Inheritance

Inheritance is a mechanism in which one class acquires the property of another class.

or

The concept of occurring one class features in to another class is called Inheritance.
● We can implement Inheritance in Java by using extends keyword.
● It uses IS-A relationship (parent-child relationship)

Types of Inheritance

1. Single Inheritance
2. Multilevel Inheritance
3. Hierarchical Inheritance
4. Multiple Inheritance
5. Hybrid Inheritance

Syntax:

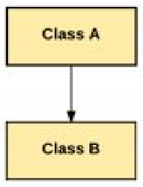
```
class Super {
  .....
  .....
}
class Sub extends Super {
  .....
  .....
}
```

Terms used in Inheritance:

● Class: A class is a group of objects which have common properties.
● Sub_class/Child_Class/derived class/extended class: Subclass is a class which inherits the other class.
● Super class/Parent class/Base class: Superclass is the class from where a subclass inherits the features.
● Reusability: As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class.

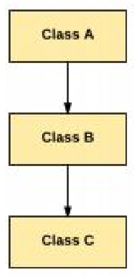
Single Inheritance:

In Single Inheritance one class extends another class (one class only).



Syntax:

```
class Super {
  .....
  .....
}
class Sub extends Super {
  .....
  .....
}
```



//Single Inheritance example program

```
class A
{
  public void methodA()
  {
   System.out.println("Base class method");
  }
}

class B extends A
{
  public void methodB()
  {
   System.out.println("Child class method");
  }
  public static void main(String args[])
  {
   B obj = new B();
   obj.methodA(); //calling super class method
   obj.methodB(); //calling local method
  }
}
```

OUTPUT:

```
Base class method

Child class method
```

Multilevel Inheritance:

When there is a chain of inheritance, it is known as *multilevel inheritance*. one class can inherit from a derived class. Hence, the derived class becomes the base class for the new class.

Syntax:

```
class A

{
}
class B extends A

{
}
class C extends B

{
}
```

//Multilevel Inheritance example program

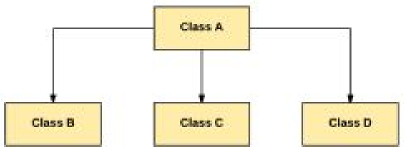
```
class X
{
  public void methodX()
  {
    System.out.println("Class X method");
  }
}
class Y extends X
{
public void methodY()
{
System.out.println("class Y method");
}
}
Class Z extends Y
{
  public void methodZ()
  {
    System.out.println("class Z method");
  }
  public static void main(String args[])
  {
    Z obj = new Z();
    obj.methodX();              //calling grand parent class method
    obj.methodY();              //calling parent class method
    obj.methodZ();  //calling local method
  }
}
```

OUTPUT:

```
Class X method

class Y method

class Z method
```

Hierarchical Inheritance:

In Hierarchical Inheritance, one class is inherited by many sub classes.



Syntax:

```
class A

{
-------
}
class B extends A

{
--------

}
class C extends A

{
---------

}
class D extends A

{
--------
}
```

//Hierarchical Inheritance example program

```
class A
{
  public void methodA()
  {
    System.out.println("method of Class A");
  }
}
class B extends A
{
  public void methodB()
  {
    System.out.println("method of Class B");
  }
}
class C extends A
```

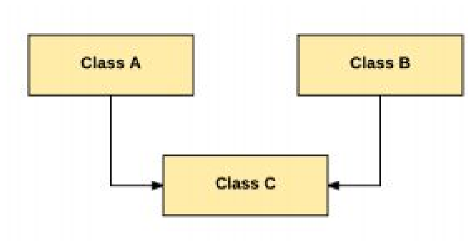
```
{
 public void methodC()
 {
   System.out.println("method of Class C");
 }
}
class D extends A
{
 public void methodD()
 {
   System.out.println("method of Class D");
 }
}
class JavaExample
{
 public static void main(String args[])
 {
   B obj1 = new B();
   C obj2 = new C();
   D obj3 = new D();
   //All classes can access the method of class A
   obj1.methodA();
   obj2.methodA();
   obj3.methodA();
 }
}
```

Output:

```
method of Class A
method of Class A
method of Class A
```

Multiple Inheritance:

"**Multiple Inheritance**" refers to the concept of one class extending (Or inherits) more than one base class. Java does not support multiple inheritance.



