Polymorphisam:

Polymorphism is one of the oops principle.

It is one object oriented programming system principle using by the java for doing different operation with the, help of same method either within the same class or different class.

It is a mechanism to do different actions within the different forms.

Polymorphism terminology comes from Greek language.

Poly means many and morphs means forms.

Polymorphism means many forms.

The method will exists in different formats for doing different actions.

Polymorphism can classified into two types

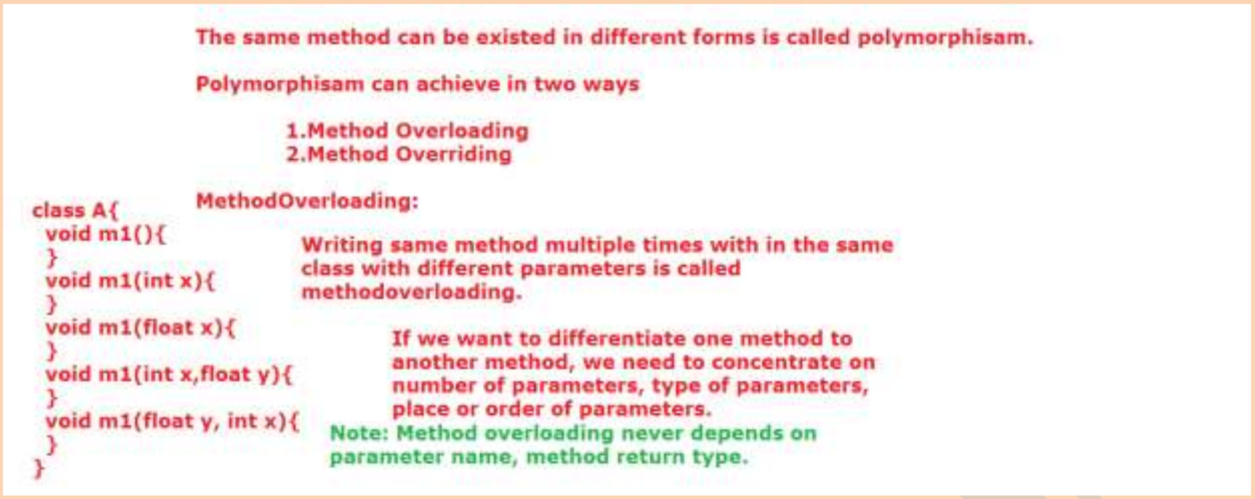
a. Static polymorphism or Compile time polymorphism.

b. Dynamic polymorphism or Runtime polymorphism.

Polymorphism can be achieved through two techniques.

a. method overloading.

b. method overriding.


The same method can be existed in different forms is called polymorphisam.

Polymorphisam can achieve in two ways

1.Method Overloading
2.Method Overriding

MethodOverloading:

Writing same method multiple times with in the same class with different parameters is called methodoverloading.

If we want to differentiate one method to another method, we need to concentrate on number of parameters, type of parameters, place or order of parameters.

Note: Method overloading never depends on parameter name, method return type.

```
class A{
 void m1(){
 }
 void m1(int x){
 }
 void m1(float x){
 }
 void m1(int x,float y){
 }
 void m1(float y, int x){
 }
}
```

Method Overloading:

Writing the same method with different parameters (Number of, type of, place of) with in the same class is called method overloading.

We cannot write same method with same parameters within the same class.

We should be differentiate one method to another method with the help of parameters.

Note: Don't consider variables names and return types.

1. Number of parameters.

2. Type of parameters.

3. Place of parameters.

Invalid syntax:

```
class A{
	void m1(){
	}
	void m1(){//duplicate methods
	}
}
```

Valid syntax:

```
class A{
        void m1(){
        }
        void m1(int x){
        }
        void m1(String x){
        }
        void m1(int x, String y){
        }
        void m1(String y, int x){
        }
}
```

Example program on Method Overloading:

```
public class MethodOverloadingDemo {
        void m1(){
                System.out.println("m1 method with zero argument");
        }
        void m1(int x){
                System.out.println("m1 method with int argument");
                System.out.println(x);
        }
        void m1(String x){
                System.out.println("m1 method with string argument");
                System.out.println(x);
```

