Polymorphism: It is a process of performing multiple tasks with the same identity.

Polymorphism
- Compiletime Polymorphism [Method Overloading]
- Runtime Polymorphism [Method Overridding]

Method Overloading: "Writing two or more methods in the same class having same method name but different parameters."

Method Overridding: "Writing two or more methods in '2' different Classes having same method name, same parameters & same returntype."

1) If we want to perform Method overriding 100%, we need to achieve Inheritance
2) If we can't inherit a method we can't override. (Ex : private methods)
3) private > default > protected > public
4) Whenever we are performing method overriding child class method should not be more restrictive than the parent class method
5) We cannot override static method, It may seem like we are overriding but that concept is known as method hiding
6) In method hiding child class method will hide the implementation of the parent class method
7) After jdk 1.5 return types may not be same in co-variant return types. (Co-variant return type concept is applicable only for object types but not for primitives)

Method Overriding

- "Writing two or more methods in super and sub classes with the same name and same signature is called Method Overriding".
- It is also known as '**late-binding' or 'run-time'** polymorphism.
- The method present in super class is called overridden method and the method present in the sub class is called over ridding method.
- When an overridden method is called through a super class reference, Java determines which version of that method is to be executed based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time. That is the reason it is called Run-time polymorphism.

Overridden Rules

- The argument list must exactly match that of the overridden method.
- The return type should exactly match that of the overridden method (upto 1.4V).
- After jdk 1.5 return types may not be same in co-variant return types. (Co-variant return type concept is applicable **only for object types** but **not for primitives)**
- The access level must not be more restrictive that that of the over ridden method. (private < default < protected < public)
- If a method can't be inherited we cannot override it (Ex: private).

- We can't override a static method as non static methods and vice versa.
- If you are over ridding two static methods then it will be "method hiding".
- The overriding method must not throw new broader checked exceptions than those declared by the overridden method (to be discussed later) .
- For unchecked exceptions there are no restrictions.

Overloading Vs Overriding

Method Overloading	Method Overriding
It occurs with in the same class	It occurs with in the super class and sub class
Inheritance is not involved since it deals with only one class	Inheritance is involved because it occurs between Super and Sub classes
In Overloading Return type need not be the same	In Overriding Return type must be same
Parameters must be different when we do Overloading	Parameters must be same
In Overloading one method can't hide another method	In Overriding sub class method hides the super class methods

Private methods we cannot inherit, so we cannot override.

Overloading does not need inheritance.

Overloading occurs in the same class.

Overriding occurs in the 2 different classes.

```java
package com.pack1;

public class ClassA
{
    void meth1()
    {
        System.out.println("Class A meth1()");
    }
}
/*
 1) If we want to perform Method overridding 100
 2) If we cant inherit a method we cant override

 */
```

```java
package com.pack1;

public class ClassB extends ClassA
{
    void meth1()
    {
        System.out.println(10);
    }
    public static void main(String[] args)
    {
        ClassA aobj=new ClassB();
        aobj.meth1();
    }
}
```

```
<terminated> Cla
10
```

Actually, in the above case we are calling method of class A but here Class B method is executed this is called method overriding.

```java
package com.pack1;

public class ClassC
{
    public static void main(String[] args)
    {
        ClassC cobj1=new ClassC();
        ClassC cobj2=new ClassC();

        System.out.println("cobj1 : "+cobj1);
        System.out.println("cobj1 : "+cobj1.toString());

        System.out.println("\ncobj2 : "+cobj2);
        System.out.println("cobj2 : "+cobj2.toString());
    }
}
/*
     Whenever we are printing an Object of a class,
     internally compiler is going to call toString()
     which is present in the Object class
 */
```

```
cobj1 : com.pack1.ClassC@3feba861
cobj1 : com.pack1.ClassC@3feba861

cobj2 : com.pack1.ClassC@b4c966a
cobj2 : com.pack1.ClassC@b4c966a
```

```java
package com.pack1;

public class ClassC
{
    @Override // Annotation
    public String toString()
    {
        return "Java is awesome!!!";
    }
    public static void main(String[] args)
    {
        ClassC cobj1=new ClassC();
        ClassC cobj2=new ClassC();

        System.out.println("cobj1 : "+cobj1);
        System.out.println("cobj1 : "+cobj1.toString());

        System.out.println("\ncobj2 : "+cobj2);
        System.out.println("cobj2 : "+cobj2.toString());
    }
}
/*
     Whenever we are printing an Object of a class,
     internally compiler is going to call toString()
```

```
cobj1 : Java is awesome!!!
cobj1 : Java is awesome!!!

cobj2 : Java is awesome!!!
cobj2 : Java is awesome!!!
```

In the above case we are inheriting object class internally.

