Object oriented programming concepts

Data Hiding

Data hiding is a concept where we ensure that our internal data should not go out directly that is an outside person can't access our internal data directly. By using private modifier, we can implement data hiding. The main advantage of data hiding is security.

Example

```
class Account
{
        private double balance;
        ..... ,
        ..... ,
}
```

Abstraction

Abstraction means hiding internal implementation and just presenting or to highlight the set of services, is called abstraction.

Example

By using ATM GUI screen bank people are highlighting the set of services what they are offering without highlighting internal implementation.

The main advantages of Abstraction are

1. We can achieve security as we are not highlighting our internal implementation.
2. Enhancement will become very easy because without effecting end user we can able to perform any type of changes in our internal system.
3. It provides more flexibility to the end user to use system very easily.
4. It improves maintainability of the application.

Encapsulation

It is the process of Encapsulating data and corresponding methods into a single module. If any java class follows data hiding and abstraction such type of class is said to be encapsulated class.

Encapsulation= Datahiding +Abstraction

Tightly encapsulated class

A class is said to be tightly encapsulated if and only if every variable of that class declared as private whether the variable has get and set methods or not and whether these methods declared as public or not, not required to check.

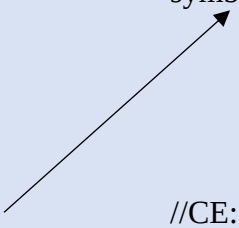
IS-A Relationship(inheritance)

IS-A is also known as inheritance. It is implemented using extends keyword. The main advantage of IS-A relationship is reusability.

```
class Parent
{
        public void methodOne()
        {
        }
}
class Child extends Parent
{
        public void methodTwo()
        {
```

```
        }
}
class Test
{
        public static void main(String[] args)
        {
                Parent p=new Parent();
                p.methodOne();
                p.methodTwo();                  //CE: cannot find symbol
                Child c=new Child();            symbol:method methodTwo()
                c.methodOne();
                c.methodTwo();
                Parent pl=new Child()·
                p1.method0ne();
                pt.methodTwo();
                Child cl=new Parent();          //CE: incompatible types

        }
}
```



Conclusion

- Whatever the parent has by default available to the child but whatever the child has by default not available to the parent. Hence on the child reference we can call both parent and child class methods. But on the parent reference we can call only methods available in the parent class and we can't call child specific methods.
- Parent class reference can be used to hold child class object but by using that reference we can call only methods available in parent class and child specific methods we can't call.
- Child class reference cannot be used to hold parent class object.

Multiple inheritance

- For all java classes the most commonly required functionality is define inside object class hence object class acts as a root for all java classes.
- Having more than one Parent class at the same level is called multiple inheritance.
- Any class can extends only one class at a time and can't extends more than one class simultaneously hence java won't provide support for multiple inheritance.

Example

```
class A{}
class B{}
class C extends A,B             (invalid)
{}
```

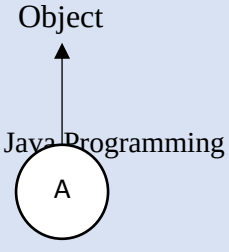
- But an interface can extends any no. Of interfaces at a time.
- java provides support for multiple inheritance through interfaces.

Example

```
interface A{}
interface B{}
interface C extends A,B
{}
```

If our class doesn't extends any other class then only our class is the direct child class of object.

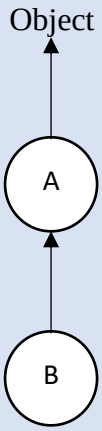
```
class A{}
```



If our class extends any other class then our class is not direct child class of object.

Exampe

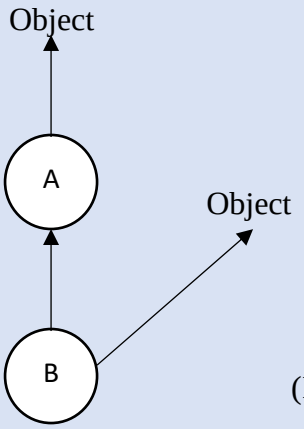
class B{}
class A extends B{}



Multilevel Inheritance

Why java won't provide support for multiple inheritance.

