

# Inner Classes

- What are Inner Classes?
- Categories of Inner Classes
- Using Inner Classes

2 - 1

---



## Welcome

This tutorial is taken from a training class produced by Joel Barnum of Descriptor Systems.

You can use this tutorial for your own private use, but any reproduction or copying for any other purpose is prohibited.

For information on Descriptor Systems, go to [www.descriptor.com](http://www.descriptor.com)



0 - 0

---

## What Are Inner Classes?

- Inner classes are classes that are defined within the block of another class

```
1   public class EnclosingClass
2   {
3       public class MyInnerClass
4       {
5           . . .
6       }
7       . . .
8   }
```

2 - 2

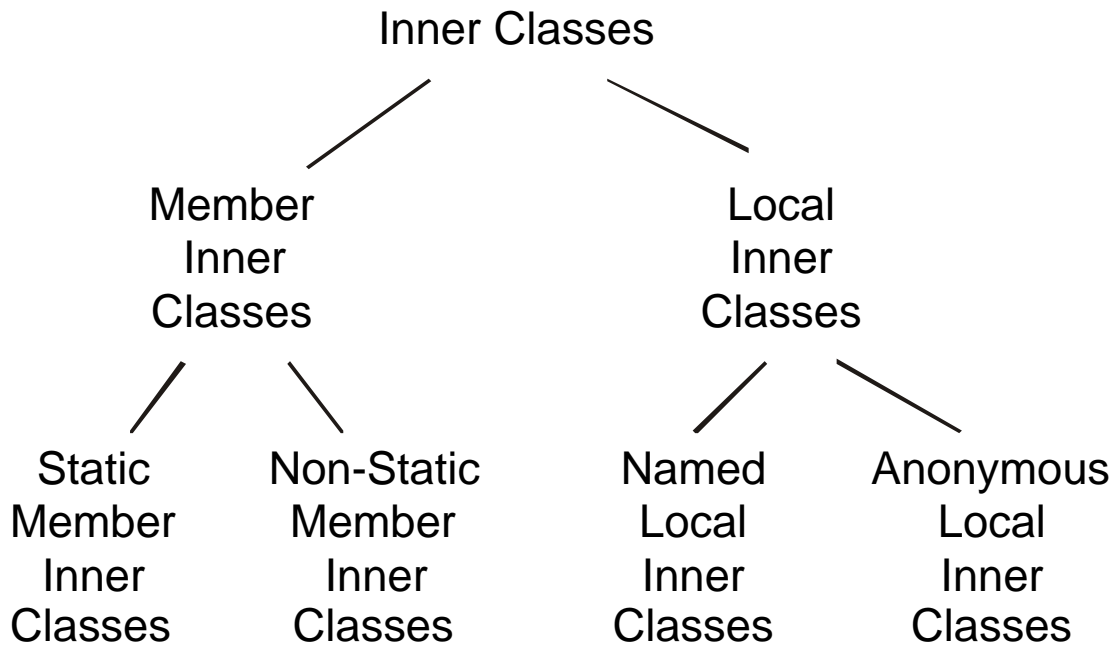
---

Inner classes let you scope a class definition to within another class.

Inner classes were introduced to Java in Version 1.1.

When you compile a class with inner classes, the compiler generates separate class files for the enclosing and inner classes. The naming convention for the inner-class file names is `EnclosingClass$MyInnerClass.class` (there is a special case for file names of anonymous inner classes).

# Inner Class Categories



2 - 3

---

These category names are widely used and descriptive, but are not actually defined in the Java documentation.

## Why Use Inner Classes?

- If a given class is used only by a single class, you can simplify the design by using an inner class
- Inner classes let you explicitly show that a class depends on another
- Anonymous local inner classes are very handy to hook up listeners to JavaBean events
- Downside: Inner class syntax is a bit obscure

2 - 4

---

Used properly, inner classes can simplify designs and make code more readable and maintainable. The syntax can be a bit intimidating until you get used to it, however.

## Static Member Inner Classes

- Static member inner classes are defined as static within the enclosing class

```
1   public class MyEnclosingClass
2   {
3   . . .
4       static public class InnerClass3
5       {
6           private int y = 14;
7
8           public void myMethod()
9           {
10              System.out.println(y);
11          }
12      }
13  . . .
14  }
```

2 - 5

---

Here we show defining a static member inner class that itself defines a private field and a public method. Within the inner class, you are free to use any of the class definition techniques available to a non-inner class.

Since the inner class is defined as public, it is accessible from outside of the enclosing class. If we had defined it as private, then it would only be visible within the enclosing class.

