**[A]. Inner Class**

* An inner class instance has access to all members of the outer class, **even those marked private***.*
* The term ‘regular class’ is used to represent a class which is not – static or method-local or anonymous. If any of these 3 are not mentioned explicitly, then inner class is same as regular class
* Class definition of the inner class is given inside the curly braces of the enclosing (outer) class, but outside of any method or other code block.
* When an outer class is compiled - javac Outer.java - we will get two class files – Outer.class and Outer$Inner.class, meaning that a separate class file is generated for each inner class.
* Inner class’s class file cannot be directly accessed (invoke its class file to run the main method in inner class, the way we do for outer or any normal class) without outer class – java **Outer$**Inner.class.
* The only way an inner class can be accessed is through an instance of outer class. To create an instance of an inner class, you must have an instance of the outer classto tie to the inner class. There are **no exceptions to this rule**: An inner class instance can never stand alone without a direct relationship to an instance of the outer class.
* Inner class can be instantiated in two ways –
  + Within outer class (**but not in static scope**) – this is done by simply creating an instance of inner class in outer class and then using that instance to access members of inner class.

Inner in = new Inner();

in.methodName();

**public** **class** CoreDemo {

**class** Inner {

**void** showInner() {

System.***out***.println("in inner class");

}

}

//accessing from within outer class CoreDemo & not static scope

**void** call() {

Inner inObj2 = **new** Inner();

System.***out***.print("From call() --> ");

inObj2.showInner();

}

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

//accessing from within outer class CoreDemo but in static scope, will need explicit reference of outer class

**new** CoreDemo().**new** Inner().showInner();

**new** CoreDemo().call();

}

}

* + From outside the outer class (**this includes static method code within the outer class**) - the inner class name must now include the outer class's name when declaring inner class reference.

Outer.Inner in;

in = out.new Inner(); or in = new Outer().new Inner();

**Note**: Instantiating an inner class is the onlyscenario in which you'll invoke new ***on***an instance as opposed to invoking new to ***construct***an instance.

Also, first method of instantiating cannot be used in static scope because in static scope we will not have implicit reference to the outer class which is needed during instantiation.

* We can use ‘this’ from inside inner class to reference instance of inner class. If we want to use reference of outer class (although normally, the inner class code doesn't need a reference to the outer class, since it already has an implicit one it's using to access the members of the outer class, it would need a reference to the outer class if it needed to pass that reference to some other code) we use ‘OuterClassName.this’.
* Modifiers that can be applied to inner class are – final, abstract, public, private, protected, static (***but static turns it into a static nested class, not an inner class***), strictfp.

**[B]. Method-Local Inner Class**

* Inner class (aka regular inner class) is scoped inside another class's curly braces, but outside any method code (in other words, at the same level that an instance variable is declared).
* But we can also define an inner class within a method thus making it a Method-Local Inner Class. Keep in mind that only declaring the class inside the function (inside outer class) is not sufficient, you need to **also instantiate** the class in order to be able to use it and most importantly, the **instance should be created in the same function where the class is declared/defined but below the class’s definition** or the compiler won’t be able to find the class.
* No other code running in any other method (inside or outside the outer class) can ever instantiate the Method-Local Inner Class.
* Like Inner Class, Method-Local Inner Class can access members of outer class including private ones. However, the inner class object cannot use the local variables of the method the inner class is in. The local variables of the method live on the stack and exist only for the lifetime of the method while the object of Method-Local Inner Class created in this method will be created in heap. So, when the method ends, the variable is history but 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 is, the inner class object can't use them. **Unless the local variables are marked final!**
* We cannot mark a Method-Local Inner Class public, private, protected, static, transient, and the like. It can only be abstract and final, but, as always, never both at the same time.
* Till JDK 7, Local-Method Inner Class could access only final local variable of the enclosing block. However, from JDK 8, it is possible to access the non-final local variable of enclosing method in local inner class.
* A method-local class declared in a static method has access to only static members of the outer class, since the method-local class has no associated instance of the outer class. If you're in a static method, there is no this, so an inner class in a static method is subject to the same restrictions as the static method. In other words, no access to instance variables.
* Local inner classes can extend an abstract class or can also implement an interface.
* Local Inner Classes are the inner classes that are defined inside a block. Generally, this block is a method body. Sometimes this block can be a for loop, or an if clause. Local inner classes are not a member of any enclosing classes. They belong to the block they are defined within, **due to which** local inner classes cannot have any access modifiers associated with them, except final or abstract.