There may be a chance of raising ambiguity problems.



(Invalid)

Why ambiguity problem won't be there in interfaces

Interfaces have only dummy declarations and they won't have implementations therefore no ambiguity will be there.

Cyclic inheritance

Cyclic inheritance is not allowed in java.

Example

```
class A extends B{}
class B extends A{}
```

HAS-A relationship

HAS-A relationship is also known as composition (or) aggregation. There is no specific keyword to implement HAS-A relationship but mostly new operator is used. The main advantage of HAS-A relationship is reusability.

Example

```
class Engine
{
        //engine specific functionality
}
class Car
{
```

```
    Engine e=new Engine();
}
```

Here Class Car HAS-A engine reference. HAS-A relationship increases dependency between the components and creates maintains problems.

Composition vs Aggregation:

Composition

Without existing container object if there is no chance of existing contained objects then the relationship between container object and contained object is called composition which is a strong association.

Example

University consists of several departments whenever university object destroys automatically all the department objects will be destroyed that is without existing university object there is no chance of existing dependent object hence these are strongly associated and this relationship is called composition.

Aggregation

Without existing container object if there is a chance of existing contained objects such type of relationship is called aggregation. In aggregation objects have weak association.

Example

Within a department there may be a chance of several professors will work whenever we are closing department still there may be a chance of existing professor object without existing department object the relationship between department and professor is called aggregation where the objects having weak association.

Method signature

In java method signature consists of name of the method followed by argument types.

Example

```
    public void methodOne(int i,float f);
```

methodOne(int,float); (signature)

In java return type is not part of the method signature. Compiler will use method signature while resolving method calls. Within the same class we can't take two methods with the same signature otherwise we will get compile time error.

Overloading

Methods having the same name and different argument types is called method overloading. Two methods are said to be overload if and only if both having the same name but different argument types.

Example

```
class Test
{
    public void methadone()
    {
        System.out.println("no-arg method");
    }
    public void methodOne(int i)
    {
        System.out.println( "int-arg method");
    }
    public void methodOne(double d)
    {
        System.out.println("double-arg method");
    }
    public static void main(String[] args)
    {
```

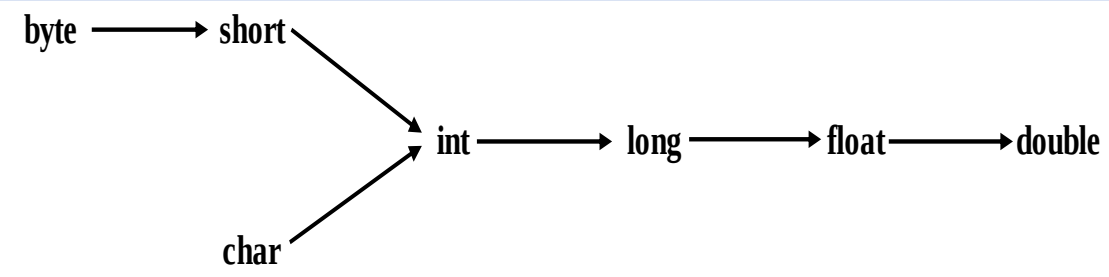
```
        Test t=new Test();
        t.methodOne();                  //no-arg method
        t.methodOne(l0);                //int-arg method
        t. methadone( 10.5) ;           //doubl e-arg method
    }

}
```

In overloading compiler is responsible to perform method resolution(decision) based on the reference type. Hence overloading is also considered as compile time polymorphism(or) static ()early biding.

Case 1: Automatic promotion in overloading.

