Inner Classes – Quick Overview

Sometimes we define a class whose purpose is only to “help out” another class. For example, the ListItem class for a LinkedList class. In that kind of situation, it would be nice if the helper class didn’t stand on its own, but instead was more closely associated with the class itself.

So far, all classes we have defined have been “top level classes”. In other words, as far as the Java compiler is concerned, the classes are independent and exist on their own. We have put the classes in the same file (only one is public) or put the class in the same package.

These are helpful, but neither really says that one class is somehow “part of” another class. To express this kind of relationship between classes, Java programmers can use something called “inner classes”. There are four varieties of inner classes:
Nested Classes

This is a class defined as a static class member. From outside its name is prefixed with the name of the containing class separated by a dot. Access to the class can be controlled through protection modifiers.

For example:

```java
class A {
    A() {
        ...
        B b = new B();
        b.click(this);
        ...
    }
    void beginA() {
        ...
    }
    private static class B {
        public void click(A a) {
            ...
            a.beginA();     // Note
            ...
        }
    }
}
```

Note: Just like with any other method, you must use `objectName.methodName()` notation for `beginA()`.

Technical Note:
- If the nested class B were not private, then can be accessed from outside as `A.B b = new A.B();`
Member Classes

They are declared like instance variables and methods (not static) and each instance of a member class has a corresponding instance of the containing class (usually the one that created it).

Instances of member classes have access to the instance variables and methods of the containing class (and vice versa), as well as other member class instance variables and methods created by the original class.

For example:

```java
public class A {
    public A() {
        ...
        B b=new B();
        b.click();
        ...
    }
    void beginA() {
        ...
    }
    private class B {
        public void click() {
            ...
            beginA();     // Note
            ...
        }
    }
}
```

Note: This time `beginA()` can be called without a source because there is only one object it could belong to.

Technical Notes:
- Member classes can't contain any static members, and they can't have the same name as any containing class or package.
- If the member class `B` were not private, then can be accessed from outside as
  ```java
  A a = new A();
  A.B b = a.new B();
  ```
Local Classes

These are similar to member classes, except that they are declared within a method, and have access to final local variables within that scope and they are only visible and usable within that scope.

For Example:

```java
public class A {
    public A() {
    ...
    class B {
        public void click() {
            ...
            beginA();
            ...
        }
    }
    ...
    B b=new B();
    b.click();
    ...
}  
void beginA() {
    ...
}
}
```

Local classes have the same restrictions as member classes: local classes can't contain any static members and they can't have the same name as any containing class or package.
Anonymous Classes

These are similar to local classes, except they have no names. Instead, they are defined within the expression that instantiates them.

Anonymous classes either
- extend a class, or
- implement an interface (and extend Object)

Consider the following example:

```java
public class A {
    public A() {
        ...
        B b = new B(???) { // Note
            public void click() {
                beginA();
            }
        };
        b.click();
        ...
    }
    void beginA() {
        ...
    }
}
```

Note: $B$ is the name of the class being extended or interface being implemented. For the first case, any parameters to the constructor for $B$ need to be included.

Technical Notes:
- An anonymous class inherits its constructor because it can't define its own.
- An anonymous class has no name.
- An anonymous class has no protection modifiers --- it has no name, so you can't access it from outside anyway.