**[C]. Anonymous Inner Class**

* Anonymous Inner Classes are inner class without any class name (hence, the word anonymous).
* We can define these classes not just within a method, but even within an argument to a method.
* The whole point of making an anonymous inner class is to override **one or more** methods of the superclass or to implement methods of an interface.
* Unlike Lambda Expressions which can implement only functional interfaces (interfaces with only 1 method), Anonymous classes can implement an interface with any number of abstract methods.
* When the anonymous class is defined as an implementer of an interface, keep in mind that the anonymous class can implement only one interface. In fact, an anonymous inner class can't even extend a class and implement an interface at the same time. The inner class has to choose either to be a subclass of a named class (and 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.

**Anonymous Inner Class Type 1:**

**class** Popcorn {

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

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

}

}

**class** Food {

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

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

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

}

};

}

🡪 Line ‘Popcorn p = **new** Popcorn() {’ shows Anonymous Inner Class in action. What this line does is it first creates a reference variable ‘p’ of type ‘Popcorn’ and then declares a new class (without any name) which is a subclass of ‘Popcorn’. Subsequent lines define the subclass. When this class is closed notice that there is a ‘;’ after ‘}’. So what we accomplish with these lines is that a Popcorn reference variable ‘p’ is created and initialized and it refers to a brand new instance of a brand new, just-in-time, anonymous subclass of ‘Popcorn’.

🡪 Polymorphism is in play here as ‘p’ is a reference of a superclass ‘Popcorn’ and is instantiated with the object of an anonymous subclass. As such, all polymorphism related constraints on which functions can be called and which can’t (ones present in subclass but not in parent class) holds.

Anonymous class with Polymorphism –

**class** Test {

**void** show() {

System.***out***.println("show() in Test");

}

**void** show2() {

System.***out***.println("show2() in Test");

}

}

**public** **class** CoreDemo {

**void** helperMeth(Test t) {

System.***out***.println("Calling Test.show() from helperMeth()");

t.show();

System.***out***.println("Calling Test.show2() from helperMeth()");

t.show2();

}

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

CoreDemo d = **new** CoreDemo();

d.helperMeth(**new** Test() {

**public** **void** show() {

System.***out***.println("Overridden show() as defined by Anonymous class");

}

});

}

}

Output –

Calling Test.show() from helperMeth()

Overridden show() as defined by Anonymous class

Calling Test.show2() from helperMeth()

show2() in Test

**Anonymous Inner Class Type 2 - Interface:**

**interface** Cookable {

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

}

**class** Food {

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

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

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

}

};

}

The only difference between Type 1 and Type 2 is that Type 1 creates an anonymous **subclass** of the specified **class** type, whereas Type 2 creates an anonymous **implementer** of the specified **interface** type. Also, note that this is the only time you will ever see a syntax as - ‘**new** Cookable()’ - where ‘Cookable’ is an interface because we cannot instantiate an interface. What we are doing here is not instantiating an interface, we are creating a sub-class that is implementing the interface and we are creating an instance of this sub-class and assigning it to the reference of interface type.

**Anonymous Inner Class Type 3 - Argument-Defined Anonymous Inner Class:**

* These are inner classes defined and instantiated in a methods argument. This need arises when we have to invoke a function which expects one argument (there may be other arguments) of an interface type and we do not have any class implementing the interface and so we do not have any object either to pass in the function.
* We can create a sub-class and its object and pass this object as an argument to the functions or, we can directly create (and pass) an Argument-Defined Anonymous Inner Class.