- In overloading if compiler is unable to find the method with exact match, we won't get any compile time error immediately.
- First compiler promotes the argument to the next level and checks whether the matched method is available or not if it is available then that method will be considered if it is not available then compiler promotes the argument once again to the next level. This process will be continued until all possible promotions. Still if the matched method is not available then we will get compile time error. This process is called automatic promotion in overloading.
- The following are various possible automatic promotions in overloading.




```
class Test
{
    public void methodOne(int i)
    {
        System.out.println("int-arg method'1);
    }
    public void methodOne(float f)
    {
        System.out.println(“float-arg method");
    }
    public static void main(String[] args)
    {
        Test t=new Test();
        //t.methodOne('a');             //int-arg method
        //t.methodOne(l0l);             //float-arg method
        t.methodOne(l0.5);              //C.E:cannot find symbol
    }
}
```

case 2:

```
class Test
{
    public void methodOne(String s)
    {
        System.out.println("String version");
    }
    public void methodOne{Object o)
```

```
	{
		System.out.println("Object version");
	}
	public static void main(String[] args)
	{
		Test t=new Test();
		t.methodOne("bhaskar");		//String version
		t.methodOne(new Object());	//Object version
		t.methodOne(null);//String version
	}
}
```

- In overloading Child will always get high priority then Parent.

```
class Test
{
	public void methodOne(int i)
	{
		System.out.println("general method");
	}
	public void methodOne(int ... i)
	{
		System.out.println("var-arg method");
	}
	public static void main(String[] args)
	{
		Test t=new Test();
		t.methodOne();			//var-arg method
		t.methodOne(l0,20);		//var-arg method
		t.methodOne(l0);		//general method
	}
}
```

In general, var-arg method will get less priority that is if no other method matched then only var-arg method will get chance for execution it is almost same as default case inside switch.

```
class Animal{}
class Monkey extends Animal{}
class Test
{
	public void methodOne(Animal a)
	{
		System.out.println("Animal version");
	}
	public void methodOne(Monkey m)
	{
		System.out.println("Monkey version");
	}
	public static void main(String[] args)
	{
		Test t=new Test();
		Animal a=new Animal();
		t.methodOne(a);				// Animal version
		Monkey m=new Monkey();
		t.methodOne(m);				//Monkey version
```

```
                Animal al=new Monkey();
                t.methodOne(al);                                    // Animal version
        }
}
```

In overloading method resolution is always based on reference type and runtime object won't play any role in overloading.

Overriding

Whatever the Parent has by default available to the Child through inheritance, if the Child is not satisfied with Parent class method implementation then Child is allowed to redefine Parent class method in its own way this process is called overriding. The Parent class method which is overridden is called overridden method. The Child class method which is overriding is called overriding method.

Example

```
class Parent
{
        public void property()
        {
                System.out.println("cash+land+gold"};
        }
        public void marry()
        {
                System.out.println("lakshmi");                          //overridden method
        }
}
class Child extends Parent
{
        public void marry()
        {
                System.out.println("Trisha/tara/anushka");              //overriding method
        }
}
class Test
{
        public static void main(String[] args)
        {
                Parent p=new Parent ();
                p.marry();                          //lakshmi(parent method)
                Child c=new Child ();
                c.marry();                          //Trisha/tara/anushka(child method)
                Parent pl=new Child ();
                pl.marry();                         //Trisha/tara/anushka(child method)
        }
}
```

In overriding method resolution is always takes care by JVM based on runtime object hence overriding is also considered as runtime polymorphism or dynamic polymorphism or late binding. The process of overriding method resolution is also known as dynamic method dispatch

Rules for overriding

- In overriding method names and arguments must be same. That is method signature must be same.

- Until 1.4 version the return types must be same but from 1.5 version onwards co-variant return types are allowed.
- According to this Child class method return type need not be same as Parent class method return type its Child types also allowed.

Example

```
class Parent
{
        public Object methodOne{)
        {
                return null;
        }
}
class Child extends Parent
{
        public String methodOne{)
        {
                return null;
                }
}
```

- It is valid in "1.5" but invalid in "1.4".
- Co-variant return type concept is applicable only for object types but not for primitives.
- Private methods are not visible in the Child classes hence overriding concept is not applicable for private methods.
- Based on own requirement we can declare the same Parent class private method in child class also. It is valid but not overriding.
- Parent class final methods we can't override in the Child class.

