

Unit 4

Inheritance and Package

Java Inheritance

- ❖ Inheritance is one of the key features of OOP that allows us to create a new class from an existing class.
- ❖ The new class that is created is known as **subclass** (child or derived class) and the existing class from where the child class is derived is known as **superclass** (parent or base class).
- ❖ For inheritance in Java, extends keyword is used.
- ❖ We create an object of child class in inheritance

```
class ParentClass{  
    //fields and methods of ParentClass  
}  
  
class ChildClass extends ParentClass{  
    // fields and methods of ParentClass  
    // fields and methods of ChildClass  
}
```

Java Inheritance

- ❖ In Java, inheritance is an is-a relationship.
- ❖ That is, we use inheritance only if there exists an is-a relationship between two classes. For example,
 - Bike is a Vehicle
 - Apple is a Fruit
- ❖ Here, Bike can inherit from Vehicle, Apple can inherit from Fruit
- ❖ Note: **Generally we use protected access specifier for members that needs to be inherited.**

Java Inheritance-Example

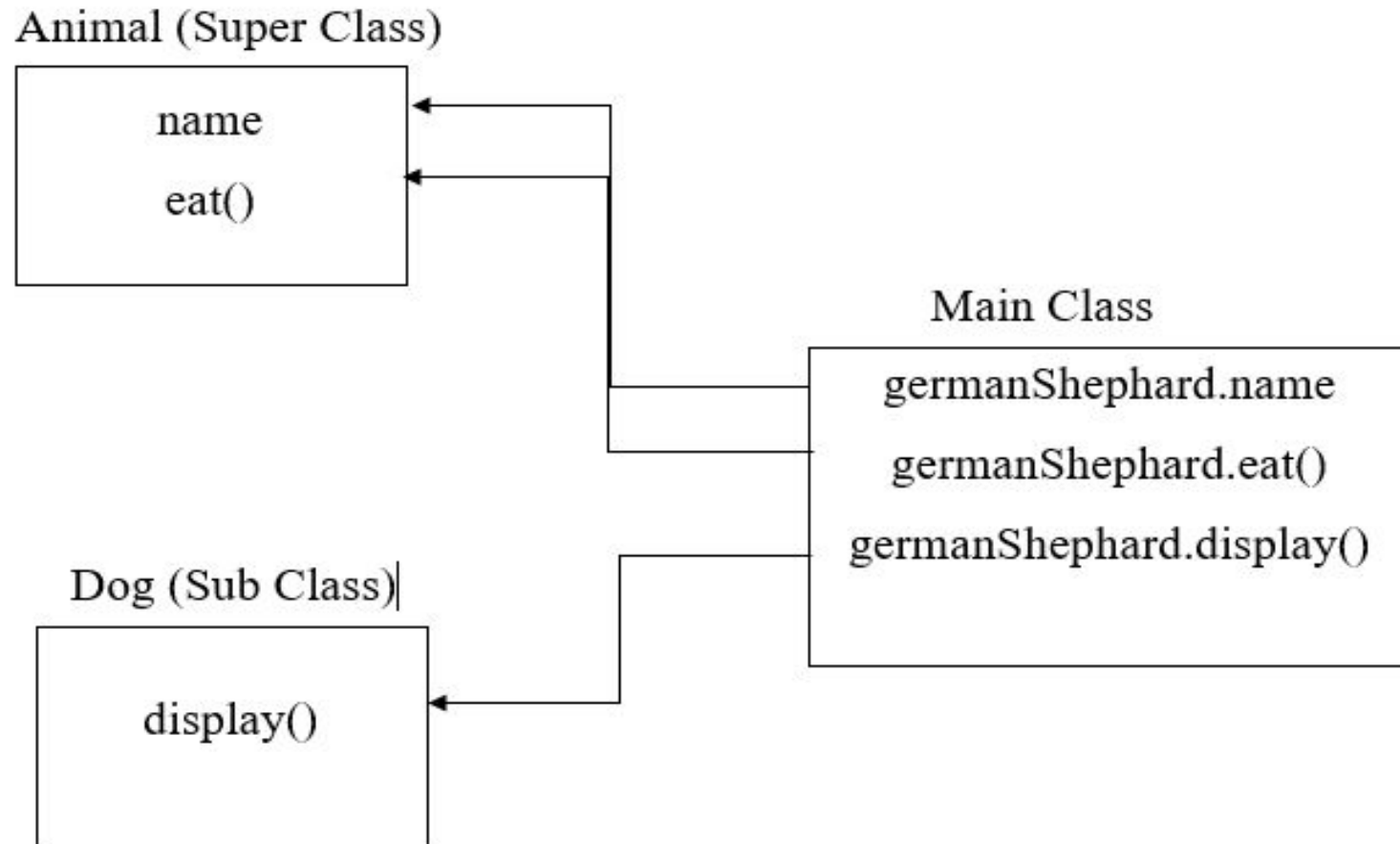
```
public class Inheritance {  
  
    public static void main(String[] args) {  
        Dog germanShephard = new Dog(); // cerating instance of child class  
        germanShephard.name="German Shephard 123"; //using field of parent class  
        germanShephard.eat(); //calling method of parent class  
        germanShephard.display();  
    }  
}
```

```
public class Animal {  
    String name;  
    public void eat() {  
        System.out.println("I am eating");  
    }  
}
```

```
public class Dog extends Animal{  
    public void display() {  
        System.out.println("My Name is " + name);  
    }  
}
```

```
Output - Inheritance (run) ×  
run:  
I am eating  
My Name is German Shephard 123  
BUILD SUCCESSFUL (total time: 0 seconds)
```

