following fragment of code which is used in the implementation of SimpleSwingGUle2.java in Lab1 code examples.

```
WindowListener wl =
new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
};
addWindowListener(wl);
```

The purpose of this code is to create an instance of type WindowListener needed to handle the GUI application window's close button .

If you look in the Java API you will find out that *java.awt.event.WindowAdapter* class is an abstract class that implements with no operations all the methods of the *java.awt.event.WindowListener* interface (and some other interfaces). Since *WindowAdapter* is an abstract class it cannot be directly instantiated, that is, you cannot create an object with writing a statement `new WindowAdapter();`. You must write a class which extends *WindowAdapter* and then override the appropriate method. This class can be a separate top-level class (see SimpleSwingGUle1.java in Lab 1) or an inner class. In the code fragment under discussion an anonymous inner class is used to create an object of type *WindowListener*. The anonymous class

(no name) extends *WindowAdapter* and overrides its *windowClosing()* method. If we do not need to reuse the object, the code could be rewritten as follows:

```
addWindowListener(new WindowAdapter() {  
    public void windowClosing(WindowEvent e) {  
        System.exit(0);  
    }  
});
```

In this case the anonymous inner class definition and instantiation are used as a parameter in the call of the *addWindowListener()* method.

In Lab 1 Example 3 code *SimpleSwingGUIe3.java* similar approach is used to create an object of type *ActionListener* which handles the action events for the button. In Lab 1 Example 3L the anonymous inner class is replaced with lambda expression since *ActionListener* is a SAM interface.

## **Exercise**

Modify the code of *SimpleSwingGUIe2.java* so that uses a member inner class instead of anonymous inner class. Call the inner class *ClosingWindowAdapter*.

## **Questions**

- Q1. Can an inner class method have access to the fields of the enclosing class?
- Q2. Can an anonymous inner class have a constructor?
- Q3. Can a local class have access to non-final local variables?

## **Submission**

No submission is required for this activity.

## **Marks**

No marks are allocated for this activity, but do not forget that understanding how to use inner classes is essential for building effective GUI applications.

And do not forget that:

*“The most potent muse of all is our own inner child.” Stephen Nachmanovitch*