Example

```
class Parent
{
        public final void methodOne()
        {
        }
}
class Child extends Parent
{
        public void methodOne()
        {
        }
}
```

Output

Compile time error.
Child.java:8: methodOne() in Child cannot override methodOne() in Parent; overridden method is final

- Parent class non final methods we can override as final in child class. We can override native methods in the child classes.
- We should override Parent class abstract methods in Child classes to provide implementation.

Example

```
abstract class Parent
{
        public abstract void methodOne();
}
class Child extends Parent
{
```

```
        public void methodOne()
        {
        }
}
```

- We can override Parent class non abstract method as abstract to stop availability of Parent class method implementation to the Child classes.

```
class Parent
{
        public void methadone ();
        {
        }
}
abstract class Child extends Parent
{
        public abstract void methadone ();
}
```

- We can override Parent class non abstract method as abstract to stop availability of Parent class method implementation to the Child classes.

Example

```
class Parent
{
        public void methadone ()
        {
        }
}
abstract class Child extends Parent
{
        public abstract void methadone ();
}
```

- While overriding we can't reduce the scope of access modifier.

Example

```
class Parent
{
        public void methodOne()
        {
        }
}
class Child extends Parent
{
        protected void methodOne()
        {
        }
}
```

Output

Compile time error
methodOne() in Child cannot override methodOne() in Parent; attempting to assign weaker access privileges; was public

Checked Vs Un-Checked Exceptions

- The exceptions which are checked by the compiler for smooth execution of the program at runtime are called checked exceptions.
- The exceptions which are not checked by the compiler are called un-checked exceptions.

- Runtime Exception and its child classes, Error and its child classes are unchecked except these the remaining are checked exceptions.
- While overriding if the child class method throws any checked exception compulsory the parent class method should throw the same checked exception or its parent otherwise we will get compile time error. But there are no restrictions for un-checked exceptions.

Overriding with respect to static methods

Case 1

- We can't override a static method as non static

Example

```
class Parent
{
        public static void methodOne()          //here static methodOne() method is a class level
        {
        }
}
class Child extends Parent
{
public void methodOne()                         //here methodOne() method Is a object level hence we can't
{                                                override methodOne() method

}
```

Case 2

- Similarly we can't override a non static method as static.

Case 3

```
class Parent
{
        public static void methodOne()
        {
        }
}
class Child extends Parent
{
        public static void methodOne(}
        {
        }
}
```

- It is valid. It seems to be overriding concept is applicable for static methods but it is not overriding it is method hiding.

Method Hiding

All rules of method hiding are exactly same as overriding except the following differences.

Overriding	Method hiding
1. Both Parent and Child class methods should be non static.	1. Both Parent and Child class methods should be static
2. Method resolution is always takes care by JVM based on runtime object	2. Method resolution is always takes care by compiler based on reference type.
3. Overriding is also considered as runtime polymorphism (or) dynamic polymorphism (or) late binding.	3. Method hiding is also considered as compile time polymorphism (or) static polymorphism (or) early biding.

Example

```
class Parent
{
        public static void methodOne()
        {
                System.out.println("parent class"};
        }
}
class Child extends Parent
{
        public static void methodOne()
        {
        }
}
class Test
{
        System.out.println("child class"};
        public static void main(String[] args)
        {
                Parent p=new Parent();
                p.methodOne();                  //parent class
                Child c=new Child();
                c.methodOne();                  //child class
                Parent pl=new Child();
                pl.methodOne();                 //parent class
        }
}
```

If both Parent and Child class methods are non static then it will become overriding and method resolution is based on runtime object. In this case the output is
Parent class
Child class
Child class

Overriding with respect to Var arg methods:

A var arg method should be overridden with var-arg method only. If we are trying to override with normal method then it will become overloading but not overriding.