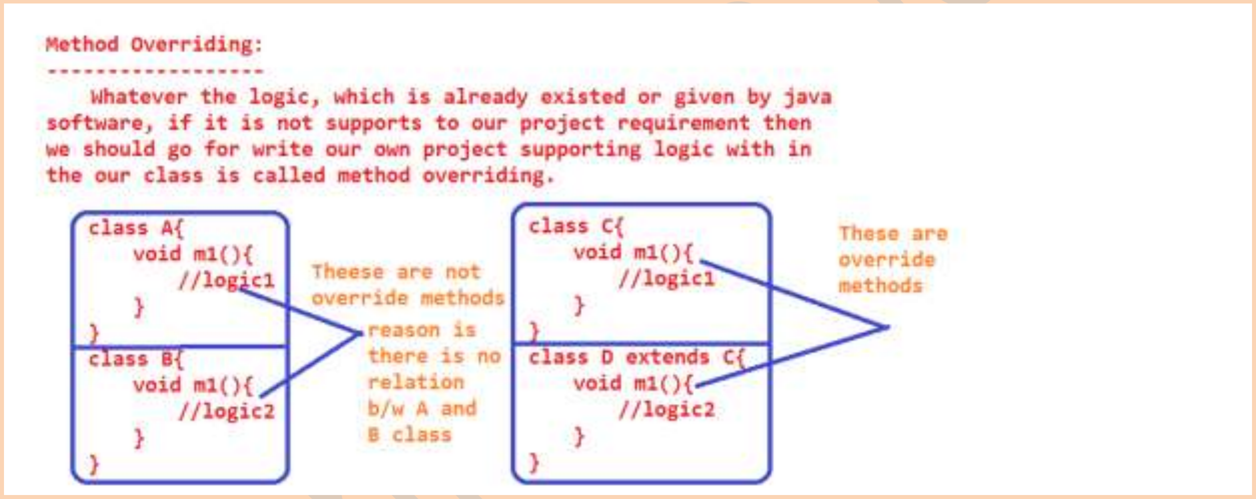
```
	}
	void m1(int x, float y){
		System.out.println("m1 method int-float argument");
		System.out.println(x);
		System.out.println(y);
	}
	void m1(float y, int x){
		System.out.println("m1 method float-int argument");
		System.out.println(y);
		System.out.println(x);
	}
	public static void main(String[] args) {
		MethodOverloadingDemo md = new  MethodOverloadingDemo();
		md.m1();
		md.m1(234);
		md.m1("suji");
		//md.m1(10,20);
		md.m1(10,23.34f);
		md.m1(23.33f,10);
	}
}
```

```
public class MethodOverloading {
	static void m1(int x){
		System.out.println("int-m1");
	}
	static void m1(String o){
		System.out.println("String-m1");
```

```
        }
        static void m1(Integer o){
                System.out.println("Integer-m1");
        }
        public static void main(String[] args) {
                m1(100);
                int a1 = 100;
                Integer a2 = 100;
                m1(new Integer(100));
        }
}
```

Method Overriding:

Whatever the logic, which is given by the java software or existed method is not suitable for our project requirement, then we should write our own logic within the existed method is called Method Overriding.



Method Overriding:

    Whatever the logic, which is already existed or given by java software, if it is not supports to our project requirement then we should go for write our own project supporting logic with in the our class is called method overriding.

```
class A{
    void m1(){
        //logic1
    }
}
class B{
    void m1(){
        //logic2
    }
}
```

```
class C{
    void m1(){
        //logic1
    }
}
class D extends C{
    void m1(){
        //logic2
    }
}
```

Theese are not override methods
reason is there is no relation b/w A and B class
These are override methods

Method Overriding must be follows the following rules:

1) Access Modifier of a method in subclass must be same or increasable.

super class → default method

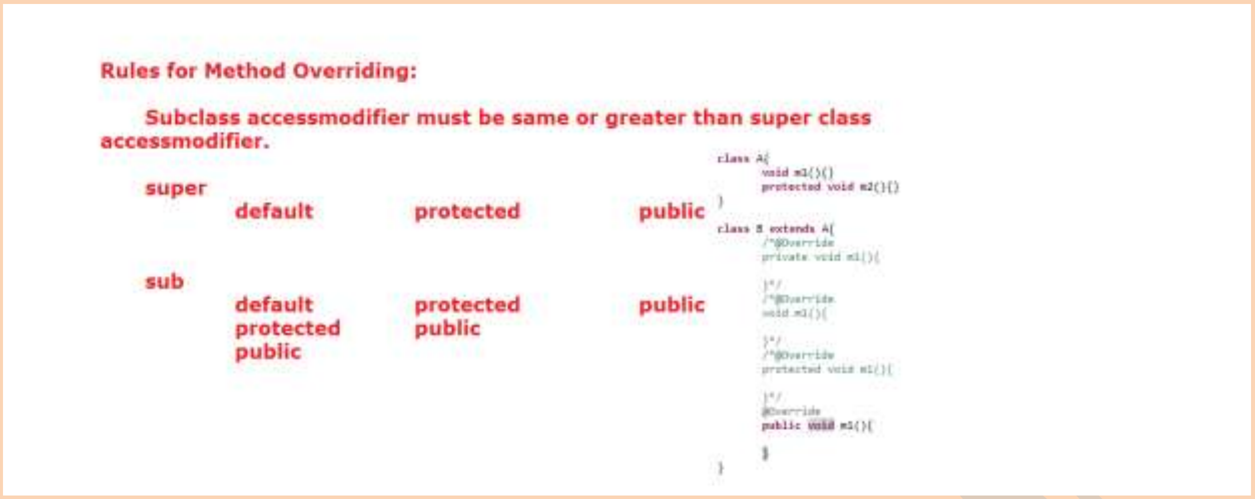
sub class → default, protected, public.

super class → protected method

sub class → protected, public

super class → public method

sub class → public



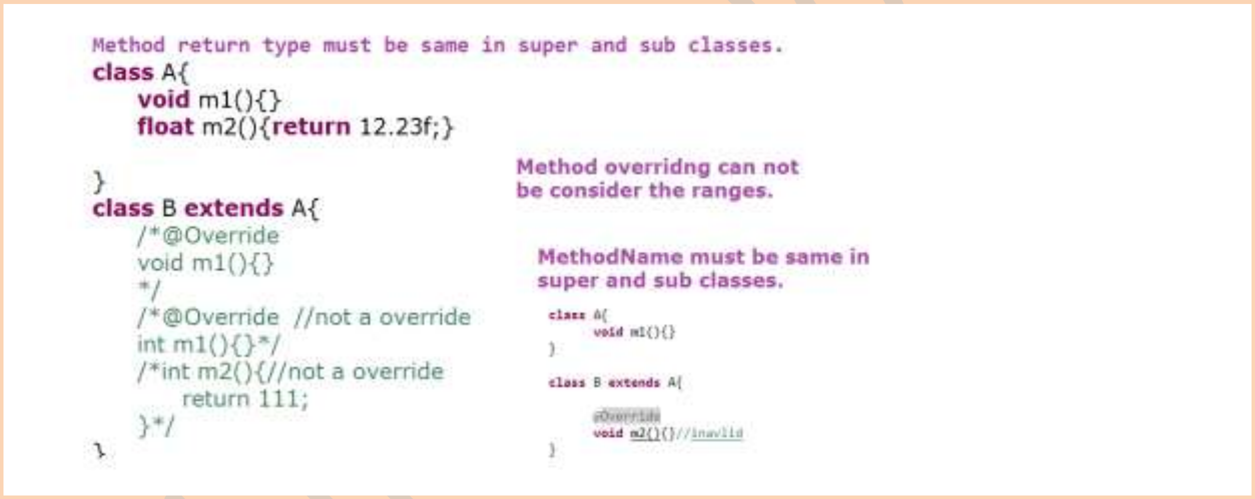
Rules for Method Overriding:

Subclass accessmodifier must be same or greater than super class accessmodifier.

super	default	protected	public
sub	default protected public	protected public	public

```
class A{
        void m1(){}
        protected void m2(){}
}
class B extends A{
        /*@Override
        private void m1(){

        }*/
        /*@Override
        void m1(){

        }*/
        /*@Override
        protected void m1(){

        }*/
        @Override
        public void m1(){

        }
}
```

1) Method return type must be same.



Method return type must be same in super and sub classes.

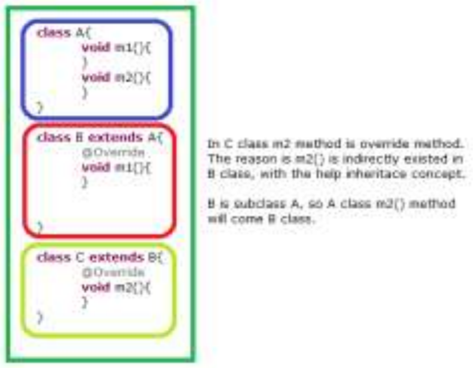
```
class A{
     void m1(){}
     float m2(){return 12.23f;}

}
class B extends A{
     /*@Override
     void m1(){}
     */
     /*@Override  //not a override
     int m1(){}*/
     /*int m2(){//not a override
          return 111;
     }*/
}
```

Method overridng can not be consider the ranges.

MethodName must be same in super and sub classes.

```
class A{
        void m1(){}
}

class B extends A{

        @Override
        void m2(){}//inavlid
}
```

2) Method name must be same.

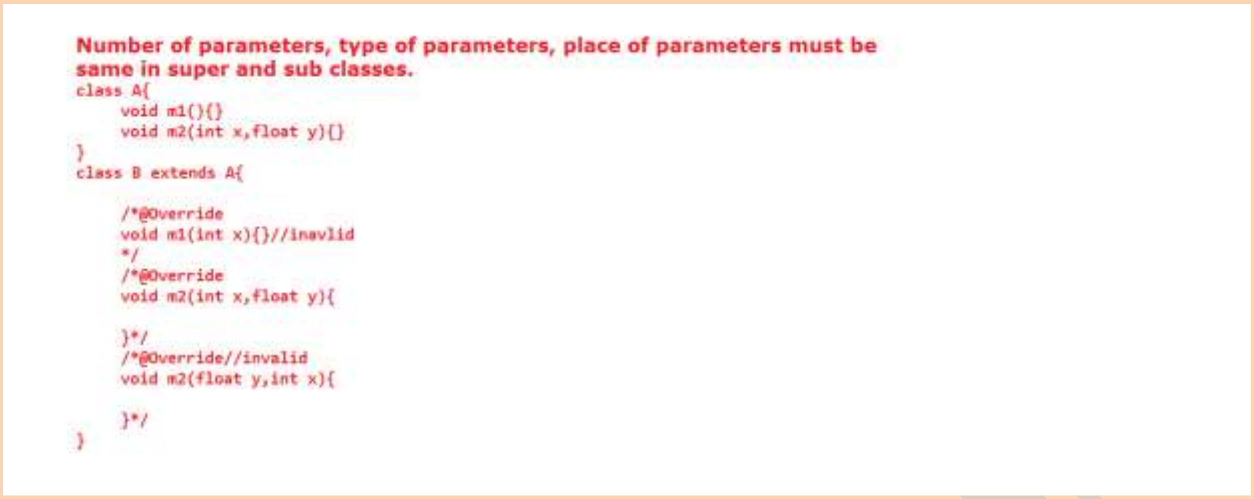


```
class A{
        void m1(){
        }
        void m2(){
        }
}
class B extends A{
        @Override
        void m1(){
        }

}
class C extends B{
        @Override
        void m2(){
        }
}
```

In C class m2 method is override method. The reason is m2() is indirectly existed in B class, with the help inheritace concept.

