**INNER CLASS**

Inner class is a class which is a member of another class or method

There are four types of inner classes: -------------

* Instance inner class or regular inner class
* Method local inner class
* Anonymous inner class
* Static inner class

**Instance inner class or regular inner class**

1. A regular inner class is defined within the curly braces of another class

, but it should be outside of any method or other code block.

**class** MyOuter {

**private** **int** x = 7;

// inner class definition

**class** MyInner {

**public** **void** seeOuter() {

System.*out*.println("Outer x is " + x);

}

} // close inner class definition

} // close outer class

1. When you compile it,

%javac MyOuter.java

You'll get *two* class files:

MyOuter.class

MyOuter$MyInner.class

1. A *regular* inner class can't have static declarations of any kind.
2. The inner class file can’t be accessible to you in the general way. You can't say

%java MyOuter$MyInner

*The only way you can access the inner class is through a live instance of the outer class!* In other words, only at runtime when there's already an instance of the outer class

1. All the outside member can be accessed inside the inner class directly.
2. All the inner class member can’t be accessed inside the outer class directly but can be accessed with an object
3. We can’t access inner class itself outside the outer class directly.

To create an instance of an inner class, *you must have an instance of the outer class*

1. **Instantiating an Inner Class from Within the Outer Class**

**class** MyOuter {

**private** **int** x = 7;

**public** **void** makeInner() {

MyInner in = **new** MyInner(); // make an inner instance

in.seeOuter();

}

**class** MyInner {

**public** **void** seeOuter() {

System.*out*.println("Outer x is " + x);

}

}

}

MyOuter code treats MyInner just as any other accessible class, no need to create the outer class object.

From inside the outer class instance code, you can instantiate the inner class using only the name of the inner class, as follows:

MyInner mi = new MyInner();

1. **Creating an Inner Class Object from Outside the Outer Class**

If we want to create an instance of the inner class, we must have an instance of the outer class.

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

MyOuter mo = **new** MyOuter(); // gotta get an instance!

**MyOuter.MyInner inner = mo.new MyInner();**

**inner.seeOuter();**

}

Or,

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

**MyOuter.MyInner inner = new MyOuter().new MyInner();**

**inner.seeOuter();**

}

1. From *outside* the outer class instance code (including static method code within the outer class), the inner class name must now include the outer class's name:

MyOuter.MyInner

To instantiate it, you must use a reference to the outer class:

new MyOuter().new MyInner(); or outerObjRef.new MyInner();

**class** MyOuter {

**private** **int** x = 7;

**public** **void** makeInner() {

MyInner in = **new** MyInner();

in.seeOuter();

}

**class** MyInner {

**public** **void** seeOuter() {

System.*out*.println("Outer x is " + x);

System.*out*.println("Inner class ref is " + **this**);

System.*out*.println("Outer class ref is " + MyOuter.**this**);

}

}

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

MyOuter.MyInner inner = **new** MyOuter().**new** MyInner();

inner.seeOuter();

}

}

the output is something like this:

Outer x is 7

Inner class ref is MyOuter$MyInner@113708

Outer class ref is MyOuter@33f1d7

1. To reference the inner class instance itself, from *within* the inner class code, use this.
2. To reference the "*outer* this" (the outer class instance) from within the inner class code, use NameOfOuterClass.this (example, MyOuter.this).
3. A regular inner class is a member of the outer class just as instance variables and methods are, so the following modifiers can be applied to an inner class:

* final
* abstract
* public
* private
* protected
* static—but static turns it into a static nested class not an inner class
* strictfp

final and abstract can never be together.

1. A regular inner class has scope same as an instance variable. (It can be accessed inside another class's curly braces, but outside any method code).
2. Inner class can be used as a helper class for the outer class.
3. We can’t access outer class member outside the outer class with inner classes.

Q. How many inner classes can I write inside the outer class?

Answer: any number of inner classes

Q. Assume that there are two inner class A and B inside the class called Hello, can I write class A extends B

A. Yes

Q. Can I write interfaces inside the class? ---------------- Yes

Q. Can I write inner class inside the interface? ----------- Yes

Q. Can I write interface inside interface? ------------------ Yes

**Class inside interface**

You can define a class inside an interface. Inside the interface, the inner class is implicitly public and static.