Example

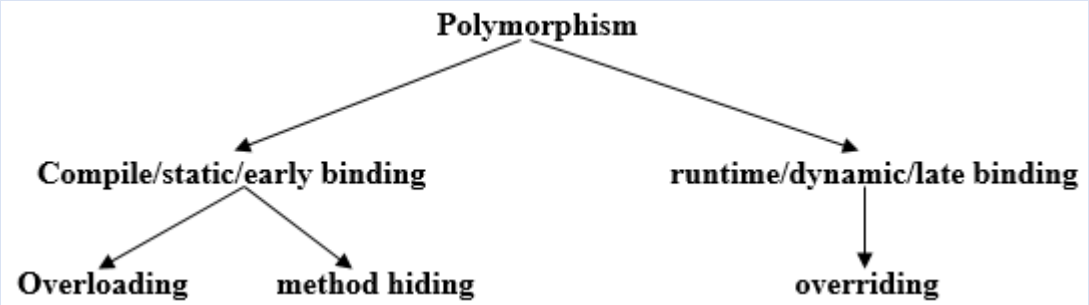
```
class Parent
{
        public void methodOne(int ... i)
        {
                System.out.println("parent class");
        }
}
class Child extends Parent
{
        public void methodOne(int i)
        {
                System.out.println("child class");
        }
class Test
```

```
{
        public static void main(String[] args)
        {
                Parent p=new Parent();
                p.methodOne(l0);                //parent class
                Child c=new Child();
                c.methodOne(l0);                //child class
                Parent pl=new Child();
                pl.methodOne(10);               //parent class
        }
}
```

In the above program if we replace child class method with var arg then it will become overriding. In this case the output is
Parent class
Child class
Child class

Important Points

- Inheritance talks about reusability
- Polymorphism talks about flexibility
- Encapsulation talks about security.



Constructors

- Object creation is not enough compulsory we should perform initialization then only the object is in a position to provide the response properly.
- Whenever we are creating an object some piece of the code will be executed automatically to perform initialization of an object this piece of the code is nothing but constructor. Therefore, the main objective of constructor is to perform initialization of an object.

Example

```
class Student
{
        String name;
        int rollno;
        Student (String name,int rollno)
        {
                this. name=name;
                this. roll no=roll no;
        }
        public static void main(String[] args)
        {
                Student sl=new Student("vijayabhaskar",101);
                Student s2=new Student("bhaskar",102);
        }
```

```
}
```

Constructor Vs instance block

- Both instance block and constructor will be executed automatically for every object creation but instance block first followed by constructor.
- The main objective of constructor is to perform initialization of an object.
- Other than initialization if we want to perform any activity for every object creation, we have to define that activity inside instance block.
- Both concepts having different purposes hence replacing one concept with another concept is not possible.
- Constructor can take arguments but instance block can't take any arguments hence we can't replace constructor concept with instance block.
- Similarly, we can't replace instance block purpose with constructor.

Demo program to track no of objects created for a class

```
class Test
{
        static int count=0;
        {
                count++;                        // Instance block
        }
        Test()
        {}
        Test(int i)
        {}
        public static void main(String[] args)
        {
                        Test tl=new Test();
                        Test t2=new Test(l0);
                        Test t3=new Test();
                        System.out.println(count);//3
        }
}
```

Rules to write constructors

1. Name of the constructor and name of the class must be same.
2. Return type concept is not applicable for constructor even void also by mistake if we are declaring the return type for the constructor, we won't get any compile time error and runtime error compiler simply treats it as a method.

Example

```
class Test
{
        void Test() it is not a constructor and it is a method
        {
        }

}
```

3. It is legal (but stupid) to have a method whose name is exactly same as class name.
4. The only applicable modifiers for the constructors are public, default, private, protected.
5. If we are using any other modifier we will get compile time error.
6. Constructor can not be declared static.

Default constructor

- For every class in java including abstract classes also constructor concept is applicable.
- If we are not writing at least one constructor then compiler will generate default constructor.

- If we are writing at least one constructor then compiler won't generate any default constructor. Hence every class contains either compiler generated constructor (or) programmer written constructor but not both simultaneously.

Prototype of default constructor

- It is always no argument constructor.
- The access modifier of the default constructor is same as class modifier. (This rule is applicable only for public and default).
- Default constructor contains only one line. super(); it is a no argument call to super class constructor.

