Course code : **CSE2005**

Course title : **Object Oriented Programming**

Method Overriding / Polymorphism

Objectives

This session will give the knowledge about

- Describe method overloading
- Describe method overriding
- Describe dynamic method dispatch, or runtime polymorphism
- Understand the use of instanceof operator

Method Overloading

- Like many OOPs languages, Java also provides method overloading feature.
- Overloading refers to the methods in a class having same name but different arguments.
- If same method names are used to define multiple methods in a class, then the methods are said to be overloaded.
- Overloaded methods must have same name but there must be a difference in other parts of the signature.
- Signature of a method includes the name of the method and its parameters (excluding the return type).

Method Overloading

Possible method overloading

By changing number of parameter

```
package vit.demo;
public class Test {
    void add(){
    }
    void add(int a) {
    }
}
```

By changing type of parameter

```
package vit.demo;
public class Test {
    void add(int a){
    }
    void add(float a){
    }
}
```

By changing order of parameter

```
package vit.demo;
public class Test {
    void add(int a, float b){
    }
    void add(float a, int b){
    }
}
```

Method Overloading

What is **NOT** a method overloading

By changing return type

```
package vit.demo;
public class Test {
    void add(){
    }
    int add() {
    }
}
```

By changing method type

```
package vit.demo;
public class Test {
    void add(){
    }
    int static add() {
    }
}
```

Belongs to different class

```
package vit.demo;
public class Test {
    void add(int a){
    }
}
public class Sample {
    void add(int a){
    }
}
```

Quiz

```
package vit.demo;
public class Test {
   void method1(double a){
            System.out.println("double "+a);
   }
   void method1(float a){
            System.out.println("float "+a);
   }
   public static void main(String[] args) {
            Test test=new Test();
            test.method1(12);
   }
}
```

Quiz

```
package vit.demo;
public class Test {
   void method1(double a){
            System.out.println("double "+a);
   }
   void method1(float a){
            System.out.println("float "+a);
   }
   public static void main(String[] args) {
            Test test=new Test();
            test.method1(12.45);
   }
}
```

Quiz

```
package vit.demo;
public class Test {
   void method1(int a){
            System.out.println("int "+a);
   }
   void method1(byte a){
            System.out.println(“byte "+a);
   }
   public static void main(String[] args) {
            Test test=new Test();
            test.method1(2000.50f);
   }
}
```

Quiz

```
package vit.demo;
public class Test {
   void method1(int a){
            System.out.println("int "+a);
   }
   void method1(byte a){
            System.out.println(“byte "+a);
   }
   public static void main(String[] args) {
            Test test=new Test();
            test.method1(20);
   }
}
```

Quiz

```
package vit.demo;
public class Test {
   void method1(double a, float b){
            System.out.println("double "+a+" "+b);
   }
   void method1(float a, double b)        {
            System.out.println("float "+a+" "+b);
   }
   public static void main(String[] args) {
            Test test=new Test();
            test.method1(12,24);
   }
}
```

Quiz

```
package vit.demo;
public class Test {
   void method1(int... a){
            System.out.println("int "+a.length);
   }
   public static void main(String[] args) {
            Test test=new Test();
            test.method1(12);
            test.method1(12,24);
            test.method1(12,24,36);
   }
}
```

Method Overriding

- When a method in a subclass has the same prototype as a method in the superclass, then the method in the subclass is said to override the method in the superclass
- When an overridden method is called from an object of the subclass, it will always refer to the version defined by the subclass
- The version of the method defined by the superclass is hidden or overridden

Method Overriding

- A method in a subclass has the same prototype as a method in its superclass if it has the same name, type signature (the same type, sequence and number of parameters), and the same return type as the method in its superclass.

- In such a case, a method in a subclass is said to override a method in its superclass.

Rules for method overriding

Overriding method should satisfy the following points:

- They must have same argument list
- They must have same return type
- They must not have a more restrictive access modifier
- They may have a less restrictive access modifier
- Must not throw new or broader checked exceptions
- May throw fewer or narrower checked exceptions or any unchecked exceptions.
- Final methods cannot be overridden.
- Constructors cannot be overridden

Example

```
package vit.demo;
class Person{
   void display(){
            System.out.println("i am person");
   }
}
class Employee extends Person{
   void display(){
            System.out.println("i am employee");
   }
}
```

```
class Main {
   public static void main(String ar[]) {
            Employee emp=new Employee();
            emp.display();
   }
}
```

