3.8. Method Overriding

- Method overriding is a process of implementing super class methods in sub class with the same signature.

Lab438.java

```
class Hai{
void m1(){
System.out.println("Hai - m1()");
}
void m2(){
System.out.println("Hai - m2()");
}
}
class Hello extends Hai{
void m2(){
System.out.println("Hello - m2()");
}
void m3(){
System.out.println("Hello - m3()");
}
}
```

```
public class Lab438{
public static void main(String args[]){
Hello hello = new Hello();
hello.m1();
hello.m2();
hello.m3();
}
}
```

Here in above program:

- Hai class has implemented m2() method of Hello class with the same signature i.e. m2() method of Hello class is overridden in Hai class.
- m1() method of Hello class is not implemented in Hai class but m1() method of Hello class is inherited to Hai class.
- m3() method is the newly added method in Hai class.

3.8.1 Why to override the method?

- According to Object Oriented best practices "Classes are closed for modifications" i.e. if you want to modify the functionality of existing class then you should not disturb existing class.
- It is always better to write a sub class and provide the required implementations in sub class.

- Subclass can be written for following purposes:
  - To add new functionality.
  - To modify existing functionality.
  - To inherit existing functionality.

3.8.2 Rules to override the method

1) **Subclass method name must be same as super class method name.**
2) **Subclass method parameters (type, order and number) must be same as super class method parameters.**
3) **Subclass method return type must be same as super class method return type.**

   Note: If the super class method return type is class type then while overriding the method in subclass you can use same class type or its subclass as return type. (FROM JAVA 5)

```
class Hello{                                        class A{ }
  A m1(){... }
}                                                   class B extends A{}
class Hai extends Hello{
A m1(){... }                                        class C{}
            or
B m1(){... }            // Valid from Java 5
C m1(){... }            // Invalid
}
```

4) **Subclass method access modifier must be same or higher than super class method access modifier.**

Super Class	Sub Class
public	public
protected	protected, public
default	default, protected, public
private	private, default, protected, public

5) **When super class method is instance method then you have to override in sub class as instance only.**
6) **When super class method is static method then you have to override in sub class as static only.**
7) **When super class method is throwing some method level checked exception then sub class method can do the following :**
   a. **Subclass method can ignore that method level exception.**
   b. **Subclass method can throw the same exception.**
   c. **Subclass method can throw the exception which is sub class to super class method exception.**
   d. **Subclass method cannot throw the exception which is super class to super class method exception.**
   e. **Subclass method cannot throw the exception which is non subclass to super class method exception.**
   f. **Subclass method can throw any unchecked exception.**

8) **When super class method is throwing some method level unchecked exception then sub class method can do the following :**
   a. **Subclass method can ignore that method level exception.**
   b. **Subclass method can throw the same exception.**
   c. **Subclass method can throw any other unchecked exception.**
   d. **Subclass method cannot throw any checked exception.**

Note: Rule No. 7 and Rule No. 8 will be fully explored with EXCEPTION HANDLING.

Lab439.java

```
class Hai{
void m1(){
System.out.println("Hai - m1()");
}
}
class Hello extends Hai{
void M1(){
System.out.println("Hello - M1()");
}
}
public class Lab439{
public static void main(String args[]){
Hello hello = new Hello();
hello.m1();
hello.M1();
}
}
```

Lab440.java

```
class Hai{
void m1(int a){
System.out.println("Hai - m1(int)");
}
}
class Hello extends Hai{
void m1(String str){
System.out.println("Hello - m1(String)");
}
}
public class Lab440{
public static void main(String args[]){
Hello hello = new Hello();
hello.m1(10);
hello.m1("JLC");
}
}
```

Lab441.java

```
class Hai{
void m1(int a){
System.out.println("Hai - m1(int)");
}
}
class Hello extends Hai{
int m1(int a){
System.out.println("Hello - m1(int)");
return 9;
}
}
public class Lab441{
public static void main(String args[]){
Hello hello = new Hello();
hello.m1(10);
}
}
```

Lab442.java

```
class Hai{
long m1(int a){
System.out.println("Hai - m1(int)");
return 9;
}
}
class Hello extends Hai{
long m1(int a){
System.out.println("Hello - m1(int)");
return 9;
}
}
public class Lab442{
public static void main(String args[]){
Hello hello = new Hello();
hello.m1(10);
} }
```