Programmers code	Compiler generated code

class Test
{
 	
}
	class Test
	{
	Test()
	{
	super();
	}
	}

public class Test
{
 	
}
	public class Test
	{
	public Test()
	{
	super();
	}
	}

class Test
{
void Test()
{
 	
}
}
	class Test
	{
	Test()
	{
	super();
	}
	void Test()
	{
	}
	}

class Test
{
Test(int i)
{
 	
}
}
	class Test
	{
	Test(int i)
	{
	super();
	}
	}

class Test
{
Test()
{
super();
}
}
	class Test
	{
	Test()
	{
	super();
	}
	}

class Test {	class Test {

```
    Test(int i)
    {
        this();
    }
    Test()
    {

    }
}
```

```
    Test(int i)
    {
        this();
    }
    Test()
    {
        super();
    }
}
```

super() vs this()

The 1st line inside every constructor should be either super() or this() if we are not writing anything compiler will always generate super().

Case 1

We have to take super() (or) this() only in the 1st line of constructor. If we are taking anywhere else we will get compile time error.

Example

```
class Test
{
    Test()
    {
        System.out.println("constructor");
        super();
    }
}
```

Output:

Compile time error.

Call to super must be first statement in constructor

Case 2

We can use either super() (or) this() but not both simultaneously.

Example

```
class Test
{
    Test()
    {
        super();
        this();
    }
}
```

Output

Compile time error.

Call to this must be first statement in constructor

Case 3

We can use super() (or) this() only inside constructor. If we are using anywhere else we will get compile time error.

Example

```
class Test
{
    public void methadone()
    {
```

```
        super();
    }
}
```

Output

Compile time error.

Call to super must be first statement in constructor

That is we can call a constructor directly from another constructor only.

super(),this()	super, this
These are constructors calls.	These are keywords which can be used to call parent class and current class instance members.
We should use only inside constructors.	We can use anywhere except static area.

Overloaded constructors

A class can contain more than one constructor and all these constructors having the same name but different arguments and hence these constructors are considered as overloaded constructors.

Example:

```
class Test
{
    Test(double d)
    {
        this(10);
        System.out.println("double-argument constructor”);
    }
    Test(int i)
    {
        this();
        System.out.println( “int-argument constructor");
    }
    Test()
    {
        System.out.println("no-argument constructor");
    }
    public static void main(String[] args)
    {
        Test tl=new Test(l0.5);          //no-argument constructor/int-argument
        constructor/double-argument constructor
        Test t2=new Test(10);            //no-argument constructor/int-argument constructor
        Test t3=new Test();              //no-argument constructor
    }
}
```

- Inheritance concept is not applicable for constructors and hence overriding concept also not applicable to the constructors. But constructors can be overloaded.
- We can take constructor in any java class including abstract class also but we can't take constructor inside inheritance.

```
class Test
{
    Test()
    {
    }
}
```

```
abstract class Test
{
    Test()
    {
    }
}
```

```
I    nterface Testl
{
        Testl()
        {
        }
}
```

valid valid invalid

- Abstract class constructor will be executed to perform initialization of child class object.

Static block

- Static blocks will be executed at the time of class loading hence if we want to perform any activity at the time of class loading, we have to define that activity inside static block.
- Within a class we can take any no. Of static blocks and all these static blocks will be executed from top to bottom.

Inner Classes

Sometimes we can declare a class inside another class such type of classes are called inner classes. The relationship between outer class and inner class is not IS-A relationship and it is Has-A relationship.

Example

```
class University                    //Outer class
{
        class Department            //inner class
        {
        }
}
```

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

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

Normal (or) 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

```
class Outer
{
        class Inner
        {
        }
}
```

In case of above example if compile this code we will have two .class file like:

javac Outer.java

Outer.class
Outer$1nner.class

Example

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

```
	}
}
```

Output

javac Outer .java
--Outer.class
--OuterSlnner .class
E:\scjp>java Outer
outer class main method
E:\scjp>java Outer$1nner
Exception in thread "main" java.lang.NoSuchMethodError: main
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 commend prompt.