Using super to Call an Overridden Method

```
package vit.demo;
class Person{
   void display(){
            System.out.println("i am person");
   }
}
class Employee extends Person{
   void display(){
            super.display();
            System.out.println("i am employee");
   }
}
```

```
class Main {
   public static void main(String ar[]) {
            Employee emp=new Employee();
            emp.display();
   }
}
```

Guess the output

```
package vit.demo;
class Person{
   void display(){
            System.out.println("i am person");
   }
}
class Employee extends Person{
   int display(){
            return 100;
   }
}
```

```
class Main {
   public static void main(String ar[]) {
            Employee emp=new Employee();
            emp.display();
   }
}
```

Why Overridden Methods? A Design Perspective

- Overridden methods in a class hierarchy is one of the ways that Java implements the "single interface, multiple implementations" aspect of polymorphism
- Part of the key to successfully applying polymorphism is understanding the fact that the super classes and subclasses form a hierarchy which moves from lesser to greater specialization
- The superclass provides all elements that a subclass can use directly
- It also declares those methods that the subclass must implement on its own

Dynamic Method Dispatch or Runtime Polymorphism

- Method overriding forms the basis of one of Java's most powerful concepts: **dynamic method dispatch**

- Dynamic method dispatch occurs when the Java language resolves a call to an overridden method at runtime, and, in turn, implements runtime polymorphism

- Java makes runtime polymorphism possible in a class hierarchy with the help of two of its features:
  - superclass reference variables
  - overridden methods

Dynamic Method Dispatch or Runtime Polymorphism

- A superclass reference variable can hold a reference to a subclass object
- Java uses this fact to resolve calls to overridden methods at runtime
- When an overridden method is called through a superclass reference, Java determines which version of the method to call based upon the type of the object being referred to at the time the call occurs

Example

```
package vit.demo;
class Shape{
   int height;
   int width;
   Shape(int x, int y){
            height=x;
            width=y;
   }
   int area(){
            return 0;
   }
}
```

```
class Rectangle extends Shape{
   Rectangle(int x, int y){
            super(x,y);
   }
   int area(){
            return height*width;
   }
}
```

Example

```
class Triangle extends Shape{
   Triangle(int x, int y){
            super(x,y);
   }
   int area(){
            return height*width/2;
   }
}
```

```
class Main {
   public static void main(String ar[]) {
            Shape shape=new Shape(10, 20);
            Rectangle rectangle=new Rectangle(2, 3);
            Triangle triangle=new Triangle(5, 10);

            Shape s; //reference variable
            s=rectangle;
            System.out.println(s.area());
            s=triangle;
            System.out.println(s.area());
   }
}
```

instance of operator

- The instanceof operator in java allows you determine the type of an object
- The instanceof operator compares an object with a specified type
- It takes an object and a type and returns true if object belongs to that type. It returns false otherwise.
- We use instanceof operator to test if an object is an instance of a class, an instance of a subclass, or an instance of a class that implements an interface

Example

```
package vit.demo;
class A {
    int i, j;
}
class B extends A {
    int a, b;
}
class C extends A {
    int m, n;
}
class Main {
    public static void main(String ar[]) {
            A a = new A();
            B b = new B();
            C c = new C();
            A ob = b;
            if (ob instanceof B)
                    System.out.println("ob refers to B");
            else
                    System.out.println("ob NOT refers to B");
            if (ob instanceof A)
                    System.out.println("ob refers to A");
            else
                    System.out.println("ob NOT refers to A");
            if (ob instanceof C)
                    System.out.println("ob refers to C");
            else
                    System.out.println("ob NOT refers to C");
    }
}
```

Example

```
package vit.demo;
class A {
   int i, j;
}
class B extends A {
   int a, b;
}
class C extends A {
   int m, n;
}
class Main {
   public static void main(String ar[]) {
            A a = new A();
            B b = new B();
            C c = new C();
            A ob = c;
            if (ob instanceof B)
                        System.out.println("ob refers to B");
            else
                        System.out.println("ob NOT refers to B");
            if (ob instanceof A)
                        System.out.println("ob refers to A");
            else
                        System.out.println("ob NOT refers to A");
            if (ob instanceof C)
                        System.out.println("ob refers to C");
            else
                        System.out.println("ob NOT refers to C");
   }
}
```

Summary

We have discussed about

- Describe method overloading
- Describe method overriding
- Describe dynamic method dispatch, or runtime polymorphism
- Understand the use of instanceof operator