From [JLS Section 9.1.4](http://docs.oracle.com/javase/specs/jls/se7/html/jls-9.html#jls-9.1.4):

The body of an interface may declare members of the interface, that is, fields (§9.3), methods (§9.4), **classes** (§9.5), and interfaces (§9.5).

From [JLS Section 9.5](http://docs.oracle.com/javase/specs/jls/se7/html/jls-9.html#jls-9.5):

Interfaces may contain member type declarations (§8.5).

A member type declaration in an interface is **implicitly static and public**. It is permitted to redundantly specify either or both of these modifiers

The only restriction on the inner class defined inside the interface or inside any other class, is that, you have to access them using the enclosing member name.  
Apart from that, there is no relation between them. The inner class will result in completely a different class file after compilation.

**package** com.test;

**interface** TestInterface {

**int** *a* = 10;

**int** *b* = 5;

**void** add();

**void** sub();

**class** Inner {

**void** add() {

**int** c = *a* + *b*;

System.*out*.println("After Addition:" + c);

}

**void** sub() {

**int** c = *a* - *b*;

System.*out*.println("After Subtraction:" + c);

}

}

}

**abstract** **public** **class** Test10 {

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

TestInterface.Inner i = **new** TestInterface.Inner();

i.add();

i.sub();

}

}

**Method local inner class**

1. An inner class within a method:

**class** MyOuter2 {

**private** String x = "Outer2";

**void** doStuff() {

**class** MyInner {

**public** **void** seeOuter() {

System.*out*.println("Outer x is " + x);

} // close inner class method

} // close inner class definition

} // close outer class method doStuff()

} // close outer class

The code above is completely useless, however, because *it never instantiates the inner class!*

1. To *use* the inner class you must make an instance of it somewhere *within the method but below the inner class definition(otherwise compiler won't be able to find the inner class)*

**class** MyOuter2 {

**private** String x = "Outer2";

**void** doStuff() {

**class** MyInner {

**public** **void** seeOuter() {

System.*out*.println("Outer x is " + x);

} // close inner class method

} // close inner class definition

MyInner mi = **new** MyInner(); // This line must come

// after the class

mi.seeOuter();

} // close outer class method doStuff()

} // close outer class

1. A method-local inner class can be instantiated only within the method where the inner class is defined.
2. A method-local inner class can access outer class’ private or any other members.
3. A method-local inner class object cannot use the local variables of the method the inner class is in. Unless the local variables are marked as final. i.e. only final variables of a method can be accessed by the method local inner class.

The local variables of the method live on the stack, and exist only for the lifetime of the method. i.e.the scope of a local variable is limited to the method the variable is declared in.

When the method ends, the stack frame is blown away and the variable is vanished. But even after the method completes, the inner class object created within it might still be alive on the heap

if,for example, a reference to it was passed into some other code and then stored in an instance variable. Because the local variables aren't guaranteed to be alive as long as the method-local inner class object, the inner class object can't use them.

**class** MyOuter2 {

**private** String x = "Outer2";

**void** doStuff() {

String z = "local variable";

**class** MyInner {

**public** **void** seeOuter() {

System.*out*.println("Outer x is " + x);

System.*out*.println("Local variable z is " + z); // Won't Compile!

} // close inner class method

} // close inner class definition

} // close outer class method doStuff()

} // close outer class

Compilation Error:

MyOuter2.java:8: local variable z is accessed from within inner class;

needs to be declared final

System.out.println("Local variable z is " + z);

^

Marking the local variable z as final fixes the problem:

final String z = "local variable"; // Now inner object can use it

1. You can't mark a method-local inner class as **public, private, protected, static, transient**. The only modifiers you can apply to a method-local inner class are **abstract and final**, but as always, never both at the same time.
2. A method local inner class declared in a static method has access to only static members of the enclosing class. If you're in a static method there is no “this”, so an inner class has no access to instance variables.

