**Inner class in java**

<http://www.cis.upenn.edu/~matuszek/General/JavaSyntax/inner-classes.html>

There are four types of inner classes: [**member**](http://www.cis.upenn.edu/~matuszek/General/JavaSyntax/inner-classes.html#member),[**static member**](http://www.cis.upenn.edu/~matuszek/General/JavaSyntax/inner-classes.html#static),[**local**](http://www.cis.upenn.edu/~matuszek/General/JavaSyntax/inner-classes.html#local), and[**anonymous**](http://www.cis.upenn.edu/~matuszek/General/JavaSyntax/inner-classes.html#anonymous).

**A member class** is defined at the top level of the [class](http://www.cis.upenn.edu/~matuszek/General/JavaSyntax/classes.html). It may have the same [access](http://www.cis.upenn.edu/~matuszek/General/JavaSyntax/access.html) modifiers as variables (public, protected, package, static, final), and is accessed in much the same way as [variables](http://www.cis.upenn.edu/~matuszek/General/JavaSyntax/variables.html) of that class.

**public class OuterClass {**

**int outerVariable = 100;**

**class MemberClass {**

**int innerVariable = 20;**

**int getSum(int parameter) {**

**return innerVariable + outerVariable + parameter;**

**}**

**}**

**public static void main(String[] args) {**

**OuterClass outer = new OuterClass();**

**MemberClass inner = outer.new MemberClass();**

**System.out.println(inner.getSum(3));**

**outer.run();**

**}**

**void run() {**

**MemberClass localInner = new MemberClass();**

**System.out.println(localInner.getSum(5));**

**}**

A **static member class** is defined like a member class, but with the keyword [static](http://www.cis.upenn.edu/~matuszek/General/JavaSyntax/static.html). Despite its position inside another class, a **static member class is actually an "outer" class--it has no special access to names in its containing class**. To refer to the static inner class from a class outside the containing class, use the syntax **OuterClassName.InnerClassName**. A static member class may contain static fields and methods.

**public class OuterClass {**

**int outerVariable = 100;**

**static int staticOuterVariable = 200;**

**static class StaticMemberClass {**

**int innerVariable = 20;**

**int getSum(int parameter) {**

**// Cannot access outerVariable here**

**return innerVariable + staticOuterVariable + parameter;**

**}**

**}**

**public static void main(String[] args) {**

**OuterClass outer = new OuterClass();**

**StaticMemberClass inner = new StaticMemberClass();**

**System.out.println(inner.getSum(3));**

**outer.run();**

**}**

**void run() {**

**StaticMemberClass localInner = new StaticMemberClass();**

**System.out.println(localInner.getSum(5));**

**}**

**}**

A **local inner class** is defined within a method, and the usual scope rules apply to it. It is only accessible within that method, therefore access restrictions (**public**, **protected**, **package**) do not apply. However, because objects (and their methods) created from this class may persist after the method returns, a local inner class *may not refer to parameters or non-****final****local variables of the method*.

**public class OuterClass {**

**int outerVariable = 10000;**

**static int staticOuterVariable = 2000;**

**public static void main(String[] args) {**

**OuterClass outer = new OuterClass();**

**System.out.println(outer.run());**

**}**

**Object run() {**

**int localVariable = 666;**

**final int finalLocalVariable = 300;**

**class LocalClass {**

**int innerVariable = 40;**

**int getSum(int parameter) {**

**// Cannot access localVariable here**

**return outerVariable + staticOuterVariable +**

**finalLocalVariable + innerVariable + parameter;**

**}**

**}**

**LocalClass local = new LocalClass();**

**System.out.println(local.getSum(5));**

**return local;**

**}**

**}**

An **anonymous inner class** is one that is declared and used to create one object (typically as a parameter to a method), all within a single statement.

An anonymous inner class may extend a class:

|  |
| --- |
| **new *SuperClass*(*parameters*){ *class body* }** |

Here, ***SuperClass*** is not the name of the class being defined, but rather the name of the class being extended. The ***parameters*** are the parameters to the constructor for that superclass.

