

Agenda

1. Introduction.
2. Normal or Regular inner classes
 - o Accessing inner class code from static area of outer class
 - o Accessing inner class code from instance area of outer class
 - o Accessing inner class code from outside of outer class
 - o The applicable modifiers for outer & inner classes
 - o Nesting of Inner classes
3. Method Local inner classes
4. Anonymous inner classes
 - o Anonymous inner class that extends a class
 - o Anonymous Inner Class that implements an interface
 - o Anonymous Inner Class that define inside method arguments
 - o Difference between general class and anonymous inner classes
 - o Explain the application areas of anonymous inner classes ?
5. Static nested classes
 - o Comparison between normal or regular class and static nested class ?
6. Various possible combinations of nested class & interfaces
 - o class inside a class
 - o interface inside a class
 - o interface inside a interface
 - o class inside a interface
7. Conclusions

=> Sometimes we can declare a class inside another class such type of classes are called inner classes.

SUNMS came up with JDK1.0 in 1991 with features like

1. Platform Independent
2. Robust
3. OOPs
4. Secured
5. Simple
- ..
- ..
- ..

But java was not upto the mark in 2 areas

- a. Performance was not upto the mark
- b. GUI concepts(AWT)

Performance was improved through "JIT compiler".

=> Sun people introduced inner classes in 1.1 version as part of "EventHandling" to resolve GUI bugs.

=> But because of powerful features and benefits of inner classes slowly the programmers starts using in regular coding also.

=> Without existing one type of object if there is no chance of existing another type of object then we should go for inner classes.

Eg1

Without existing University object there is no chance of existing Department object

hence we have to define Department class inside University class.

```
class University{//Outer class
    class Department{//Inner class

    }
}
```

Eg2

Without existing Bank object there is no chance of existing Account object hence we

have to define Account class inside Bank class.

```
class Bank{//Outer class
    class Account{//Inner class

    }
}
```

Eg3:

Without existing Car Object, there is no chance of existing Engine Object hence we have to

define Engine class Inside Car class.

```
class Car{//Outer class
    class Engine{//Inner class

    }
}
```

Eg4:

Without existing Map object there is no chance of existing Entry object hence Entry interface is define inside Map interface.

Map is a collection of key-value pairs, each key-value pair is called an Entry.

```
interface Map{//Outer interface
    interface Entry<K,V>{//Inner interface

    }
}
```

Note:

The relationship between outer class and inner class is not IS-A relationship and it is Has-A relationship.

It promotes Composition/Aggregation.

Based on the purpose and position of declaration all inner classes are divided into 4 types.

They are:

1. Normal or Regular inner classes.
2. Method Local inner classes.
3. Anonymous inner class.
4. static nested classes.

Regular inner class

=====

If we are declaring any named class inside another class directly without static modifier such type of inner classes are called normal or regular inner classes.

Example:

```
Outer.java
class Outer{
    class Inner{

    }
}
```

Output:

```
javac Outer.java
Outer.class
Outer$Inner.class
```

```

java Outer
    NoSuchMethodError/main method not found
java Outer$Inner
    NoSuchMethodError/main method not found

```

Example:

```

class Outer{
    class Inner{
    }
    public static void main(String[] args){
        System.out.println("outer class main method");
    }
}

```

Output:

```

javac Outer
java Outer
    outer class main method
java Outer$Inner
    NoSuchMethodError/main method not found

```

=> Inside inner class we can't declare static members. Hence it is not possible to declare main() method and we can't invoke inner class directly from the command prompt.

Example

```

class Outer {
    class Inner{
        public static void main(String[] args) {
            System.out.println("Inner class main method");
        }
    }
}

```

Output: CE: can't declare static inside Inner class.