Why multiple inheritance not supported in java?

● The reason behind this is to prevent ambiguity. Consider a case where class B extends class A and Class C and both class A and C have the same method show(). Now java compiler cannot decide, which display method it should inherit. To prevent such situation, multiple inheritances is not allowed in java.

//Multiple Inheritance example program

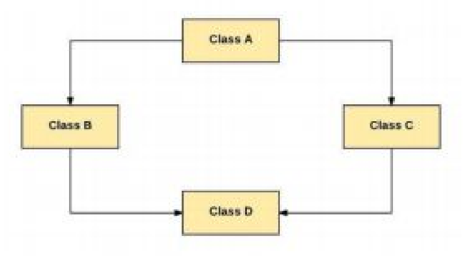
```
    class A
    {
    void msg()
    {
    System.out.println("Hello");
    }
    }
    class B
    {
    void msg()
    {
    System.out.println("Welcome");
    }
  }
    class C extends A,B
    {
    public static void main(String args[])
    {
     C obj=new C();
     obj.msg();          //Now which msg() method would be invoked?
    }
    }
```

<u>OUTPUT:</u>

```
Compile Time Error
```

Hybrid Inheritance:

Hybrid inheritance is one of the inheritance types in Java which is a combination of Single and Multiple inheritance. Since in Java Multiple Inheritance is not supported directly we can achieve Hybrid inheritance also through Interfaces only.



Member Access Rules:
● The '.' operator is also known as member operator it is used to access the member of a package or a class.
Rules:
● If superclass is public then members can be accessed to subclass.
● If superclass is private then members can't be accessed to subclass.
● If superclass is default then members can be accessed to subclass.
● If superclass is protected then members can be accessed to subclass.
● If superclass is default then members can be accessed to subclass.

super uses

The super keyword in Java is used in subclasses to access superclass members (attributes, constructors and methods).

The most common use of the super keyword is to eliminate the confusion between superclasses and subclasses that have methods with the same name.

Uses of super keyword:

1. To call methods of the superclass that is overridden in the subclass.
2. To access attributes (fields) of the superclass if both superclass and subclass have attributes with the same name.
3. To explicitly call superclass no-arg (default) or parameterized constructor from the subclass constructor.

- By using super keyword we can refer to super class members.
```
super.variable_name;
super.method_name();
```
- By using super keyword we can refer to super class constructors.
```
        super()
super(parameters).
```

// super keyword to access the variables of parent class

```
class Superclass
{
  int num = 100;
}
class Subclass extends Superclass
{
  int num = 110;
  void printNumber()
{

            System.out.println(super.num);
  }
  public static void main(String args[])
{
            Subclass obj= new Subclass();
            obj.printNumber();
  }
}
```

Output:

```
100
```

// super keyword to invoke constructor of parent class

```
class Parentclass
{
  Parentclass()
{
            System.out.println("no-arg constructor of parent class");
  }
  Parentclass(String str)
{
            System.out.println("parameterized constructor of parent class");
  }
}
class Subclass extends Parentclass
{
  Subclass()
{

            super("abc");
            System.out.println("Constructor of child class");

  }
  void display(){
```

```
            System.out.println("Hello");
  }
  public static void main(String args[]){
            Subclass obj= new Subclass();
            obj.display();
  }
}
```

OUTPUT:

```
parameterized constructor of parent class
Constructor of child class
Hello
```