An anonymous inner class may implement an interface:

|  |
| --- |
| **new *Interface*(){ *class body* }** |

Because anonymous inner classes are almost always used as event listeners, the example below uses an anonymous inner class as a button listener.

**import java.awt.event.ActionEvent;**

**import java.awt.event.ActionListener;**

**import javax.swing.\*;**

**public class OuterClass extends JFrame {**

**public static void main(String[] args) {**

**OuterClass outer = new OuterClass();**

**JButton button = new JButton("Don't click me!");**

**button.addActionListener(new ActionListener() {**

**public void actionPerformed(ActionEvent event) {**

**System.out.println("Ouch!");**

**}**

**});**

**outer.add(button);**

**outer.pack();**

**outer.setVisible(true);**

**}**

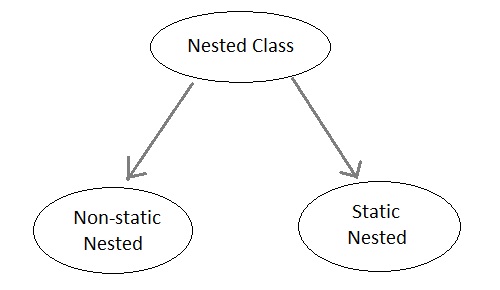
**}**

Because anonymous inner classes occur within a method, they break up the flow and add several lines to the method. Consequently, the actual code within an anonymous inner class should be kept very short--usually a single method call.

<http://www.studytonight.com/java/nested-classes.php>

**Nested Class**

A class within another class is known as Nested class. The scope of the nested is bounded by the scope of its enclosing class.



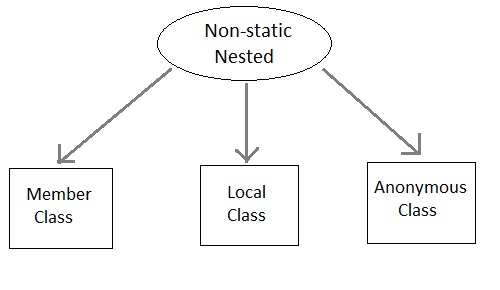
**Static Nested Class**

A static nested class is the one that has **static** modifier applied. **Because it is static it cannot refer to non-static members of its enclosing class directly**. Because of this restriction static nested class is seldom used.

**Non-static Nested class**

Non-static Nested class is most important type of nested class. It is also known as **Inner** class. It has access to all variables and methods of **Outer** class and may refer to them directly. But the reverse is not true, that is,**Outer** class cannot directly access members of **Inner** class.

One more important thing to notice about an **Inner** class is that it can be created only within the scope of**Outer** class. Java compiler generates an error if any code outside **Outer** class attempts to instantiate **Inner**class.



**Example of Inner class**

class Outer

{

public void display()

{

Inner in=new Inner();

in.show();

}

class Inner

{

public void show()

{

System.out.println("Inside inner");

}

}

}

class Test

{

public static void main(String[] args)

{

Outer ot=new Outer();

ot.display();

}

}

**Output:**

Inside inner

**Example of Inner class inside a method**

class Outer {

int count;

public void display() {

for(int i=0;i<5;i++)

{

class Inner //Inner class defined inside for loop

{

public void show()

{

System.out.println("Inside inner "+(count++));

}

}

Inner in=new Inner();

in.show();

}

}

}

class Test {

public static void main(String[] args) {

Outer ot=new Outer();

ot.display();

}

}

**Output:**

Inside inner 0

Inside inner 1

Inside inner 2

Inside inner 3

Inside inner 4

**Example of Inner class instantiated outside Outer class**

class Outer {

int count;

public void display() {

Inner in=new Inner();

in.show();

}

class Inner {

public void show() {

System.out.println("Inside inner "+(++count));