Lab443.java

```
class Hai{
long m1(int a){
System.out.println("Hai - m1(int)");
return 9;
}
}
class Hello extends Hai{
int m1(int a){
System.out.println("Hello - m1(int)");
return 9;
}
}

public class Lab443{
public static void main(String args[]){
Hello hello = new Hello();
hello.m1(10);
}
}
```

Lab444.java

```
class A{ }
class B extends A{ }
class C extends B{}
class D{}

class Hai{
B m1(){
System.out.println("Hai - m1()");
return new B();
}
}
class Hello extends Hai{
B m1(){
System.out.println("Hello - m1()");
return new B();
}
}

public class Lab444{
public static void main(String args[]){
Hello hello = new Hello();
hello.m1();
}
}
```

Lab445.java

```
class A{ }
class B extends A{ }
class C extends B{}
class D{}

class Hai{
B m1(){
System.out.println("Hai - m1()");
return new B();
}
}
class Hello extends Hai{
C m1(){
System.out.println("Hello - m1()");
return new C();
}
}

public class Lab445{
public static void main(String args[]){
Hello hello = new Hello();
hello.m1();
}
}
```

Lab446.java

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

class Hai{
B m1(){
System.out.println("Hai - m1()");
return new B();
}
}
class Hello extends Hai{
A m1(){
System.out.println("Hello - m1()");
return new A();
}
}

public class Lab446{
public static void main(String args[]){
Hello hello = new Hello();
hello.m1();
}
}
```

Lab447.java

```
class Hai{
final void m1(){
System.out.println("Hai - m1()");
}
}
class Hello extends Hai{
void m1(){
System.out.println("Hello - m1()");
}
}

public class Lab447{
public static void main(String args[]){
Hello hello = new Hello();
hello.m1();
}
}
```

Lab448.java

```
class Hai{
void m1(){
System.out.println("Hai - m1()");
}
}
class Hello extends Hai{
final void m1(){
System.out.println("Hello - m1()");
}
}

public class Lab448{
public static void main(String args[]){
Hello hello = new Hello();
hello.m1();
}
}
```

Lab449.java

```
class Hai{
public void m1(){
System.out.println("Hai - m1()");
}
}
class Hello extends Hai{
public void m1(){
System.out.println("Hello - m1()");
}
}

public class Lab449{
public static void main(String args[]){
Hello hello = new Hello();
hello.m1();
}
}
```

Lab450.java

```
class Hai{
public void m1(){
System.out.println("Hai - m1()");
}
}
class Hello extends Hai{
void m1(){
System.out.println("Hello - m1()");
}
}

public class Lab450{
public static void main(String args[]){
Hello hello = new Hello();
hello.m1();
}
}
```

Lab451.java

```
class Hai{
protected void m1(){
System.out.println("Hai - m1()");
}
}
class Hello extends Hai{
protected void m1(){
System.out.println("Hello - m1()");
}
}

public class Lab451{
public static void main(String args[]){
Hello hello = new Hello();
hello.m1();
}
}
```

Lab452.java

```
class Hai{
protected void m1(){
System.out.println("Hai - m1()");
}
}
class Hello extends Hai{
public void m1(){
System.out.println("Hello - m1()");
}
}

public class Lab452{
public static void main(String args[]){
Hello hello = new Hello();
hello.m1();
}
}
```

Lab453.java

```
class Hai{
protected void m1(){
System.out.println("Hai - m1()");
}
}
class Hello extends Hai{
void m1(){
System.out.println("Hello - m1()");
}
}

public class Lab453{
public static void main(String args[]){
Hello hello = new Hello();
hello.m1();
}
}
```

Lab454.java

```
class Hai{
void m1(){
System.out.println("Hai - m1()");
}
}
class Hello extends Hai{
void m1(){
System.out.println("Hello - m1()");
}
}

public class Lab454{
public static void main(String args[]){
Hello hello = new Hello();
hello.m1();
}
}
```

Lab455.java

```
class Hai{
void m1(){
System.out.println("Hai - m1()");
}
}
class Hello extends Hai{
public void m1(){
System.out.println("Hello - m1()");
}
}

public class Lab455{
public static void main(String args[]){
Hello hello = new Hello();
hello.m1();
}
}
```

Lab456.java

```
class Hai{
void m1(){
System.out.println("Hai - m1()");
}
}
class Hello extends Hai{
private void m1(){
System.out.println("Hello - m1()");
}
}

public class Lab456{
public static void main(String args[]){
Hello hello = new Hello();
hello.m1();
}
}
```

Lab457.java