//super keyword in case of method overriding

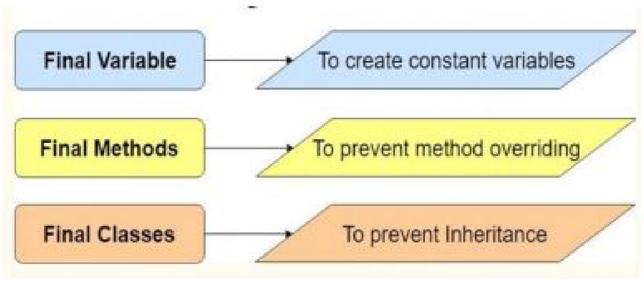
```
class Parentclass
{

  void display()
{
            System.out.println("Parent class method");
  }
}
class Subclass extends Parentclass
{

  void display()
{
            System.out.println("Child class method");
  }
  void printMsg()
{

            display();                              //This would call Overriding method

            super.display();     //This would call Overridden method
  }
  public static void main(String args[])
{
            Subclass obj= new Subclass();
            obj.printMsg();
  }
}
```

OUTPUT:

```
Child class method
Parent class method
```

Final classes and methods

The final keyword in java is used to restrict the user. Final can be:
○ variable
○ method
○ class



Final Variable

Final variable is a constant. We cannot change the value of final variable after initialization.

//Sample program for final keyword in Java

```
class FinalVarDemo
{
   final int a = 10;
   void show()
   {
       a = 20;
      System.out.println("a : "+a);
   }
   public static void main(String args[])
   {
      FinalVarDemo var = new FinalVarDemo();
      var.show();
   }
}
```

Output:
Compile time error

Final method

When we declare any method as final, it cannot be override.

//Example: Sample program for final method in Java

```
class XYZ
{
  final void demo()
{
     System.out.println("XYZ Class Method");
  }
}

class ABC extends XYZ
{
  void demo()
{
    System.out.println("ABC Class Method");
  }

  public static void main(String args[])
{
    ABC obj= new ABC();
    obj.demo();
  }
}
```

Output:
Compile time error

Final Class

When a class is declared as a final, it cannot be extended.

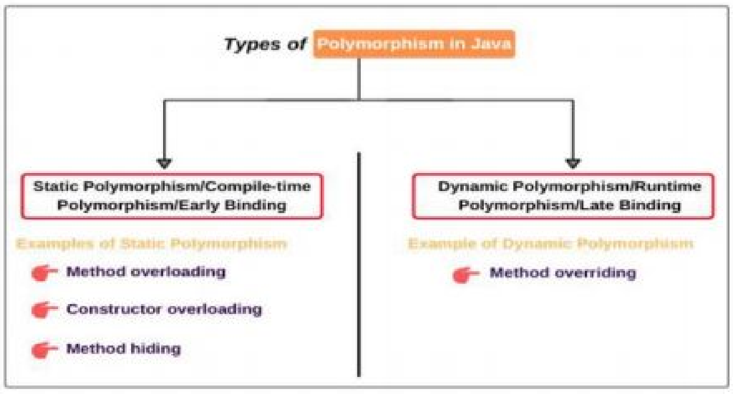
//Example: Sample program for final class in Java

```
final class XYZ
{
}

class ABC extends XYZ
{
  void demo()
{
    System.out.println("My Method");
  }
  public static void main(String args[])
{
    ABC obj= new ABC();
    obj.demo();
  }
}
```

Output:
Compile time error

Polymorphism

Polymorphism means "many forms". Polymorphism is one of the OOPs feature that allows us to perform a single action in different ways.
we use method overloading and method overriding to achieve polymorphism.
Types of polymorphism:
1 . Method Overloading – This is an example of compile time (or static polymorphism).
2. Method Overriding – This is an example of runtime time (or dynamic polymorphism).


Types of Polymorphism in Java
Static Polymorphism/Compile-time Polymorphism/Early Binding
Examples of Static Polymorphism
Method overloading
Constructor overloading
Method hiding
Dynamic Polymorphism/Runtime Polymorphism/Late Binding
Example of Dynamic Polymorphism
Method overriding

Method Overloading :

If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.