Accessing inner class code from static area of outer class:

```

=====
class Outer {
    class Inner{
        public void m1(){
            System.out.println("inner class instance m1()");
        }
    }
    public static void main(String[] args) {
        Outer o= new Outer();
        Outer.Inner i=o.new Inner();
        i.m1();
    }
}

```

Output

```

=====
javac Outer
java Outer
    inner class instance m1()

```

Note:

Scenario1:

```

Outer.Inner i= new Outer().new Inner();
i.m1();

```

Scenario2:

```
new Outer().new Inner().m1();
```

Accessing inner class code from instance area of outer class

```
=====
class Outer {
    class Inner{
        public void m1(){
            System.out.println("inner class instance m1()");
        }
    }
    public void m2(){
        Inner i= new Inner();
        i.m1();
    }
    public static void main(String[] args) {
        Outer o= new Outer();
        o.m2();
    }
}
javac Outer
java Outer
    inner class instance m1()
```

Accessing inner class code from outside of outer class

```
=====
class Outer {
    class Inner{
        public void m1(){
            System.out.println("inner class instance m1()");
        }
    }
}
class Test{
    public static void main(String[] args) {
        Outer o= new Outer();
        Outer.Inner i=o.new Inner();
        i.m1();
    }
}
}
Output
=====
javac Test
java Test
    inner class instance m1()
```

=>From inner class we can access all members of outer class (both static and non-static, private and non private methods and variables) directly.

example

```
class Outer {
    int x=10;
    static int y=20;
    class Inner{
        public void m1(){
            System.out.println(x);
        }
    }
}
```

```

        System.out.println(y);
    }
}
public static void main(String[] args) {
    new Outer().new Inner().m1();
}
}
javac Outer
java Outer
10
20

```

eg#2.

```

class Outer {
    int y=10;
    class Inner{
        int y=100;
        public void m1(){
            int z=1000;
            System.out.println(x);
            System.out.println(y);
            System.out.println(z);
        }
    }
    public static void main(String[] args) {
        new Outer().new Inner().m1();
    }
}

```

=> Within the inner class "this" always refers current inner class object. To refer current outer class object we have to use "outer class name.this".

```

class Outer {
    int x=10;
    class Inner{
        int x=100;
        public void m1(){
            int x=1000;
            System.out.println(x);
            System.out.println(this.x);
            System.out.println(Outer.this.x);
        }
    }
    public static void main(String[] args) {
        new Outer().new Inner().m1();
    }
}

```

Output

=====

```

javac Outer
java Outer
1000
100
10

```

The applicable modifiers for outer classes are:

1. public
2. default
3. final

- 4. abstract
- 5. strictfp

But for the inner classes in addition to this the following modifiers also allowed.

- 1.private
- 2.protected
- 3.static

Nesting of Inner classes :

We can declare an inner class inside another inner class

```
class A {
    class B{
        class C{
            public void m1(){
                System.out.println("C class method");
            }
        }
    }
}
class Test{
    public static void main(String[] args) {
        A a = new A();
        A.B b = a.new B();
        A.B.C c = b.new C();
        c.m1();
    }
}
javac Test
java Test
C class method
```

Note:

```
new A().new B().new C().m1();
```

Method local inner classes:

=> Sometimes we can declare a class inside a method such type of inner classes are called method local inner classes.

=> The main objective of method local inner class is to define method specific repeatedly required functionality.

=> Method Local inner classes are best suitable to meet nested method requirement.

=> We can access method local inner class only within the method where we declared it.

That is from outside of the method we can't access. As the scope of method local inner classes is very less,

this type of inner classes are most rarely used type of inner classes.

eg::

```
class Outer {
    public void m1(){
        class Inner{
            public void sum(int x,int y){
                System.out.println("The sum is ::"+(x+y));
            }
        }
        new Inner().sum(10,20);
        ////
        ////
        ////
    }
}
```

```

        new Inner().sum(100,200);
        ;;;;
        ;;;;
        ;;;;
        ;;;;
        new Inner().sum(1,2);
    }
    public static void main(String[] args) {
        new Outer().m1();
    }
}
javac Outer
java Outer
The sum is :: 30
The sum is :: 300
The sum is :: 3

```

=> If we are declaring inner class inside instance method then we can access both static and non static members of outer class directly.
=> But if we are declaring inner class inside static method then we can access only static members of outer class directly and we can't access instance members directly.

eg#1.

```

class Outer {
    int x=10;
    static int y=20;
    public void m1(){
        class Inner{
            public void m2(){
                System.out.println(x);
                System.out.println(y);
            }
        }
        Inner i=new Inner();
        i.m2();
    }
    public static void main(String[] args) {
        new Outer().m1();
    }
}

```

Output
10
20

case2:

if we declare inner class inside static method then it would result in CE.