An object class is parent class for all the classes in java by default we need not inherit(extends) or import

To override to string() in child class we are writing same method name by this we are overriding to String() from object class.

Why do we need to override?

I am not satisfied with that method implementation I want to write my own implementation.

Then why can't you take the new method name.

There are some situations where you cannot change the method name, but we can change its method implementation.

4key points in inheritance

1 is for has a relationship

2 is for method overriding

3 is for is a relationship (actual inheritance)

4th key point invalid

Method overriding is also known as late-binding or run time polymorphism.

When an overridden method is called through parent class reference java determines which version of that method is to be called based upon the type of object being referred to whenever you're calling that method thus this determination will done at runtime.

```java
package com.pack1;

public class ClassA
{
    void meth1() // overridden method
    {
        System.out.println("Class A meth1()");
    }
    void meth2(int x)
    {
        System.out.println("Class A meth2()");
    }
}
/*
 1) If we want to perform Method overridding 100%
 2) If we cant inherit a method we cant override.

 */
```

```java
package com.pack1;

public class ClassB extends ClassA
{
    void meth1() // overridding method
    {
        System.out.println(10);
    }
    @Override
    void meth2(int num)
    {
        System.out.println(20);
    }
    public static void main(String[] args)
    {
        ClassA aobj1=new ClassA(); // 1st-point
        aobj1.meth1();

        ClassA aobj2=new ClassB();  // 2nd-point
        aobj2.meth1();
        aobj2.meth2(100);
    }
}
```

```
<terminated> ClassB [Jav
Class A meth1()
10
20
```

```java
package com.pack1;

public class ClassA
{
    void meth1() // overridden method
    {
        System.out.println("Class A meth1()");
    }
    void meth2(int x)
    {
        System.out.println("Class A meth2()");
    }
    void meth3(int x, String s)
    {
        System.out.println("Class A meth3()");
    }
}
/*
 1) If we want to perform Method overridding 100%
 2) If we cant inherit a method we cant override.

 */
```

```java
    void meth1() // overridding method
    {
        System.out.println(10);
    }
    @Override
    void meth2(int num)
    {
        System.out.println(20);
    }
    @Override
    void meth3(int i, String msg)
    {
        System.out.println(30);
    }

    public static void main(String[] args)
    {
        ClassA aobj1=new ClassA(); // 1st-point
        aobj1.meth1();

        ClassA aobj2=new ClassB();  // 2nd-point
        aobj2.meth1();
        aobj2.meth2(100);
        aobj2.meth3(100, "Java");
    }
}
```

```
<terminated> ClassB [Java
Class A meth1()
10
20
30
```

Is it mandatory that both access modifiers of method should be same?

The answer is no

Can we override static method with a non-static method?

The answer is no

```java
    }
    static void meth4()
    {
        System.out.println("Class A meth3()");
    }
}
```

```java
    }
    void meth4()
    {
        System.out.println(40);
    }
```

Can we override static method with a static method?

The answer is no

```
Training ▸ src ▸ com.pack1 ▸ ClassA ▸ meth4() : void
    protected void meth2(int x)
    {
        System.out.println("Class A meth2()");
    }
    void meth3(int x, String s)
    {
        System.out.println("Class A meth3()");
    }
    static void meth4()
    {
        System.out.println("Class A meth3()");
    }
}
/*
 1) If we want to perform Method overridding 100% we
 2) If we cant inherit a method we cant override. (E
 3) private > default > protected > public
 4) Whenever we are performing method overriding chi
    should not be more restrictive than the parent c
 */
```

```
Training ▸ src ▸ com.pack1 ▸ ClassB ▸
    public void meth2(int num)
    {
        System.out.println(20);
    }
    @Override
    void meth3(int i, String msg)
    {
        System.out.println(30);
    }
    static void meth4()
    {
        System.out.println(40);
    }

    public static void main(String[] args)
    {
        ClassA aobj1=new ClassA(); // 1st-point
        aobj1.meth1();

        ClassA aobj2=new ClassB();  // 2nd-point
        aobj2.meth1();
        aobj2.meth2(100);
        aobj2.meth3(100, "Java");
        aobj2.meth4();
    }
}
```

```
<terminated> ClassB [Java A
Class A meth1()
10
20
30
Class A meth3()
```

Static belongs to the class not to the object

It may seem like you are overriding but no, this concept is known as a method hiding. We should not write annotation because we will get an error.