//Method Overloading Examples

```
public class Add
{
public int add(int a , int b)
{
    return (a + b);
}
public int add(int a , int b , int c)
{
    return (a + b + c) ;
}
public double add(double a , double b)
{
    return (a + b);
}
public static void main( String args[])
{
Add ob = new Add();
ob.add(15,25);
ob.add(15,25,35);
ob.add(11.5 , 22.5);
}
}
```

OUTPUT:

```
        40
         75
        34
```

Method Overriding:

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding.**

```
//Java Program to illustrate the use of Java Method Overriding

class Vehicle{

void run(){System.out.println("Vehicle is running");}

}

 class Bike2 extends Vehicle

{

 void run()

{

System.out.println("Bike is running safely");}

 public static void main(String args[]){

 Bike2 obj = new Bike2();//creating object

 obj.run();//calling method

 }

}
```

OUTPUT:

```
        Bike is running safely
```

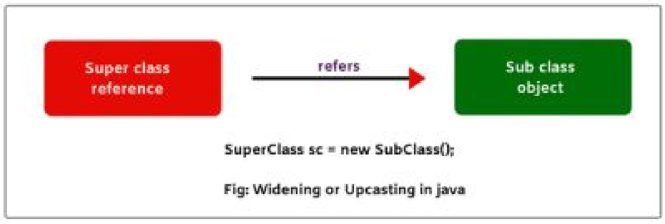
Runtime Polymorphism

Runtime polymorphism or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

Dynamic Method Dispatch

- Runtime polymorphism or Dynamic Method Dispatch is a process in which a call to an overridden method is resolved at runtime rather than compile-time.
- In this process, an overridden method is called through the reference variable of a superclass.
- The determination of the method to be called is based on the object being referred to by the reference variable.

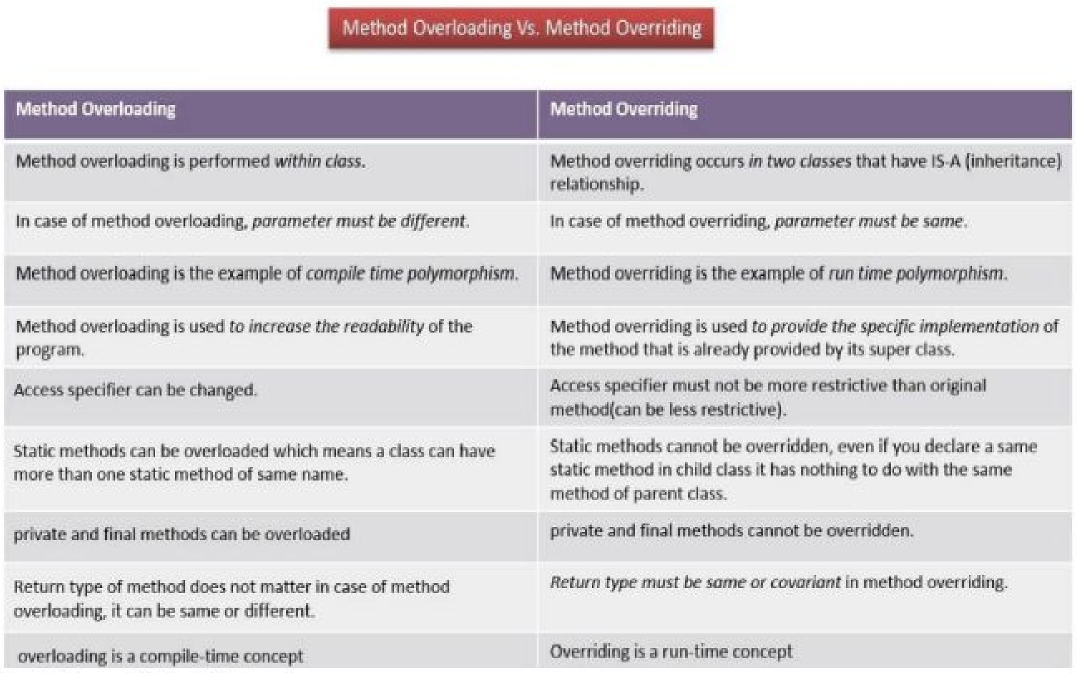
When the reference variable of super class refers to the object of subclass, it is known as widening or **upcasting in java**.