=> If we declare methodOne() method as static then we will get compile time error saying "non-static variable x cannot be referenced from a static context".

=> From method local inner class we can't access local variables of the method in which we declared it.

But if that local variable is declared as final then we won't get any compile time error.

eg#1.

```

class Outer {

```

```

        public void m1(){
            final int x=10;
            class Inner{
                public void m2(){
                    System.out.println(x);
                }
            }
            Inner i=new Inner();
            i.m2();
        }
        public static void main(String[] args) {
            new Outer().m1();
        }
    }
}

```

CaseStudy

=====

```

class Outer {
    int i=10;
    static int j=20;
    public void m1(){
        int k=30;
        final int l=40;
        class Inner{
            public void m2(){
                //line-1
            }
        }
        Inner i=new Inner();
        i.m2();
    }
    public static void main(String[] args) {
        new Outer().m1();
    }
}

```

At line-1 how many variables we can access?

i,j,l

if we make m1() as static how many variables we can access?

j,l

if we make m2() as static how many variables we can access

compile time error.

The only applicable modifiers for method local inner classes are:

1. final
2. abstract
3. strictfp

Anonymous inner classes:

=> Sometimes we can declare inner class without name such type of inner classes are called anonymous inner classes.

=> The main objective of anonymous inner classes is "just for instant use".

=> There are 3 types of anonymous inner classes

1. Anonymous inner class that extends a class.
2. Anonymous inner class that implements an interface.
3. Anonymous inner class that defined inside method arguments.

1. Anonymous inner class that extends a class.

```
class PopCorn{
    public void taste(){
        System.out.println("spicy");
    }
}
class Test {
    public static void main(String[] args) {
        PopCorn p=new PopCorn()
        {
            public void taste(){
                System.out.println("salty");
            }
        };
        p.taste();//salty
        PopCorn p1=new PopCorn()
        {
            public void taste(){
                System.out.println("spicy");
            }
        };
        p1.taste();//spicy
    }
}
```

Analysis:

1. PopCorn p=new PopCorn();
We are just creating a PopCorn object.

2. PopCorn p=new PopCorn(){
};

We are creating child class without name for the PopCorn class and for that child class we are creating an object with Parent PopCorn reference.

2. PopCorn p=new PopCorn()
{
 @Override
 public void taste(){
 System.out.println("salty");
 }
};

1. We are creating child class for PopCorn without name.

2. We are overriding taste() method.

3. We are creating object for that child class with parent reference.

Note:

Inside Anonymous inner classes we can take or declare new methods but outside of anonymous inner classes we can't call these methods directly because we are depending on parent reference.[parent reference can be used to hold child class object but by using that reference we can't call child specific methods]. These methods just for internal purpose only.

eg#1.

Example 1:

```
class PopCorn{
    public void taste(){
        System.out.println("spicy");
    }
}
class Test {
    public static void main(String[] args) {
        PopCorn p=new PopCorn()
    }
}
```

```

        public void taste(){
            methodOne();//valid call(internal purpose)
            System.out.println("salty");
        }
        public void methodOne(){
            System.out.println("child specific method");
        }
    };
    //p.methodOne();//here we can not call(outside inner class)
    p.taste();//salty
    PopCorn p1=new PopCorn();
    p1.taste();//spicy
}

```

Output:
 Child specific method
 Salty
 Spicy

Example 2:

```

class Test {
    public static void main(String[] args){
        Thread t=new Thread(){
            public void run(){
                for(int i=0;i<10;i++){
                    System.out.println("child thread");
                }
            }
        };
        t.start();
        for(int i=0;i<10;i++){
            System.out.println("main thread");
        }
    }
}