}

}

}

class Test {

public static void main(String[] args)

{

Outer ot=new Outer();

Outer.Inner in= ot.new Inner();

in.show();

}

}

**Output**

Inside inner 1

**Annonymous class**

A class without any name is called Annonymous class.

interface Animal

{

void type();

}

public class ATest {

public static void main(String args[])

{

Animal an = new Animal(){ //Annonymous class created

public void type()

{

System.out.println("Annonymous animal");

}

};

an.type();

}

}

**Output**

Annonymous animal

Here a class is created which implements **Animal** interace and its name will be decided by the compiler. This annonymous class will provide implementation of **type()** method.

<http://stackoverflow.com/questions/70324/java-inner-class-and-static-nested-class>

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.

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();

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

class OuterClass {

...

class InnerClass {

...

}

}

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();

see: [Java Tutorial - Nested Classes](http://download.oracle.com/javase/tutorial/java/javaOO/nested.html)

For completeness note that there is also such a thing as an [inner class without an enclosing instance](http://stackoverflow.com/questions/20468856/is-it-true-that-every-inner-class-requires-an-enclosing-instance):

class A {

int t() { return 1; }

static A a = new A() { int t() { return 2; } };

}

Here, new A() { ... } is an inner class defined in a static context and does not have an enclosing instance.

There are three reasons you might create a nested class:

* organization: sometimes it seems most sensible to sort a class into the namespace of another class, especially when it won't be used in any other context
* access: nested classes have special access to the variables/fields of their containing classes (precisely which variables/fields depends on the kind of nested class, whether inner or static).
* convenience: having to create a new file for every new type is bothersome, again, especially when the type will only be used in one context

There are **four kinds of nested class in Java**. In brief, they are:

* **static class**: declared as a static member of another class
* **inner class**: declared as an instance member of another class
* **local inner class**: declared inside an instance method of another class
* **anonymous inner class**: like a local inner class, but written as an expression which returns a one-off object

In more detail:

**static classes**

Static classes are the easiest kind to understand because they have nothing to do with instances of the containing class.

A static class is a class declared as a static member of another class. Just like other static members, such a class is really just a hanger on that uses the containing class as its namespace, *e.g.* the class*Goat* declared as a static member of class *Rhino* in the package *pizza* is known by the name*pizza.Rhino.Goat*.

package pizza;

public class Rhino {

...

public static class Goat {

...

}

}

Frankly, static classes are a pretty worthless feature because classes are already divided into namespaces by packages. The only real conceivable reason to create a static class is that such a class has access to its containing class's private static members, but I find this to be a pretty lame justification for the static class feature to exist.

**inner classes**

An inner class is a class declared as a non-static member of another class:

package pizza;

public class Rhino {

public class Goat {

...

}

private void jerry() {

Goat g = new Goat();

}

}

Like with a static class, the inner class is known as qualified by its containing class name,*pizza.Rhino.Goat*, but inside the containing class, it can be known by its simple name. However, every instance of an inner class is tied to a particular instance of its containing class: above, the *Goat*created in *jerry*, is implicitly tied to the *Rhino* instance *this* in *jerry*. Otherwise, we make the associated*Rhino* instance explicit when we instantiate *Goat*:

Rhino rhino = new Rhino();

Rhino.Goat goat = rhino.new Goat();

(Notice you refer to the inner type as just *Goat* in the weird *new* syntax: Java infers the containing type from the *rhino* part. And, yes *new rhino.Goat()* would have made more sense to me too.)

So what does this gain us? Well, the inner class instance has access to the instance members of the containing class instance. These enclosing instance members are referred to inside the inner class*via* just their simple names, not *via* *this* (*this* in the inner class refers to the inner class instance, not the associated containing class instance):

public class Rhino {

private String barry;

public class Goat {

public void colin() {

System.out.println(barry);

}

}

}

In the inner class, you can refer to *this* of the containing class as *Rhino.this*, and you can use *this* to refer to its members, *e.g. Rhino.this.barry*.