Example

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

Output

inner classes cannot have static declarations
public static void main(String[] args)

Accessing inner class code from static area of outer class

Example

```
class Outer
{
	class Inner
	{
		public void methodOne(){
		System.out.println("inner class method");
	}
	public static void main(String[] args)
	{
		Outer o=new Outer();
		Outer.Inner i=o.new Inner();
		i.methodOne();
			//or
		Outer.Inner i=new Outer().new lnner()
		i.methodOne();
			//or
		new Outer().new lnner().methodOne();
	}
}
```

Accessing inner class code from instance area of outer class

Example

```
class Outer
```

```
{
        class Inner
        {
                public void methodOne()
                {
                        System.out.println("inner class method");
                }
        }
        public void methodTwo()
        {
                Inner i=new Inner();
                i.methodOne();
        }
        public static void main(String[l args}
        {
                Outer o=new Outer();
                o.methodTwo();
        }
}
```

Output

```
javac Outer.java
java Outer
Inner class method
```

Accessing inner class code from outside of outer class:

Example

```
class Outer
{
        class Inner
        {
                public void methodOne()
                {
                        System.out.println("inner class method"};
                }
        }
}
class Test
{
        public static void main(String[] args)
        {
                new Outer().new lnner().methodOne();
        }
}
```

Output

```
Inner class method
```

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=l0;
```

```
	static int y=20;
	class Inner
	{
		public void methodOne()
		{
			System.out.println(x);//10
			System.out.println(y);//20
		}
	}
	public static void main(String□ args)
	{
		new Outer().new lnner().methodOne();
	}
}
```

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".

Example

```
class Outer
{
	int x=l0;
	class Inner
	{
		intx=l00;
		public void methodOne()
		{
			intx=l000;
			System.out.println(x);//1000
			System.out.println(this.x);//100
			System.out.println(Outer.this.x);//10
		}
	}
	public static void main(String[] args)
	{
		new Outer().new lnner().methodOne();
	 }
}
```

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.
- 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.

Example

```
class Test
{
	public void methodOne()
	{
		class Inner
		{
```

```
                public void sum(int i,int j)
                {
                        System.out.println("The sum:"+(i+j));
                }
        }
        Inner i=new Inner();
        i.sum(l0,20);
        i.sum(l00,200);
        i.sum(l000,2000);
    }
    public static void main(String[] args)
    {
        new Test().methodOne();
    }
}
```

Output

The sum: 30
The sum: 300
The sum: 3000

- 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.

Example

```
class Test
{
    int x=l0;
    static int y=20;
    public void methodOne()
    {
        class Inner
        {
            public void methodTwo()
            {
                System.out.println(x);          //10
                System.out.println(y);          //20
            }
        }
    }
    Inner i=new Inner();
    i.methodTwo();
    }
    public static void main(String[] args)
    {
        new Test().methodOne();
    }
}
```

- 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.

Example

```
class Test
{
	int x=10;
	public void methodOne()
	{
		int y=20;
		class Inner
		{
			public void methodTwo()
			{
				System.out.println(x);		//10
				System.out.println(y);		//C.E: local variable y is accessed from
									within inner class; needs to be declared final.
			}
		}
		Inner i=new Inner();
		i.methodTwo();
	}
	public static void main(String[J args)
	{
		new Test().methodOne();
	}
}
```

- If we declared y as final then we won't get any compile time error.
- The only applicable modifiers for method local inner classes are:

Final
Abstract
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.

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 pl=new Popcorn();
        pl.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.

3. Popcorn p=new Popcorn()

```
{
        public void taste()
        {
                System.out.println("salty" );
        }
};
```

- We are creating child class for Popcorn without name.
- We are overriding taste() method.
- We are creating object for that child class with parent reference

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.

Example

```
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 pl=new Popcorn(};
pl.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<l0;i++)
                                {
                                        System.out.println("child thread"};
                                }

                        }
                };
                t.start();
                for(int i=0;i<l0;i++)
                {
                        System.out.println("main thread");
                }
        }
}
```