## Accessing Enclosing Class Members

- Static member inner classes can access any non-static field or method defined in the enclosing class

```
1  public class MyEnclosingClass
2  {
3      private int x = 12;
4      static private int y = 20;
5      static private int z = 21;
6
7      static public class InnerClass3
8      {
9          private int y = 14;
10
11         public void myMethod()
12         {
13             System.out.println(y);
14             System.out.println(z);
15             System.out.println(MyEnclosingClass.y);
16         }
17     }
18     . . .
19 }
```

2 - 6

---

The inner class can access private members of the enclosing class. That makes sense if you consider that the inner class is itself considered a member of the enclosing class.

Note the odd syntax that's required if the inner class defines a field with the same name as a field in the enclosing class.

## Instantiating Static Member Inner Class Objects

- From within a static method in the enclosing class, you can use normal **new** operator syntax
- From outside the enclosing class, you use **new** and specify both the enclosing and static member inner class name

### Inside enclosing class

```
InnerClass3 m3 = new InnerClass3();
```

### Outside of the enclosing class

```
MyEnclosingClass.InnerClass3 m3 =  
    new MyEnclosingClass.InnerClass3();
```

2 - 7

---

The syntax to instantiate a static member inner class object from outside of the class is a bit weird, but is consistent with the syntax for accessing any kind of static member.

Note that in either case, since the inner class is static, you don't have to first create an instance of the enclosing class. We will see that with non-static member classes, you will need to first create an enclosing class object.

## Non-Static Member Inner Classes

- Non-static member inner classes are defined within the boundary of the enclosing class

```
1   public class MyEnclosingClass
2   {
3   . . .
4       public class InnerClass2
5       {
6           private int x = 14;
7
8           public void myMethod()
9           {
10              System.out.println(x);
11          }
12      }
13  . . .
14  }
```

2 - 8

---

Non-static member inner classes are syntactically similar to static -- just omit the static keyword. You instantiate objects and access them a bit differently, however.

## Accessing Enclosing Class Members

- Non-static member inner classes can access any of the enclosing class fields or methods

```
1  public class MyEnclosingClass
2  {
3      private int x = 12;
4
5      static public class InnerClass3
6      {
7          private int x = 14;
8          private static int y = 17;
9
10         public void myMethod()
11         {
12             System.out.println(x);
13             System.out.println(y);
14             System.out.println(MyEnclosingClass.this.x);
15         }
16     }
17     . . .
18 }
```

2 - 9

---

Non-static member inner classes can access any of the enclosing class's members, be they private or not, static or not.

Note the odd syntax required if an inner class defines a field with the same name as field from the enclosing class. The implication is that within an inner class, there are actually two "this" references -- the "this" for the inner class itself and the "this" for the enclosing class.

## Instantiating Non-Static Member Inner Class Objects

- Before you can instantiate a non-static member inner class object, you must first instantiate an enclosing class object
- From within a non-static method in the enclosing class, you can use normal **new** operator syntax
- From outside, you use **new** along with the reference to the enclosing class object

Inside enclosing class non-static method

```
InnerClass2 m2 = new InnerClass2();
```

Outside of an enclosing class non-static method

```
MyEnclosingClass m = new MyEnclosingClass();  
MyEnclosingClass.InnerClass2 m2 =  
m.new InnerClass2();
```

2 - 10

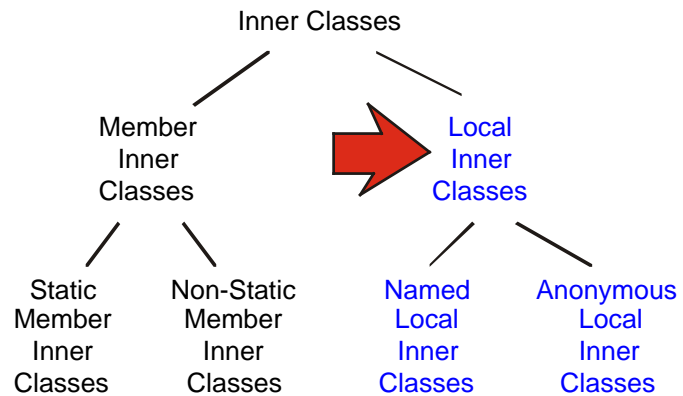
---

Non-static member inner classes are considered to part of an enclosing class INSTANCE, not of the enclosing class itself (static inner classes ARE considered part of the enclosing class). Therefore, the enclosing class instance must first exist before you can create an instance of the inner class.

Again, the syntax is different depending on where you are trying to instantiate the inner class object. From within an enclosing-class, non-static method, there's an implicit "this" for the enclosing class, so you can use the normal "new" syntax. From anywhere else, you need to use the rather odd "new" syntax that uses the dot operator following the reference to the enclosing class instance.