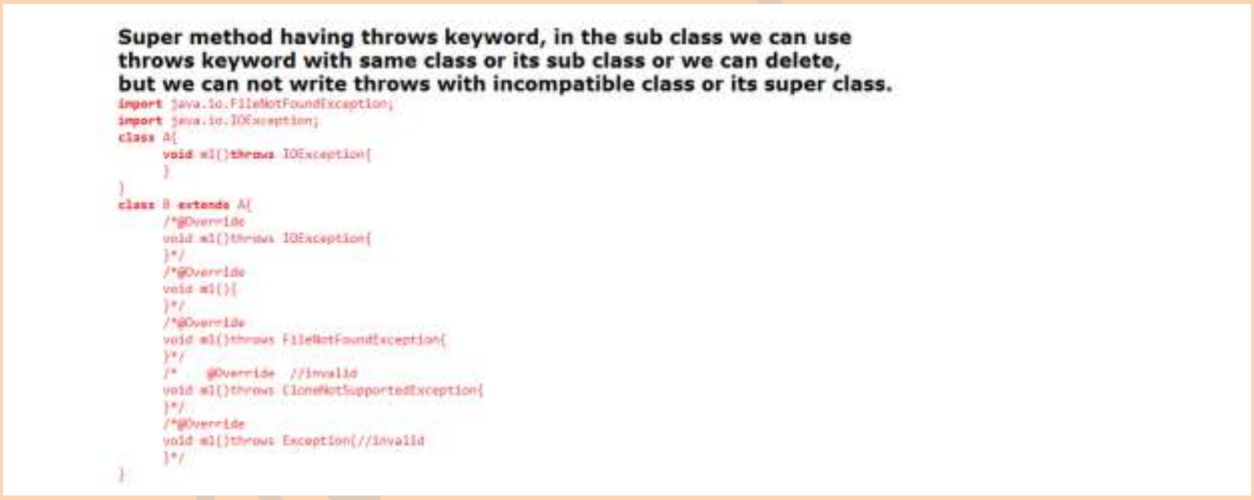
B is subclass A, so A class m2() method will come B class.

4) Number of parameters, type of parameters and place of parameters must be same.(Method signature must be same)


Number of parameters, type of parameters, place of parameters must be same in super and sub classes.

```
class A{
      void m1(){}
      void m2(int x,float y){}
}
class B extends A{

      /*@Override
      void m1(int x){}//inavlid
      */
      /*@Override
      void m2(int x,float y){

      }*/
      /*@Override//invalid
      void m2(float y,int x){

      }*/
}
```

5) throws clause exception class name can be same or we can delete throws clause or we can use throws clause with lower class(subclass) exception. But we cannot increase the exception level (superclass) and incompitable classes.


Super method having throws keyword, in the sub class we can use throws keyword with same class or its sub class or we can delete, but we can not write throws with incompatible class or its super class.

```
import java.io.FileNotFoundException;
import java.io.IOException;
class A{
      void m1()throws IOException{
      }
}
class B extends A{
      /*@Override
      void m1()throws IOException{
      }*/
      /*@Override
      void m1(){
      }*/
      /*@Override
      void m1()throws FileNotFoundException{
      }*/
      /*    @Override  //invalid
      void m1()throws CloneNotSupportedException{
      }*/
      /*@Override
      void m1()throws Exception{//invalid
      }*/
}
```

Note:

Method Overriding must be depends upon inheritance.

Variable name is not consider in the concept of method overloading and method overriding.

package inheritence;

import java.io.FileNotFoundException;

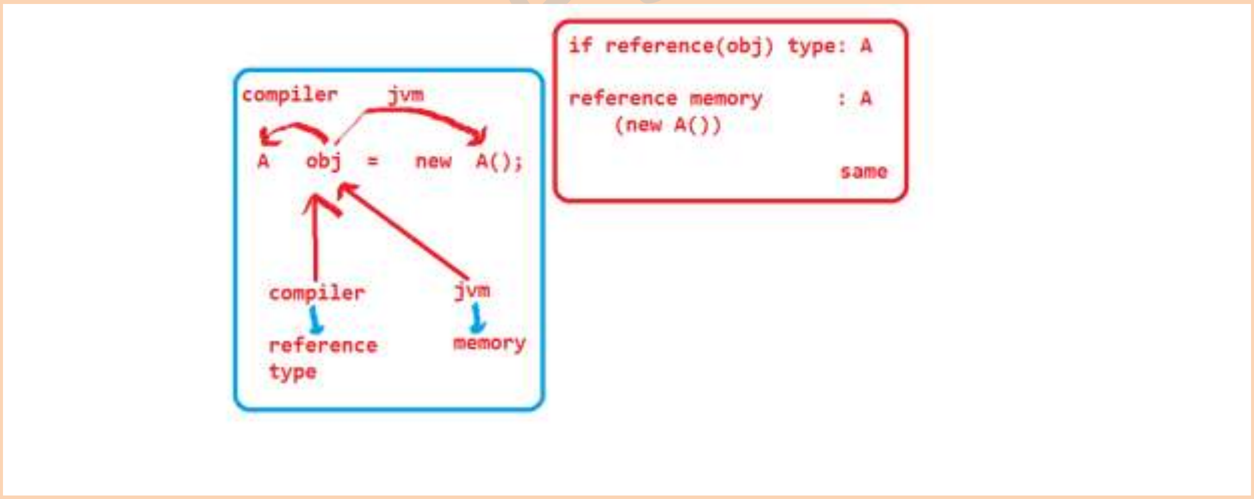
```
import java.io.IOException;
class L{
        void m1(){
        }
        void m2(int x, float y){
        }
        void m3() throws IOException{
        }
}
class M extends L{
        /*@Override
        void m3()throws IOException{
        }*/
        /* @Override
         * void m3(){
        }*/
        /*@Override
         void m3() throws FileNotFoundException{
        }*/
        /*@Override
        void m3()throws Exception{ //not override method
        }*/
        /*@Override
         * void m1(){
        }*/
        /*@Override
         * protected void m1(){
        }*/
        /*@Override
        public void m1(){
        }*/
        /*@Override
        public int m1(){//not override method
                return 100;
        }*/
        /*@Override
        public void m2(){
        }*/
        /*@Override
        void m2(int x){//not override method
        }*/
        /*@Override
        void m2(int x, int y){ //not override method
        }*/
        /*@Override
        void m2(float x, int y){//not override method
        }*/
}
```