Super class reference
refers
Sub class object
SuperClass sc = new SubClass();
Fig: Widening or Upcasting in java

```
//Example program for upcasting

class Parent
{
void view()
{
System.out.println("this is a parent class method);
}}
class Child extends Parent
{
void view()
{
System.out.println("this is a child class method);
}}
public static void main(String args[])
{
Parent ob = new Parent();
ob.view();
Parent ob1 = new Child();
ob1.view();
```

Output:

```
this is a child class method.
```



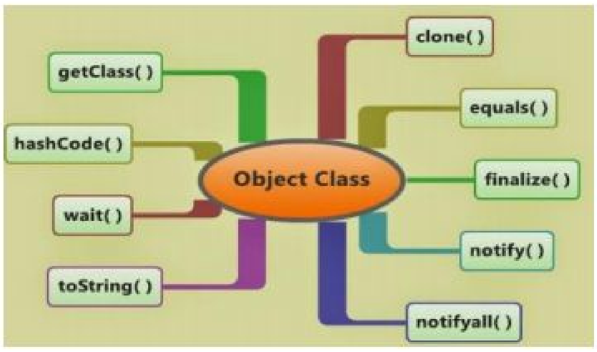
Method Overloading Vs. Method Overriding

Method Overloading	Method Overriding
Method overloading is performed *within class*.	Method overriding occurs *in two classes* that have IS-A (inheritance) relationship.
In case of method overloading, *parameter must be different*.	In case of method overriding, *parameter must be same*.
Method overloading is the example of *compile time polymorphism*.	Method overriding is the example of *run time polymorphism*.
Method overloading is used *to increase the readability* of the program.	Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class.
Access specifier can be changed.	Access specifier must not be more restrictive than original method(can be less restrictive).
Static methods can be overloaded which means a class can have more than one static method of same name.	Static methods cannot be overridden, even if you declare a same static method in child class it has nothing to do with the same method of parent class.
private and final methods can be overloaded	private and final methods cannot be overridden.
Return type of method does not matter in case of method overloading, it can be same or different.	*Return type must be same or covariant* in method overriding.
overloading is a compile-time concept	Overriding is a run-time concept

<u>Object Class & its Methods:</u>

● The Object class is the parent class of all the classes in java by default.

Methods of Object Class:



● getClass: This method gives an object that contains name of class to which an object belongs.

<u>Syntax:</u> public final Class getClass()

● hashCode(): returns the hashcode number of object.

<u>Syntax:</u> public int hashCode()

● equals(): compares the given object to this object.

<u>Syntax</u>: public boolean equals(Object obj)

● clone(): creates and returns the exact copy (clone) of this object.

<u>Syntax:</u> protected Object clone() throws CloneNotSupportedException

● toString: returns the string representation of this object.
Syntax: public String toString()

● notify(): wakes up single thread, waiting on this object's monitor.
Syntax: public final void notify()

● notifyAll(): wakes up all the threads, waiting on this object's monitor.
Syntax: public final void notifyAll()

(wait() in different formates)
● wait(): causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method).
Syntax: public final void wait(long timeout)throws InterruptedException

● wait():causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method).
Syntax: public final void wait(long timeout,int nanos)throws InterruptedException

● wait(): causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method).
Syntax: public final void wait()throws InterruptedException

● finalize(): It is invoked by the garbage collector before object is being garbage collected or removed from memory.
Syntax: protected void finalize()throws Throwable.

Abstract

Abstraction is a process of hiding the implementation details and showing only functionality to the user.
By using abstract classes and interfaces we can implement abstraction.

- It can have abstract and non-abstract methods (method with the body).
- It can't be instantiated.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the
  method.

The abstract keyword is a non-access modifier, used for classes and methods:

- **Abstract class:** is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
- **Abstract method:** can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

Syntax:
```
abstract class <classname>
  {
        public abstract void abstractMethod();
        public void normalMethod()
        {
             //method body
        }
   }
```

Ways to achieve Abstraction:

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

//Example program for abstraction

```
    abstract class Car
    {
       abstract void accelerate();
    }
    class Suzuki extends Car
    {
           void accelerate()
        {
              System.out.println("Suzuki::accelerate");

       }
        }
    class Main{
       public static void main(String args[])
    {
          Car obj = new Suzuki();    //Car object =>contents of Suzuki
          obj.accelerate();          //call the method
            }
        }
```

Output:

```
Suzuki::accelerate
```