In method hiding child class method will hide the implementation of the parent class method.

```java
    protected void meth2(int x)
    {
        System.out.println("Class A meth2()");
    }
    void meth3(int x, String s)
    {
        System.out.println("Class A meth3()");
    }
    static void meth4()
    {
        System.out.println("Class A meth4()");
    }
    ClassA meth5()
    {
        System.out.println("Class A meth5()");
        return new ClassA();
    }
}
/*
 1) If we want to perform Method overridding 100% we need t
 2) If we cant inherit a method we cant override. (Ex : pri
 3) private > default > protected > public
 4) Whenever we are performing method overriding child clas
    should not be more restrictive than the parent class me
 5) We cannot override static method,It may seem like we ar
 6) In method hiding child class method will hide the imple
 */
```

```java
    public void meth2(int num)
    {
        System.out.println(20);
    }
    @Override
    void meth3(int i, String msg)
    {
        System.out.println(30);
    }
    static void meth4()
    {
        System.out.println(40);
    }
    ClassA meth5()
    {
        System.out.println(50);
        return new ClassA();
    }
    public static void main(String[] args)
    {
        ClassA aobj1=new ClassA(); // 1st-point
        aobj1.meth1();

        ClassA aobj2=new ClassB();  // 2nd-point
        aobj2.meth1();
        aobj2.meth2(100);
        aobj2.meth3(100, "Java");
        aobj2.meth4();
        aobj2.meth5();
```

```
<terminated> ClassB [Java
Class A meth1()
10
20
30
Class A meth4()
50
```

```
Training ▸ src ▸ com.pack1 ▸ ClassA ▸ meth5() : ClassA
```

```
Training ▸ src ▸ com.pack1 ▸ ClassB ▸ meth4() : void
```

Considering the above two images of method 5

This process is known as covariant return types applicable for classes not for primitive data types

After jdk 1.5 return types may not be same in co-variant return types. (Co-variant return type concept is applicable only for object types but not for primitives)

Can we override a constructor?

No, we have 2 reasons

1. We cannot inherit a constructor, without inheritance we cannot override.
2. In class A the constructor name will be class A and in class B the constructor name will be Class B, these are not the same.

```java
package com.pack1;

public class ClassA
{
    void meth1() // overridden method
    {
        System.out.println("Class A meth1()");
    }
    protected void meth2(int x)
    {
        System.out.println("Class A meth2()");
    }
    void meth3(int x, String s)
    {
        System.out.println("Class A meth3()");
    }
    static void meth4()
    {
        System.out.println("Class A meth4()");
    }
    ClassA meth5()
    {
        System.out.println("Class A meth5()");
        return new ClassA();
    }
}
```

```java
public class ClassB extends ClassA
{
    protected void meth1() // overridding method
    {
        System.out.println(10);
    }
    @Override
    public void meth2(int num)
    {
        System.out.println(20);
    }
    @Override
    void meth3(int i, String msg)
    {
        System.out.println(30);
    }
    static void meth4()
    {
        System.out.println(40);
    }
    ClassB meth5()
    {
        System.out.println(50);
        return new ClassB();
    }
    public static void main(String[] args)
```

```java
    {
        ClassA aobj1=new ClassA(); // 1st-point
        aobj1.meth1();

        ClassA aobj2=new ClassB();  // 2nd-point
        aobj2.meth1();
        aobj2.meth2(100);
        aobj2.meth3(100, "Java");
        aobj2.meth4();
        aobj2.meth5();
    }
```

```java
package com.pack1;

public class ClassC
{
    @Override // Annotation
    public String toString()
    {
        return "Java is awesome!!!";
    }
    public static void main(String[] args)
    {
        ClassC cobj1=new ClassC();
        ClassC cobj2=new ClassC();

        System.out.println("cobj1 : "+cobj1);
        //System.out.println("cobj1 : "+cobj1.toString());

        System.out.println("\ncobj2 : "+cobj2);
        //System.out.println("cobj2 : "+cobj2.toString());
    }
}
/*
     Whenever we are printing an Object of a class,
     internally compiler is going to call toString()
     which is present in the Object class
 */
```

private > default > protected > public

parent class child class

private invalid(inheritance)

public----------public

protected ----------protected & public

default ---------- default, protected & public

```java
package com.pack1;

public class ClassC
{
    @Override // Annotation
    public String toString()
    {
        return "Java is awesome!!!";
    }
    public static void main(String[] args)
    {
        Object cobj1=new ClassC();
        Object cobj2=new ClassC();

        System.out.println("cobj1 : "+cobj1);
        //System.out.println("cobj1 : "+cobj1.toSt

        System.out.println("\ncobj2 : "+cobj2);
        //System.out.println("cobj2 : "+cobj2.toSt
    }
}
/*
     Whenever we are printing an Object of a class
     internally compiler is going to call toString
     which is present in the Object class
 */
```

```
<terminated> ClassC [Java Application] C:\
cobj1 : Java is awesome!!!

cobj2 : Java is awesome!!!
```

Can the final method override?

The Answer is no (wait for the coming classes)