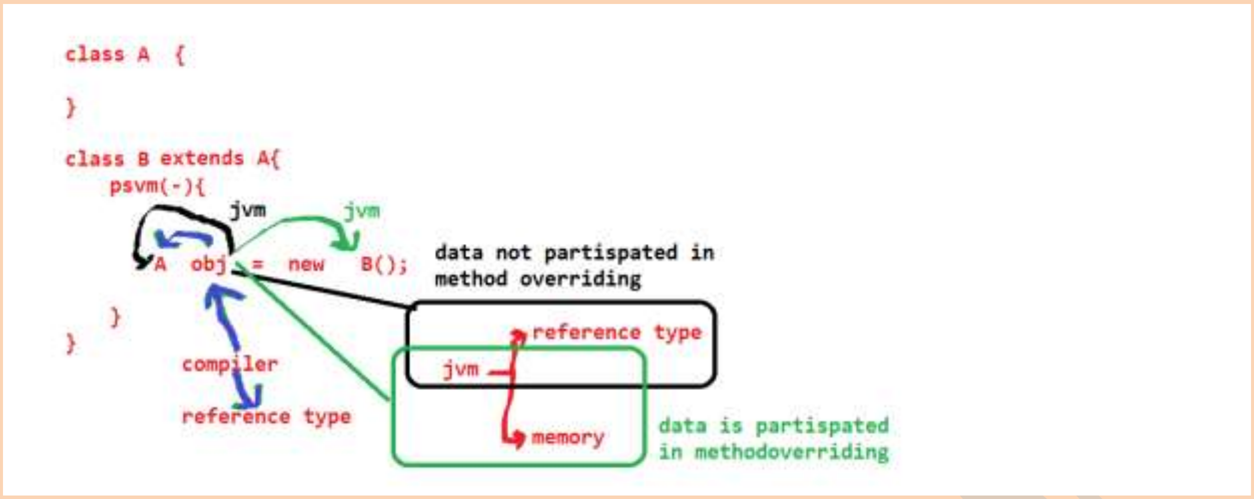
```
public class MethodOverridingDemo {

        public static void main(String[] args) {

        }

}
```

Note: private, static, final methods are not participated in the method overriding concept.

```
class M {

  private void m1(){

     System.out.println("M class m1 method");

   }

  static void m2(){

     System.out.println("M class m2 method");

  }

   final void m3(){

     System.out.println("M class m3 method");

   }

}

public class MethodOverridingDemo extends M {

/*@Override

private void m1(){

System.out.println("MethodOverridingDemo class m1
                          method");

}*/

/*@Override

        static void m2(){
                System.out.println("MethodOverridingDemo class m2 method");
}*/
/*final void m3(){
        System.out.println("MethodOverridingDemo class m3 method");
}*/
public static void main(String[] args) {
        MethodOverridingDemo md = new MethodOverridingDemo();
       //md.m1();
       // md.m2();
        }
}
```

If we are not using @Override annotation both private and static method are looks like participated in method overriding but they are not really participated in method overriding.





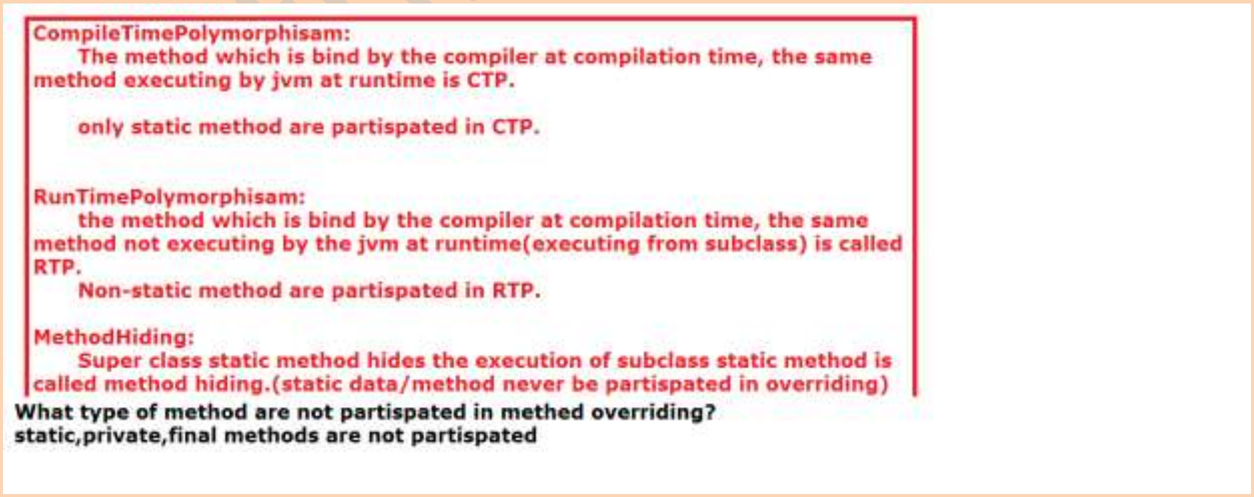
```
class A  {

}
class B extends A{
    psvm(-){
      A  obj  =  new   B();
    }
}
```

Static polymorphism: (Early binding)

A method which is bind by the compiler at compilation time, the same method is executed by the JVM at run time is called compile time polymorphism or static polymorphism.

Dynamic polymorphism: (lazy binding)

A method which is bind by the compiler at compilation time, the same method is not executed by the JVM at run time is called dynamic polymorphism or run time polymorphism or lazy binding.