**local inner classes**

A local inner class is a class declared in the body of a method. Such a class is only known within its containing method, so it can only be instantiated and have its members accessed within its containing method. The gain is that a local inner class instance is tied to and can access the final local variables of its containing method. When the instance uses a final local of its containing method, the variable retains the value it held at the time of the instance's creation, even if the variable has gone out of scope (this is effectively Java's crude, limited version of closures).

Because a local inner class is neither the member of a class or package, it is not declared with an access level. (Be clear, however, that its own members have access levels like in a normal class.)

If a local inner class is declared in an instance method, an instantiation of the inner class is tied to the instance held by the containing method's *this* at the time of the instance's creation, and so the containing class's instance members are accessible like in an instance inner class. A local inner class is instantiated simply *via* its name, *e.g.* local inner class *Cat* is instantiated as *new Cat()*, not new this.Cat() as you might expect.

**anonymous inner classes**

An anonymous inner class is a syntactically convenient way of writing a local inner class. Most commonly, a local inner class is instantiated at most just once each time its containing method is run. It would be nice, then, if we could combine the local inner class definition and its single instantiation into one convenient syntax form, and it would also be nice if we didn't have to think up a name for the class (the fewer unhelpful names your code contains, the better). An anonymous inner class allows both these things:

new \*ParentClassName\*(\*constructorArgs\*) {\*members\*}

This is an expression returning a new instance of an unnamed class which extends*ParentClassName*. You cannot supply your own constructor; rather, one is implicitly supplied which simply calls the super constructor, so the arguments supplied must fit the super constructor. (If the parent contains multiple constructors, the “simplest” one is called, “simplest” as determined by a rather complex set of rules not worth bothering to learn in detail--just pay attention to what NetBeans or Eclipse tell you.)

Alternatively, you can specify an interface to implement:

new \*InterfaceName\*() {\*members\*}

Such a declaration creates a new instance of an unnamed class which extends Object and implements *InterfaceName*. Again, you cannot supply your own constructor; in this case, Java implicitly supplies a no-arg, do-nothing constructor (so there will never be constructor arguments in this case).

Even though you can't give an anonymous inner class a constructor, you can still do any setup you want using an initializer block (a {} block placed outside any method).

Be clear that an anonymous inner class is simply a less flexible way of creating a local inner class with one instance. If you want a local inner class which implements multiple interfaces or which implements interfaces while extending some class other than *Object* or which specifies its own constructor, you're stuck creating a regular named local inner class.

First to get the terms right:

* A nested class is a class which is contained in another class at the source code level.
* It is static if you declare it with the **static** modifier.
* A non-static nested class is called inner class. (I stay with non-static nested class.)

Martin's answer is right so far. However, the actual question is: What is the purpose of declaring a nested class static or not?

You use **static nested classes** if you just want to keep your classes together if they belong topically together or if the nested class is exclusively used in the enclosing class. There is no semantic difference between a static nested class and every other class.

**Non-static nested classes** are a different beast. Similar to anonymous inner classes, such nested classes are actually closures. That means they capture their surrounding scope and their enclosing instance and make that accessible. Perhaps an example will clarify that. See this stub of a Container:

public class Container {

public class Item{

Object data;

public Container getContainer(){

return Container.this;

}

public Item(Object data) {

super();

this.data = data;

}

}

public static Item create(Object data){

// does not compile since no instance of Container is available

return new Item(data);

}

public Item createSubItem(Object data){

// compiles, since 'this' Container is available

return new Item(data);

}

}

In this case you want to have a reference from a child item to the parent container. Using a non-static nested class, this works without some work. You can access the enclosing instance of Container with the syntax Container.this.