**interface** Test {

**void** show();

}

**public** **class** CoreDemo {

**void** helperMeth(Test t) {

System.***out***.println("Calling Test.show() from helperMeth()");

t.show();

}

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

CoreDemo d = **new** CoreDemo();

d.helperMeth(***new*** *Test() {*

***public void*** *show() {*

*System.****out****.println("show() as defined by Anonymous class");*

*}//show ends*

*});//Anonymous class ends, parameter list ends, call to helperMeth is terminated with ;*

}

}

Part in italics and underlined is Anonymous class code.

helperMeth() needs an argument of type Test viz. an interface. So instead of creating an implementer class and passing its object to helperMeth() we are directly passing an Anonymous class. On passing Anonymous class, helperMeth() gets a reference to an object which implements the Test interface and its abstract method show().

Note that the ‘;’ is not added after anonymous class ends, rather

its added after the call to ‘doStuff()’ ends.

**[D]. Static Nested Class:**

* Just as a static method does not have access to the instance variables and non-static methods of the class, a static nested class does not have access to the instance variables and non-static methods of the outer class.
* It is simply a non-inner (also called "top-level") class scoped within another. A static nested class is simply a class (there is no such thing as static class) that's a static member of the enclosing class. That means it can be accessed, as with other static members, without having an instance of the outer class.

**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(); // outer class not needed

b2.goB2();

}

}

**public** **class** CoreDemo {

**static** **class** Inner {

**void** showInner() {

System.***out***.println("in inner class");

}

}

// accessing from outer class

**void** call() {

// object is needed to access the static class’s function because the function is not static. Class is static and so we don’t need

Any reference of outer class for accessing it.

Inner inObj2 = **new** Inner();

System.***out***.print("From call() --> ");

inObj2.showInner();

}

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

**new** Inner().showInner();

**new** CoreDemo().call();

}

}

* If the static nested class has static functions, then accessing them is done in usual static way. Because the nested class is static, it does not need any object/reference for accessing and because its function is also static now, object/reference of static nested class is also not needed.

**public** **class** CoreDemo {

**static** **class** Inner {

**static** **void** showInner() {

System.***out***.println("in inner class");

}

}

**void** call() {

Inner inObj2 = **new** Inner();

System.***out***.print("From call() --> ");

Inner.*showInner*();

}

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

//new Inner().showInner(); this also works but it gives a warning – “The static method showInner() from the typeCoreDemo.Inner should be accessed in a static way”.

Inner.*showInner*();

**new** CoreDemo().call();

}

}

🡪 29th January, 2019.

**Code Snippets:**

1. cannot create object of inner class in main without explicit reference of enclosing class.

**public** **class** CoreDemo {

**class** Inner {

**void** showInner() {

System.***out***.println("in inner class");

}

}

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

Inner inObj = **new** Inner();

}

}

Output: Error message – “No enclosing instance of type CoreDemo is accessible. Must qualify the allocation with an enclosing instance of type CoreDemo (e.g. x.new A() where x is an instance of CoreDemo).”

Note: main is static and so does not have implicit reference of CoreDemo i.e. outer class. To make this work we add the reference/name of the outer class - CoreDemo.Inner inObj = new CoreDemo().new Inner();

**[E]. Lambda Expresssion:**

* Lambda expressions are used to implement only 1 method of the functional interface. Unlike Anonymous Classes, these cannot implement more than 1 method.
* Lambda expression need the ‘->’ operator.
* Lambda expression contexts can be following - variable initialization, assignment, argument passing, casts, the ? operator, array initializers, return statements and lambda expressions, themselves.
* Lambda expression can throw an exception. However, it if throws a checked exception then that exception must be compatible with the exception(s) listed in the throws clause of the abstract method in the functional interface.
* A lambda expression can use an instance or static variable defined by its enclosing class. A lambda expression also has access to this (both explicitly and implicitly), which refers to the invoking instance of the lambda expression’s enclosing class.
* When a lambda expression uses a local variable from its enclosing scope, a special situation is created that is referred to as a **variable capture**. In this case, a lambda expression may only use local variables that are effectively final. An effectively final variable is one whose value does not change, **either inside the lambda or outside of it**, after it is first assigned.
* Lambda expression can use and modify an instance variable from its invoking class. It just can’t use a local variable of its enclosing scope unless that variable is effectively final.
* An important feature related to Lambda expression is Method Reference. Method reference provides a way to refer to a function without executing it. It is following types –
  + Method reference to static methods.
  + Method reference to instance methods.
  + Method reference to generic methods.
  + Constructor reference.