**Anonymous Inner Classes**

1. Inner classes declared without any class name at all (hence the word anonymous).
2. This inner class can be defined not only inside the method but also within an argument to a method.
3. **Plain-Old Anonymous Inner Classes, Flavor One**

**class** Popcorn {

**public** **void** pop() {

System.*out*.println("popcorn");

}

}

**class** Food {

Popcorn p = **new** Popcorn() {

**public** **void** pop() {

System.*out*.println("anonymous popcorn");

}

};

}

* The Popcorn reference variable refers not to an instance of Popcorn, but to an instance of an anonymous (unnamed) subclass of Popcorn.

Popcorn p = new Popcorn() { // a curly brace, not a semicolon

* Above line can be read as,

Declare a reference variable, p, of type Popcorn. Then declare a new class that has no name, but that is a subclass of Popcorn. And here's the curly brace that opens the class definition…

* Inside the anonymous class Overriding the pop() method of the superclass Popcorn. This is the whole point of making an anonymous inner class—to override one or more methods of the superclass!
* The curly brace closing off the anonymous inner class definition must has a semicolon.
* The compiler will complain if you try to invoke any method on an anonymous inner class reference that is not in the superclass class definition.

**class** Popcorn {

**public** **void** pop() {

System.*out*.println("popcorn");

}

}

**class** Food {

Popcorn p = **new** Popcorn() {

**public** **void** sizzle() {

System.*out*.println("anonymous sizzling popcorn");

}

**public** **void** pop() {

System.*out*.println("anonymous popcorn");

}

};

**public** **void** popIt() {

p.pop(); // OK, Popcorn has a pop() method

p.sizzle(); // Not Legal! Popcorn does not have sizzle()

}

}

Compiling the preceding code gives us something like,

Anon.java:19: cannot resolve symbol

symbol : method sizzle ()

location: class Popcorn

p.sizzle();

^

1. **Plain-Old Anonymous Inner Classes, Flavor Two (anonymous interface implementers)**

* The anonymous inner class would be an implementer of the interface rather than a subclass of the class in case of interface.

**interface** Cookable {

**public** **void** cook();

}

**class** Food {

Cookable c = **new** Cookable() {

**public** **void** cook() {

System.*out*.println("anonymous cookable implementer");

}

};

}

* It’s not instantiating a Cookable object, it's creating an instance of a new, anonymous, implementer of Cookable.
* Anonymous interface implementers— can implement only one interface.

1. The inner class has to choose either to be a subclass of a named class - or not directly implement any interfaces at all—or to implement a single interface. By directly, we mean actually using the keyword implements as part of the class declaration. If the anonymous inner class is a subclass of a class type, it automatically becomes an implementer of any interfaces implemented by the superclass.
2. **Define an anonymous inner class, right inside the argument.**

**class** MyWonderfulClass {

**void** go() {

Bar b = **new** Bar();

**b.doStuff(new Foo() {**

**public** **void** foof() {

System.*out*.println("foofy");

} // end foof method

}); // end inner class def, arg, and b.doStuff stmt.

} // end go()

} // end class

**interface** Foo {

**void** foof();

}

**class** Bar {

**void** doStuff(Foo f) { }

}

1. If they're argument local, they end like this:

});

But if they're just plain-old anonymous classes, then they end like this:

};

**Static Nested Classes**

1. Static nested class is not a static inner class.

It is simply a non-inner (also called "top-level") class scoped within another.

1. A static nested class is simply a class that's a static member of the enclosing class:

**class** BigOuter {

**static** **class** Nested { }

}

The static modifier in this case says that the nested class is a static member of the outer class. That means it can be accessed, as with other static members, without having an instance of the outer class.

**class** BigOuter {

**static** **class** Nest {

**void** go() {

System.*out*.println("hi");

}

}

}

**class** Broom {

**static** **class** B2 {

**void** goB2() {

System.*out*.println("hi 2");

}

}

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

BigOuter.Nest n = **new** BigOuter.Nest(); // both class names

n.go();

B2 b2 = **new** B2(); // access the enclosed class

b2.goB2();

}

}

Output:

hi

hi 2

1. A static nested class does not have access to the instance variables and non-static methods of the outer class.
2. A nested class marked with the static modifier is quite similar to any other non-inner class, except that to access it, code must have access to both the nested and enclosing class.