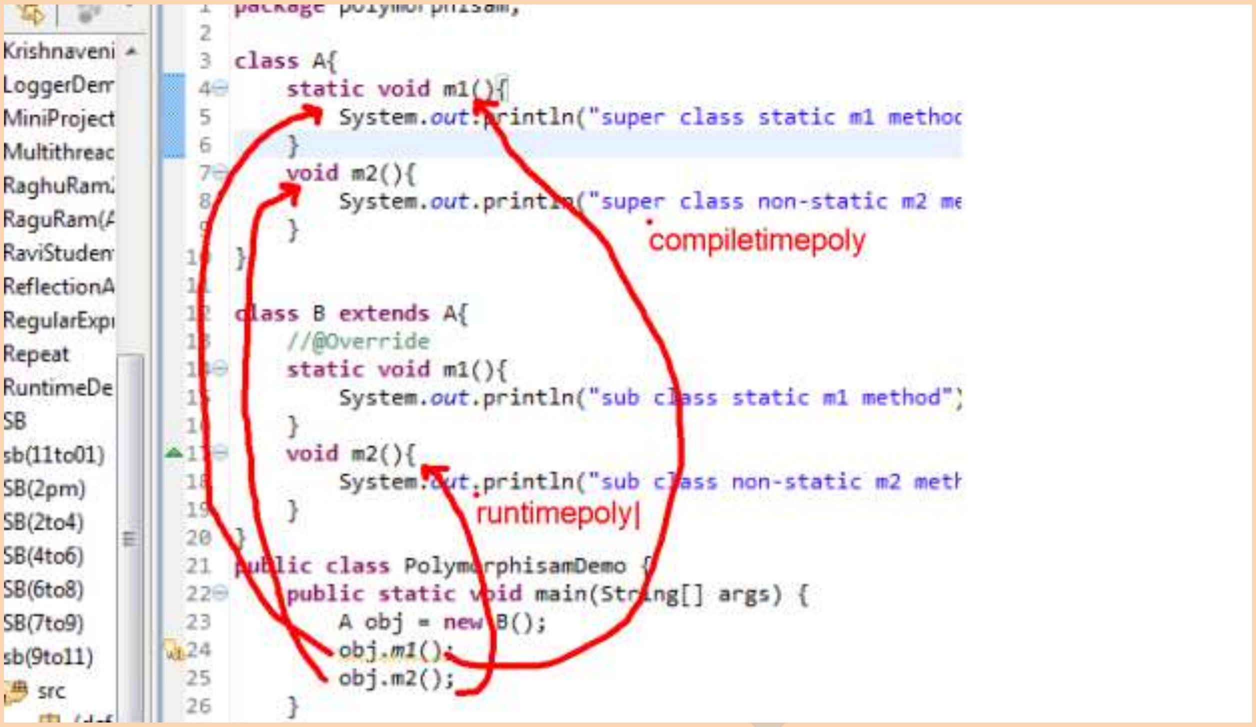
CompileTimePolymorphisam:
The method which is bind by the compiler at compilation time, the same method executing by jvm at runtime is CTP.

only static method are partispated in CTP.

RunTimePolymorphisam:
the method which is bind by the compiler at compilation time, the same method not executing by the jvm at runtime(executing from subclass) is called RTP.
Non-static method are partispated in RTP.

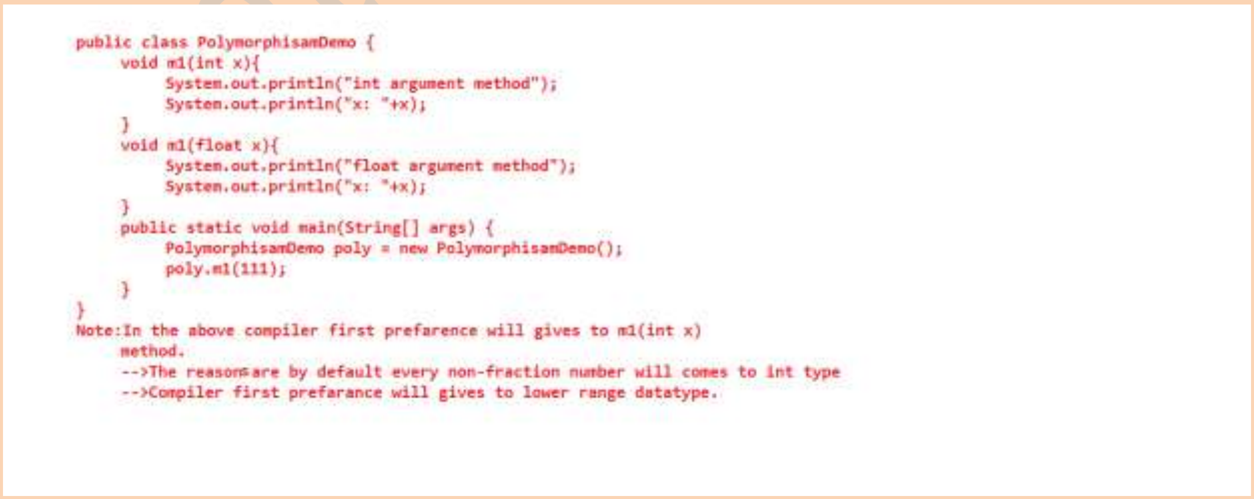
MethodHiding:
Super class static method hides the execution of subclass static method is called method hiding.(static data/method never be partispated in overriding)

What type of method are not partispated in methed overriding?
static,private,final methods are not partispated