```

Anonymous Inner Class that implements an interface

=====

Example:

```

class InnerClassesDemo{
    public static void main(String[] args) {
        Runnable r=new Runnable(){ //here we are not creating for Runnable
            interface are creating implements class object.

            public void run(){
                for(int i=0;i<10;i++){
                    System.out.println("Child thread");
                }
            }
        };
        Thread t=new Thread(r);
        t.start();
        for(int i=0;i<10;i++){
            System.out.println("Main thread");
        }
    }
}

```

Anonymous Inner Class that define inside method arguments

=====

Example:

```
class Test {
    public static void main(String[] args) {
        new Thread(new Runnable(){
            public void run(){
                for(int i=0;i<10;i++){
                    System.out.println("child thread");
                }
            }
        }).start();
        for(int i=0;i<10;i++){
            System.out.println("main thread")}
    }
}
```

Output:

=> This output belongs to example 2, anonymous inner class that implements an interface example and anonymous inner class that define inside method arguments example.

```
Main thread
Main thread
Main thread
Main thread
Main thread
Main thread
Main thread
Main thread
Main thread
Main thread
Main thread
Main thread
Child thread
Child thread
Child thread
Child thread
Child thread
Child thread
Child thread
Child thread
Child thread
```

lambda Expression

=====

```
new Thread(
    ()-> {
        for (int i =1;i<=5;i++ )
        {
            System.out.println("child thread");
        }
    }
).start();
```

When we use lambda expression, code would be compact and no .class file for lambda expression.

Difference between general class and anonymous inner classes:

General Class

- 1) A general class can extend only one class at a time.
- 2) A general class can implement any no. of interfaces at a time.
- 3) A general class can extend a class and can implement an interface simultaneously.
- 4) In normal Java class we can write constructor because we know name of the class.

Anonymous Inner class

1. Of course anonymous inner class also can extend only one class at a time.
- 2) But anonymous inner class can implement only one interface at a time.
- 3) But anonymous inner class can extend a class or can implement an interface but not both simultaneously.
- 4) But in anonymous inner class we can't write constructor because anonymous inner class not having any name of the class.

static nested classes

=====

=> Sometimes we can declare inner classes with static modifier such type of inner classes are called static nested classes.

=> In the case of normal or regular inner classes without existing outer class object there is no chance of existing inner class object.

i.e., inner class object is always strongly associated with outer class object.

=> But in the case of static nested class without existing outer class object there may be a chance of existing static nested class object.

i.e., static nested class object is not strongly associated with outer class object.

eg#1.

Example:

```
class Test {
    static class Nested{
        public void methodOne(){
            System.out.println("nested class method");
        }
    }
    public static void main(String[] args){
        //Test.Nested t=new Test.Nested();
        //t.methodOne();

        Nested n=new Nested();
        n.methodOne();
    }
}
```

=> Inside static nested classes we can declare static members including main() method also.

Hence it is possible to invoke static nested class directly from the command prompt.

Example:

```
class Test {
    static class Nested{
        public static void main(String[] args){
            System.out.println("nested class main method");
        }
    }
}
```

```

    }
    public static void main(String[] args){
        System.out.println("outer class main method");
    }
}
java Test
outer class main method
java Test$Nested
nested class main method

```

=> From the normal inner class we can access both static and non static members of outer class but from static nested class we can access only static members of outer class.

Example:

```

class Test {
    int x=10;
    static int y=20;
    static class Nested{
        public void methodOne(){
            System.out.println(x);//C.E:non-static variable x cannot be
reference from a static context
            System.out.println(y);
        }
    }
}
javac Test.java (Test.class, Test$Nested.class)

```

Comparison between normal or regular class and static nested class ?

Normal /regular inner class

1) Without existing outer class object there is no chance of existing inner class object.

That is inner class object is always associated with outer class object.

2) Inside normal or regular inner class we can't declare static members.

3) Inside normal inner class we can't declare main() method and hence we can't invoke regular inner class directly from the

command prompt.