```
class Hai{
private void m1(){
System.out.println("Hai - m1()");
}
}
class Hello extends Hai{
void m1(){
System.out.println("Hello - m1()");
}
}

public class Lab457{
public static void main(String args[]){
Hello hello = new Hello();
hello.m1();
}
}
```

Lab458.java

```
class Hai{
void m1(){
System.out.println("Hai - m1()");
}
}
class Hello extends Hai{
void m1(){
System.out.println("Hello - m1()");
}
}

public class Lab458{
public static void main(String args[]){
Hello hello = new Hello();
hello.m1();
}
}
```

Lab459.java

```
class Hai{
void m1(){
System.out.println("Hai - m1()");
}
}
class Hello extends Hai{
static void m1(){
System.out.println("Hello - m1()");
}
}

public class Lab459{
public static void main(String args[]){
Hello hello = new Hello();
hello.m1();
}
}
```

Lab460.java

```
class Hai{
static void m1(){
System.out.println("Hai - m1()");
}
}
class Hello extends Hai{
static void m1(){
System.out.println("Hello - m1()");
}
}

public class Lab460{
public static void main(String args[]){
Hello hello = new Hello();
hello.m1();
}
}
```

Lab461.java

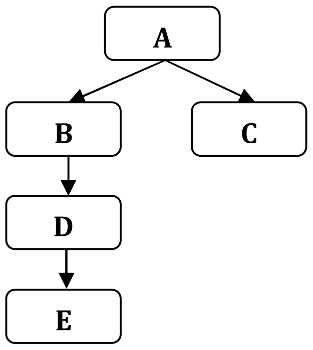
```
class Hai{
static void m1(){
System.out.println("Hai - m1()");
}
}
class Hello extends Hai{
void m1(){
System.out.println("Hello - m1()");
}
}

public class Lab461{
public static void main(String args[]){
Hello hello = new Hello();
hello.m1();
}
}
```

3.9 Dynamic Dispatch

The process of assigning subclass object to super class reference variable is called as dynamic dispatch.

Which of the following are valid?

A ob = new A();	//Valid
A ob = new B();	//Valid
A ob = new D();	//Valid
B ob = new B();	//Valid
B ob = new A();	//Invalid
B ob = new C();	//Invalid
B ob = new D();	//Valid
D ob = new A();	//Invalid
A ob = new E();	//Valid



Lab462.java

```
class Hai{
void m1(){
System.out.println("Hai - m1()");
}
}

class Hello extends Hai{
void m1(){
System.out.println("Hello - m1()");
}
}

public class Lab462{
public static void main(String args[]){
Hai hai = new Hello();
hai.m1();
}
}
```

Lab463.java

```
class Hai{
static void m1(){
System.out.println("Hai - m1()");
}
}

class Hello extends Hai{
static void m1(){
System.out.println("Hello - m1()");
}
}

public class Lab463{
public static void main(String args[]){
Hai hai = new Hello();
hai.m1();
}
}
```

SUMMARY

1) When you modify the method arguments in sub class then it is method overloading not overriding.
2) When your super class method return type is primitive then you have to use exactly same return type while overriding this method in subclass.
3) When your super class method return type is class Type then you can use exactly same return type or covariant type (sub class of super class method return type) while overriding this method in subclass.
4) Final method of super class cannot be overridden in sub class.
5) Non final method of super class can be overridden as final in sub class.
6) Native method of super class can be overridden as non native in sub class.
7) Non native method of super class can be overridden as native in sub class.
8) strictfp method of super class can be overridden as non strictfp in sub class.
9) Non strictfp method of super class can be overridden as strictfp in sub class.
10) synchronized method of super class can be overridden as non synchronized in sub class.
11) Non synchronized method of super class can be overridden as synchronized in sub class.
12) Private methods of super class will not be visible to subclasses. So whatever method you are writing in sub class that will be treated as new method not overridden method.
13) Access modifiers of subclass method must be same or higher then super class method.
14) In the case of method overriding:
    a. When you invoke the method with super class object then method will be called from super class only.
    b. When you invoke the method with sub class object then method will be called from subs class only.

3.10 Polymorphism

- Polymorphism (One name - Many forms).
- Polymorphism is the ability of an object to behave differently at different situations.

Types of Polymorphism

1. Static Polymorphism
2. Dynamic Polymorphism

Static Polymorphism

- Static Polymorphism can be achieved using method overloading.
- It can be achieved with both instance and static methods.
- Static Polymorphism is also called as compile time polymorphism or early binding because Java compiler is responsible to bind the method calls with actual method at compile time.
- Java compiler will verify the method name first then method parameters to verify the matching method.
- Return type will not be considered.