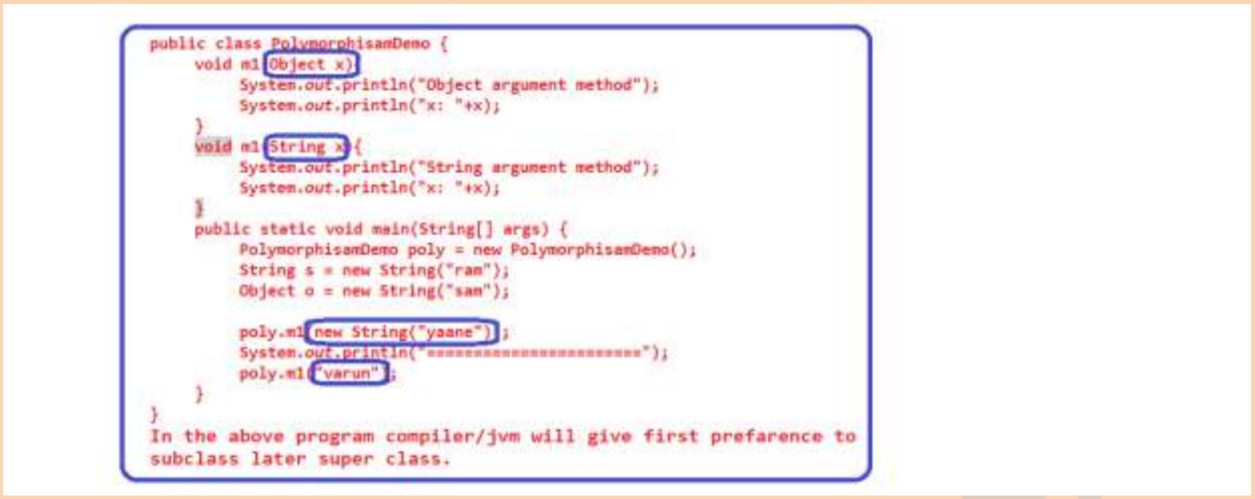


```
package polymorphisam;

class A{
    static void m1(){
        System.out.println("super class static m1 methoc
    }
    void m2(){
        System.out.println("super class non-static m2 me
    }
}

class B extends A{
    //@Override
    static void m1(){
        System.out.println("sub class static m1 method")
    }
    void m2(){
        System.out.println("sub class non-static m2 meth
    }
}
public class PolymorphisamDemo {
    public static void main(String[] args) {
        A obj = new B();
        obj.m1();
        obj.m2();
    }
```

```
class MM{
        void m1(){
         System.out.println("MM m1 method");
        }
        void m2(){
         System.out.println("MM m2 method");
        }
        static void m3(){
         System.out.println("MM m3 method");
        }
}
class NN extends MM{
        void m2(){
         System.out.println("NN m2 method");
        }
        static void m3(){
         System.out.println("NN m3 method");
        }
}
public class Polymorphisam {
        public static void main(String[] args) {
                NN obj = new NN();
                obj.m1();
                obj.m2();
                System.out.println("-------------");
                MM obj1 = new NN();//upcasting
                obj1.m1();//CTP
                obj1.m2();//RTP
                obj1.m3();//CTP
        }
}
```

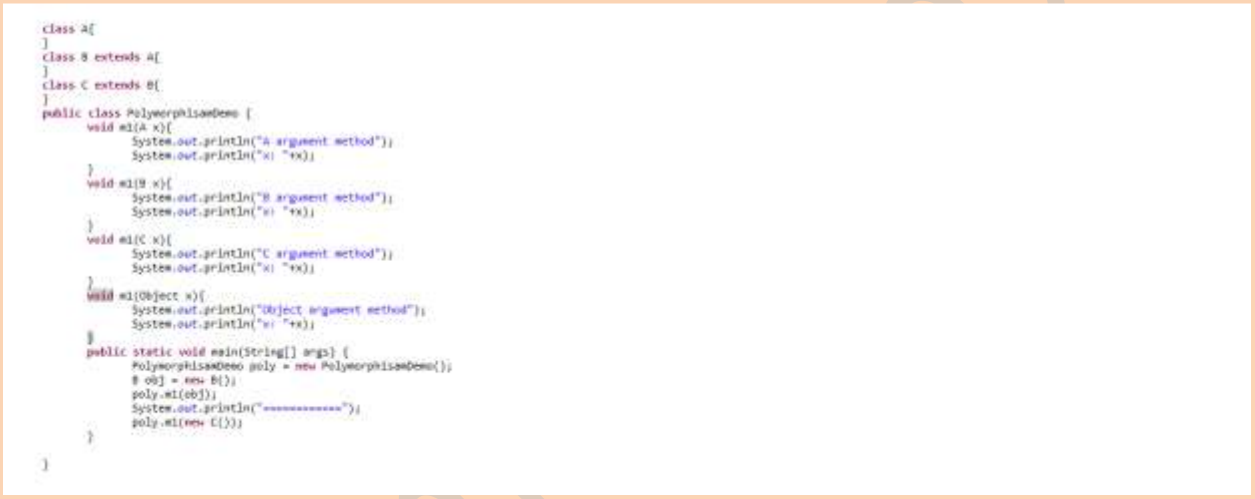


```
public class PolymorphisamDemo {
    void m1(int x){
        System.out.println("int argument method");
        System.out.println("x: "+x);
    }
    void m1(float x){
        System.out.println("float argument method");
        System.out.println("x: "+x);
    }
    public static void main(String[] args) {
        PolymorphisamDemo poly = new PolymorphisamDemo();
        poly.m1(111);
    }
}
Note:In the above compiler first prefarence will gives to m1(int x)
    method.
    -->The reasonsare by default every non-fraction number will comes to int type
    -->Compiler first prefarance will gives to lower range datatype.
```