More hardcore explanations following:

If you look at the Java bytecodes the compiler generates for an (non-static) nested class it might become even clearer:

// class version 49.0 (49)

// access flags 33

public class Container$Item {

// compiled from: Container.java

// access flags 1

public INNERCLASS Container$Item Container Item

// access flags 0

Object data

// access flags 4112

final Container this$0

// access flags 1

public getContainer() : Container

L0

LINENUMBER 7 L0

ALOAD 0: this

GETFIELD Container$Item.this$0 : Container

ARETURN

L1

LOCALVARIABLE this Container$Item L0 L1 0

MAXSTACK = 1

MAXLOCALS = 1

// access flags 1

public <init>(Container,Object) : void

L0

LINENUMBER 12 L0

ALOAD 0: this

ALOAD 1

PUTFIELD Container$Item.this$0 : Container

L1

LINENUMBER 10 L1

ALOAD 0: this

INVOKESPECIAL Object.<init>() : void

L2

LINENUMBER 11 L2

ALOAD 0: this

ALOAD 2: data

PUTFIELD Container$Item.data : Object

RETURN

L3

LOCALVARIABLE this Container$Item L0 L3 0

LOCALVARIABLE data Object L0 L3 2

MAXSTACK = 2

MAXLOCALS = 3

}

As you can see the compiler creates a hidden field Container this$0. This is set in the constructor which has an additional parameter of type Container to specify the enclosing instance. You can't see this parameter in the source but the compiler implicitly generates it for a nested class.

Martin's example

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

would so be compiled to a call of something like (in bytecodes)

new InnerClass(outerObject)

For the sake of completeness:

An anonymous class **is** a perfect example of a non-static nested class which just has no name associated with it and can't be referenced later.

<http://www.programcreek.com/2009/02/4-inner-classes-tutorial-examples/>

There are 4 different types of inner classes you can use in Java. The following gives their name and examples.

1. Static Nested Classes

|  |
| --- |
| **class** Outer {  **static** **class** Inner {  **void** go() {  System.out.println("Inner class reference is: " + **this**);  }  }  }    **public** **class** Test {  **public** **static** **void** main(String[] args) {  Outer.Inner n = **new** Outer.Inner();  n.go();  }  } |

Inner class reference is: Outer$Inner@19e7ce87

2. Member Inner Class

Member class is instance-specific. It has access to all methods, fields, and the Outer's this reference.

|  |
| --- |
| **public** **class** Outer {  **private** **int** x = 100;    **public** **void** makeInner(){  Inner in = **new** Inner();  in.seeOuter();  }    **class** Inner{  **public** **void** seeOuter(){  System.out.println("Outer x is " + x);  System.out.println("Inner class reference is " + **this**);  System.out.println("Outer class reference is " + Outer.**this**);  }  }  **public** **static** **void** main(String [] args){  Outer o = **new** Outer();  Inner i = o.**new** Inner();  i.seeOuter();  }  } |

Outer x is 100

Inner class reference is Outer$Inner@4dfd9726

Outer class reference is Outer@43ce67ca

3. Method-Local Inner Classes

|  |
| --- |
| **public** **class** Outer {  **private** String x = "outer";    **public** **void** doStuff() {  **class** MyInner {  **public** **void** seeOuter() {  System.out.println("x is " + x);  }  }    MyInner i = **new** MyInner();  i.seeOuter();  }    **public** **static** **void** main(String[] args) {  Outer o = **new** Outer();  o.doStuff();  }  } |

x is outer

|  |
| --- |
| **public** **class** Outer {  **private** **static** String x = "static outer";    **public** **static** **void** doStuff() {  **class** MyInner {  **public** **void** seeOuter() {  System.out.println("x is " + x);  }  }    MyInner i = **new** MyInner();  i.seeOuter();  }    **public** **static** **void** main(String[] args) {  Outer.doStuff();  }  } |

x is static outer

4. Anonymous Inner Classes

This is frequently used when you add an action listener to a widget in a GUI application.

|  |
| --- |
| button.addActionListener(**new** ActionListener(){  **public** **void** actionPerformed(ActionEvent e){  comp.setText("Button has been clicked");  }  }); |