```
class Arithmetic{
      void   sum(double d){ ...}
      void   sum(int a, boolean b){ ...}
}
...
Arithmetic   ar  = new Arithmetic();
      1.   ar.sum1();
      2.   ar.sum(10, 20);
      3.   ar.sum(10.0);
```

Case 1 : ar.sum1()

- Compiler will do the following tasks:
  - sum1() method will be verified in the class.
  - sum1() method is not available, so compile time error will be given.

Case 2 : ar.sum(10, 20)

- Compiler will do the following tasks:
  - sum() method will be verified in the class.
  - sum() method is available.
  - Now method parameters will be verified.
  - Method with given parameters is not available, So compile time error will be given.

Case 3 : ar.sum(10.0);

- Compiler will do the following tasks:
  - sum() method will be verified in the class.
  - sum() method is available.
  - Now method parameters will be verified.
  - Method with given parameters is available.
  - Compiler will bind this method calls to actual matched method and it will go to next line.

Lab464.java

```
class Hello {
void sum(int a,int b){
System.out.println("Hello - sum(int,int)");
}
void sum(int a,int b,int c){
System.out.println("Hello - sum(int,int,int)");
}
}
```

```
public class Lab464{
public static void main(String args[]){
Hello  h = new Hello();

h.sum(10);
h.sum(10,20);
h.sum(10,20,30);

}
}
```

Dynamic Polymorphism

- Dynamic Polymorphism can be achieved using both method overriding and dynamic dispatch.
- It is also called as runtime polymorphism or late binding because JVM is responsible to decide which method will be invoked based on actual object available in reference variable.
- It can be achieved only with instance method, can't be achieved using static method.

A.java

```
class A{
void m1(){
System.out.println("A - m1()");
}
void m2(){
System.out.println("A - m2()");
}
}
```

B.java

```
class B extends A{
void m2(){
System.out.println("B - m2()");
}
void m3(){
System.out.println("B - m3()");
}
}
```

C.java

```
class C extends A{
void m2(){
System.out.println("C - m2()");
}
void m4(){
System.out.println("C - m4()");
}
}
```

Lab465.java

```
public class Lab465{
public static void main(String args[]){
A ao=null;

ao=new B();
ao.m1();  //Inherited
ao.m2(); // Overriden

ao = new C();
ao.m1();  //Inherited
ao.m2(); // Overriden

}
}
```

Lab466.java

```
public class Lab466{
public static void main(String args[]){
A ao=null;

ao=new B();
ao.m3();  // Newly Added

ao = new C();
ao.m4(); // Newly Added

}
}
```

Lab467.java

```
public class Lab467{
public static void main(String args[]){
A ao=null;

ao=new B();
ao.m1();

ao = new C();
ao.m1();

}
}
```

Lab468.java

```
class A{
int x=10;
static int y=20;
}

class B extends A{
String x="JLC";
static String y="MyJLC";
}
```

```
public class Lab468{
public static void main(String args[]){
A ao= new B();
System.out.println(ao.x);
System.out.println(ao.y);
}
}
```

SUMMARY

Runtime Polymorphism

1) **When you are calling an instance method with super class reference variable which contains subclass object then method calls depends on the type of object available in reference variable.**
2) **When you are calling a static method with super class reference variable which contains subclass object then method calls depends on the type of reference variable.**
3) **When you are calling a method with super class reference variable which contains subclass object then that method signature should be available in super class.**
4) **When you have variable in super class and sub classes with the same name then:**
    a. **When you refer variable with super class reference which contains subclass object then variable will be referred from super class only.**
    b. **When you refer variable with sub class reference which contains subclass object then variable will be referred from sub class only.**
    c. **When you want to refer variable of sub class, with super class reference which contains subclass object then you need to type cast reference into sub class.**
5) **When you refer any variable with super class reference variable which contains subclass object then that variable should be available in super class.**
6) **Overriding concept available only with method not with variable. When you define same variable in subclass then it is variable hiding.**