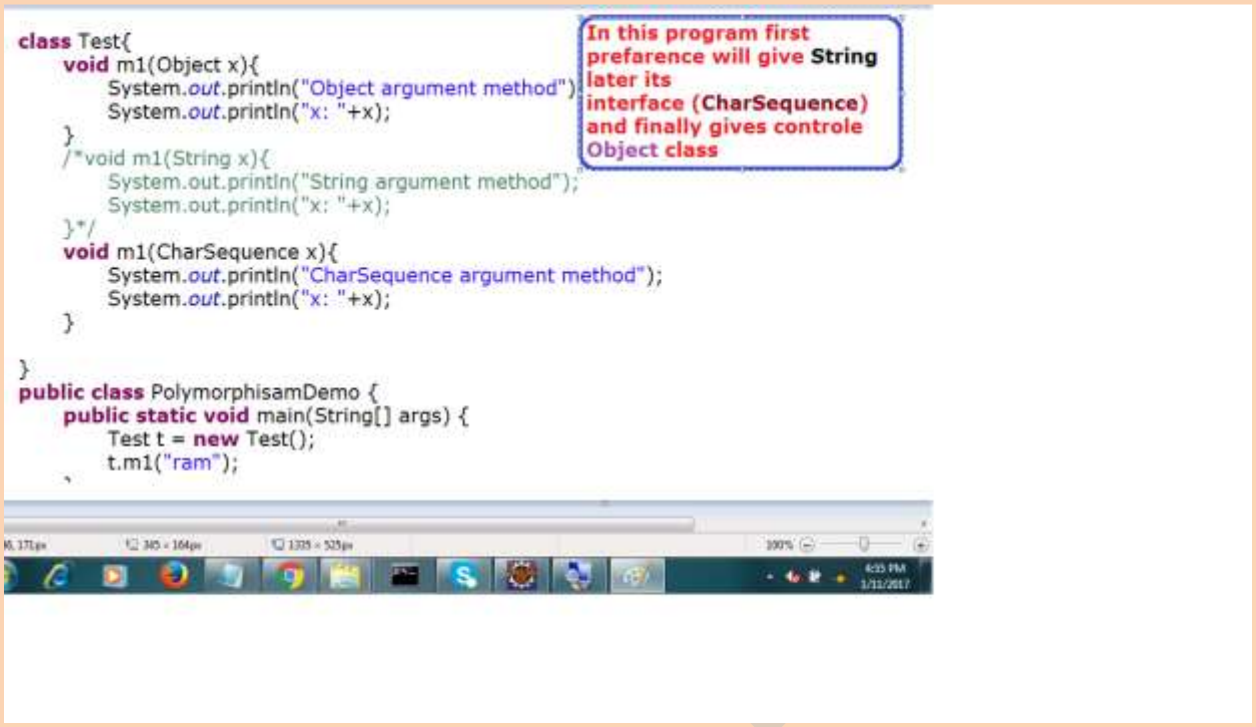


```
public class PolymorphisamDemo {
	void m1(Object x){
		System.out.println("Object argument method");
		System.out.println("x: "+x);
	}
	void m1(String x){
		System.out.println("String argument method");
		System.out.println("x: "+x);
	}
	public static void main(String[] args) {
		PolymorphisamDemo poly = new PolymorphisamDemo();
		String s = new String("ram");
		Object o = new String("sam");

		poly.m1(new String("yaane"));
		System.out.println("======================");
		poly.m1("varun");
	}
}
```

In the above program compiler/jvm will give first prefarence to subclass later super class.



```
class A{
}
class B extends A{
}
class C extends B{
}
public class PolymorphisamDemo {
	void m1(A x){
		System.out.println("A argument method");
		System.out.println("x: "+x);
	}
	void m1(B x){
		System.out.println("B argument method");
		System.out.println("x: "+x);
	}
	void m1(C x){
		System.out.println("C argument method");
		System.out.println("x: "+x);
	}
	void m1(Object x){
		System.out.println("Object argument method");
		System.out.println("x: "+x);
	}
	public static void main(String[] args) {
		PolymorphisamDemo poly = new PolymorphisamDemo();
		B obj = new B();
		poly.m1(obj);
		System.out.println("============");
		poly.m1(new C());
	}

}
```



```
class Test{
	void m1(Object x){
		System.out.println("Object argument method");
		System.out.println("x: "+x);
	}
	/*void m1(String x){
		System.out.println("String argument method");
		System.out.println("x: "+x);
	}*/
	void m1(CharSequence x){
		System.out.println("CharSequence argument method");
		System.out.println("x: "+x);
	}

}
public class PolymorphisamDemo {
	public static void main(String[] args) {
		Test t = new Test();
		t.m1("ram");
```

In this program first prefarence will give String later its interface (CharSequence) and finally gives controle Object class

Method Hiding:

Super class static method, which is hiding the execution of sub class static method, is called method hiding.

JVM is always give the preference to subclass memory (new NN()).

If it is non-static method then JVM is executing from subclass, so there is no method hiding.

But if it is static method then JVM is not executing from subclass, it is executing from super class is called method hiding.

In the above class non-static m2 method is comes under method overriding, static m1 method is comes under method hiding.

Co-Variant return types:

This functionality introduced in java 1.5 versions.

In the method overriding concept the method return type must be same up to java 1.4.

But from java 1.5 onwards we can write subclass override method return type can be subclass of super class method return type.

Note: Return type must be referenced type(non-primitive).

Ex:

```
class A{
        A m1(){
                new A();
        }
}
class B extends A{
        B m1(){//co-varient method
                return new B();
        }
}
```