Static inner class: ------------

* When you write inner class as static then the class area becomes static context.
* Variable and method available in static class will not become automatically as static. If you want static then you’ll have to write member of the class as static.
* If you want to access static inner class static member outside the inner class and inside the outer class, you can access with class name.
* If you want to access instance member as static class you have to create an object in outer class.
* When you want to access outer class member inside the static inner class then those members must be static.
* Instance member of outer class not allowed inside static inner class.
* If you want to access inner class functionality outside the outer class you need to create object with the following syntax:------

Outer.inner oi = new outer. inner();

Q. When a static nested class get loaded in the memory?

Answer:

Nested static class is just like a static method, it won’t be loaded at outer class loading time. It’ll be loaded at the time, when this class is called explicitly by calling its static members or by creating its object.

**public** **class** Test7 {

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

}

**static** {

System.*out*.println("inside outer class static block......");

}

**static** **class** MyTest{

{

System.*out*.println("inside nested static class' static block......");

}

}

Output:

inside outer class static block......

**public** **class** Test7 {

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

MyTest mt = **new** MyTest();

System.*out*.println(MyTest.*inside*);

}

**static** {

System.*out*.println("inside outer class static block......");

}

**static** **class** MyTest {

**public** **static** **final** **int** *inside* = 0;

{

System.*out*.println("inside nested static class' static block......");

}

}

}

Output:

inside outer class static block......

inside nested static class' static block......

0

Q. Why the final local variable can be used inside the method local inner class?

**Benefits of Java Nested Class**

1. If a class is useful to only one class, it makes sense to keep it nested and together. It helps in packaging of the classes.
2. Nested classes increases encapsulation. Note that inner classes can access outer class private members and at the same time we can hide inner class from outer world.
3. Nesting small classes within top-level classes places the code closer to where it is used and makes code more readable and maintainable.

**When should inner classes be used in java?**

When you actually need to write some code that seems like it belongs in it’s very own class. But, at the same time, the code that you want to write needs to be intimately tied to some other class’s code.

An inner class is when a class definition is contained within another class. There are basically two types: static and non-static. The real difference between these are:

* Static inner classes:
  + Are considered "top-level".
  + Do not require an instance of the containing class to be constructed.
  + May not reference the containing class members without an explicit reference.
  + Have their own lifetime.
* Non-Static inner classes:
  + Always require an instance of the containing class to be constructed.
  + Automatically have an implicit reference to the containing instance.
  + May access the container's class members without the reference.
  + Lifetime is *supposed* to be no longer than that of the container.

**Garbage Collection and Non-Static Inner Classes**

Garbage Collection is automatic but tries to remove objects based on whether it thinks they are being used. The Garbage Collector is pretty smart, but not flawless. It can only determine if something is being used by whether or not there is an active reference to the object.

The real issue here is when a Non-Static Inner Class has been kept alive longer than its container. This is because of the implicit reference to the containing class. The only way this can occur is if an object outside of the containing class keeps a reference to the inner object, without regard to the containing object.

This can lead to a situation where the inner object is alive (via reference) but the references to the containing object has already been removed from all other objects. The inner object is, therefore, keeping the containing object alive because it will *always* have a reference to it. The problem with this is that unless it is programmed, there is no way to get back to the containing object to check if it is even alive.

The most important aspect to this realization is that it makes no difference whether it is in an Activity or is a drawable. You will ***always*** have to be methodical when using non-static inner classes and make sure that they never outlive objects of the container. Luckily, if it isn't a core object of your code, the leaks may be small in comparison. Unfortunately, these are some of the hardest leaks to find, because they are likely to go unnoticed until many of them have leaked.

**Solutions: Non-static Inner Classes**

* Gain temporary references from the containing object.
* Allow the containing object to be the only one to keep long-lived references to the inner objects.
* Use established patterns such as the Factory.
* If the inner class does not require access to the containing class members, consider turning it into a static class.
* Use with caution, regardless of whether it is in an Activity or not.