## Introduction to Local Inner Classes

- Local inner classes are defined within the boundary of a method
- The local class is only visible within the method that defines it
- Local class methods can access enclosing class fields *and* method local variables defined as **final**
- There are two categories of local inner classes: **named** and **anonymous**



2 - 11

---

The key difference between local inner classes and the two categories we've already covered is that local inner classes reside within a method, rather than being a member of the enclosing class itself.

## Named Local Inner Classes

- Named local inner classes are defined within the boundary of a method and are only visible within that method

```
1   public class MyEnclosingClass
2   {
3       public void anotherMethod()
4       {
5           class Inner4
6           {
7               private int x = 17;
8
9               public void myMethod()
10              {
11                  System.out.println(x);
12              }
13          }
14      }
15      . . .
16  }
```

2 - 12

---

Named local inner classes reside within a method and are considered part of the method. They are useful if you need a one-off class just for use by a single method.

Note that you cannot use "public" or "private" modifiers on the inner class definition, since such modifiers are only allowed on members defined at class scope.

## Accessing Enclosing Class Members

- Named local inner classes can access any enclosing class field or method *and* method local variables defined as **final**

```
1  public class MyEnclosingClass
2  {
3  . . .
4      public void anotherMethod()
5      {
6          int z = 44;
7          final int q = 32;
8          class Inner4
9          {
10             private int x = 17;
11             public void myMethod()
12             {
13                 System.out.println(x);
14                 System.out.println(q);
15                 System.out.println(MyEnclosingClass.this.x);
16             }
17         }
18 . . .
19 }
```

2 - 13

---

Named local inner classes have basically the same access to fields as do non-static member inner classes, but there's a twist: local inner classes can also access local variables defined in the method in which they reside. Caveat: The local variables must use the final modifier, which means they are constant, and greatly reduces the usefulness of this technique.

## Instantiating Named Local Inner Class Objects

- Since named local inner classes are only visible within the method that defined them, you can use normal **new** operator syntax

```
1  public class MyEnclosingClass
2  {
3      . . .
4      public void anotherMethod()
5      {
6          class Inner4
7          {
8              public void myMethod()
9              {
10                 . . .
11             }
12         }
13
14         Inner4 m5 = new Inner4();
15         m5.myMethod();
16     }
17 }
```

2 - 14

---

Since local inner classes are only visible from within the method that encloses them, you instantiate and use objects typed to the inner class within the enclosing method.

## Intro to Anonymous Local Inner Classes

- Anonymous local inner classes are defined within the boundary of an enclosing class method
- The definition of an anonymous inner class is a Java expression, not a complete statement, so it can reside within an expression (e.g. a call to a method)
- Anonymous inner classes can reference a superclass or a super-interface -- if you specify a super-interface, Object is the implicit superclass

2 - 15

---

Anonymous inner classes might seem a bit weird at first -- how can you define a class without naming it? But they are quite useful, especially in GUI applications. You do have to get used to the syntax, however.

## Writing and Instantiating an Anonymous Local Inner Class

- An anonymous class can reference a superclass -- Java implicitly defines a new class with the specified fields and methods

```
1  public class MyEnclosingClass
2  {
3      public void anotherMethod2()
4      {
5          System.out.println (new Object()
6          {
7              private int x = 17;
8              public String toString()
9              {
10                 return "Inner x: " + x +
11                 " Enclosing x: " + MyEnclosingClass.this.x;
12             }
13         } );
14     }
15     . . .
16 }
```

### 2 - 16

---

For anonymous inner classes, you define the class and instantiate an object all in one step.

Note that we defined the inner class within a call to `System.out.println` -- it's a common scenario to define anonymous classes within method calls, especially registration calls for JavaBean events.

In this case, we subclassed `Object` and override `toString()` -- remember that `System.out.println()` automatically calls `toString()` on any object references passed.

Anonymous inner classes have the same access to enclosing class members and final local variables as do named local inner classes.

## Anonymous Local Inner Class and Interfaces

- Instead of referencing a superclass, you can specify a super-interface -- Java implicitly defines a new class that subclasses Object and implements the interface
- You will need to include all interface methods in the anonymous inner class!

```
1 public class MyEnclosingClass
2 {
3     public void anotherMethod3()
4     {
5         System.out.println (new java.io.Serializable()
6         {
7             public String toString()
8             {
9                 return "Hello";
10            }
11        } );
12    }
13    . . .
14 }
```

2 - 17

---

In this case, we defined an anonymous inner class that subclasses Object and implements the java.io.Serializable interface (which is a tagging interface with no methods).

It does look odd to use the "new" operator on an interface!

## Review Questions

- What are the four categories of inner classes?
- How is a static member inner class different from the other categories?

## Chapter Summary

In this chapter, you learned:

- About the four categories of inner classes
- How to define inner classes and instantiate inner class-typed objects