4) From the normal or regular inner class we can access both static and non static members of outer class directly.

Static nested class

1) Without existing outer class object there may be a chance of existing static nested class object.

That is static nested class object is not associated with outer class object.

2) Inside static nested class we can declare static members.

3) Inside static nested class we can declare main() method and hence we can invoke static nested class directly from the

command prompt.

4) From static nested class we can access only static members of outer class directly

Various possible combinations of nested class & interfaces :

1. class inside a class :

We can declare a class inside another class

Without existing one type of object, if there is no chance of existing another type

of object, then we should go for class inside a class.

```

class University {
    class Department {
    }
}

```

Without existing University object, there is no chance of existing Department object.

i.e., Department object is always associated with University.

2. interface inside a class :

We can declare interface inside a class

```

class X {
    interface Y {
    }
}

```

Inside class if we required multiple implementation of an interface and these implementations of relevant to a particular class, then we should declare interface inside a class.

```

class VehicleType {
    interface Vehicle {
        public int getNoOfWheels();
    }
    class Bus implements Vehicle {
        public int getNoOfWheels() {
            return 6;
        }
    }
    class Auto implements Vehicle {
        public int getNoOfWheels() {
            return 3;
        }
    }
}

```

3. interface inside a interface :

We can declare an interface inside another interface.

```

interface Map {
    interface Entry {
        public Object getKey();
        public Object getValue();
        public Object setValue(Object newValue);
    }
}

```

Nested interfaces are always public,static whether we are declaring or not. Hence we can implements inner interface directly with out implementing outer interface.

```

interface Outer {
    public void methodOne();
    interface Inner {
        public void methodTwo();
    }
}
class Test implements Outer.Inner {
    public void methodTwo() {
        System.out.println("Inner interface method");
    }
    public static void main(String args[]) {
        Test t=new Test();
    }
}

```

```

        t.methodTwo();
    }
}

```

Whenever we are implementing Outer interface , it is not required to implement Inner interfaces.

```

class Test implements Outer {
    public void methodOne() {
        System.out.println("Outer interface method ");
    }
    public static void main(String args[]) {
        Test t=new Test();
        t.methodOne();
    }
}

```

i.e., Both Outer and Inner interfaces we can implement independently.

4. class inside a interface :

We can declare a class inside interface. If a class functionality is closely associated with

the user interface then it is highly recommended to declare class inside interface.

Example:

```

interface EmailServer {
    public void sendEmail(EmailDetails e);
    class EmailDetails {
        String from;
        String to;
        String subject;
    }
}

```

In the above example EmailDetails functionality is required for EmailService and we are

not using anywhereelse . Hence we can declare EmailDetails class inside EmailServiceinterface .

We can also declare a class inside interface to provide default implementation for that interface.

Example :

```

interface Vehicle {
    public int getNoOfWheels();
    class DefaultVehicle implements Vehicle {
        public int getNoOfWheels() {
            return 2;
        }
    }
}
class Bus implements Vehicle {
    public int getNoOfWheels() {
        return 6;
    }
}
class Test {
    public static void main(String args[]) {
        Bus b=new Bus();
        System.out.println(b.getNoOfWheels()); //2
    }
}

```

```

        Vehicle.DefaultVehicle d=new Vehicle.DefaultVehicle();
        System.out.println(d.getNoOfWheels());//3
    }
}

```

In the above example DefaultVehicle in the default implementation of Vehicle interface where as Bus customized implementation of Vehicle interface. The class which is declared inside interface is always static ,hence we can create object directly without having outer interface type object.

Conclusions :

1. We can declare anything inside any thing with respect to classes and interfaces.

```

class A{
    class B{

    }
}
class A{
    interface B{//static

    }
}
interface A{
    interface B{//public static

    }
}
interface A{
    class B{//public static

    }
}
}

```

2. Nesting interfaces are always public, static whether we are declaring or not.

3. class which is declared inside interface is always public,static whether we are declaring or not

4. interface declared inside a class is always static,but need not be public.