Java Inheritance-Example



Access Specifier For Java Inheritance

- ❖ If we need to inherit any field on child class then that method must be declared **protected**.

Multi-level Inheritance Example In Java

```
package javaapplication44;

public class Employee {
    protected int id;
    protected String name;
    public void displayEmployee() {
        System.out.println("I am an Employee of the organization");
    }
}

public class Manager extends Employee{
    protected String managerType;
    public void displayManager(){
        System.out.println("I am manager of the organization");
    }
}

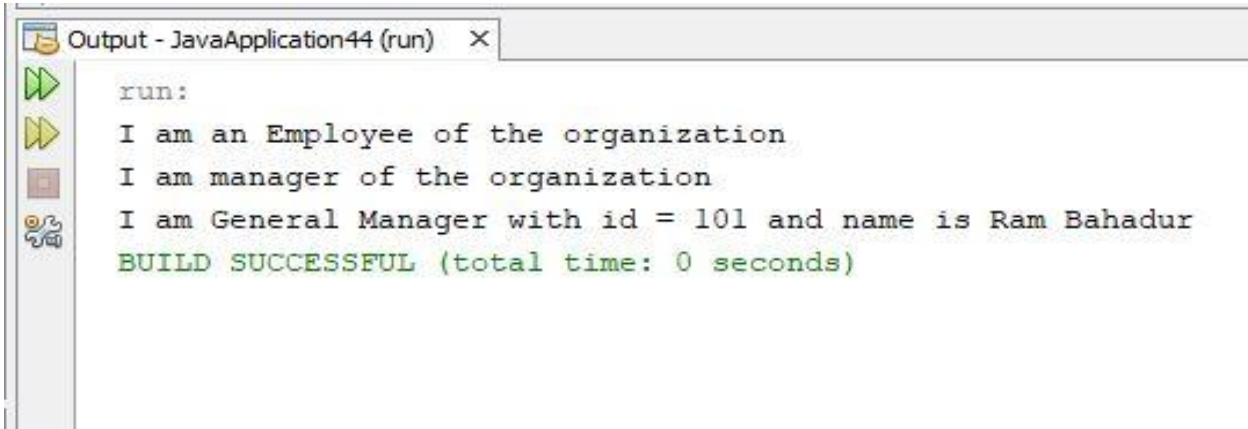
public class GMManger extends Manager{

    public void setValues(int id, String name, String managerType){
        this.id=id;
        this.name=name;
        this.managerType=managerType;
    }

    public void displayGM(){
        System.out.println("I am "+ managerType+" with id = "+id+" and name is "+name);
    }
}
```

Multi-level Inheritance Example In Java

```
public class JavaApplication44 {  
  
    public static void main(String[] args) {  
        GMManger gmm = new GMManger();  
        gmm.setValues(101, "Ram Bahadur", "General Manager");  
        gmm.displayEmployee();  
        gmm.displayManager();  
        gmm.displayGM();  
    }  
}
```



The screenshot shows an IDE output window titled "Output - JavaApplication44 (run)". It contains the following text:

```
run:  
I am an Employee of the organization  
I am manager of the organization  
I am General Manager with id = 101 and name is Ram Bahadur  
BUILD SUCCESSFUL (total time: 0 seconds)
```

The output window includes standard IDE icons on the left: a green play button, a yellow play button, a red stop button, and a magnifying glass icon.

Construction Invocation Order in Inheritance (Multilevel Inheritance)

```
package javaapplication44;
public class Employee {

    public Employee(){
        System.out.println("Default constructor of Employee class");
    }
}

package javaapplication44;
public class Employee {

    public Employee(){
        System.out.println("Default constructor of Employee class");
    }
}

public class GMManger extends Manager{
    public GMManger(){
        System.out.println("Default Constructor of GMMnager ");
    }
}

public class JavaApplication44 {

    public static void main(String[] args) {
        GMManger gmm = new GMManger();
    }
}
```



Output - JavaApplication44 (run) X

run:

Default constructor of Employee class

Default constructor of Manager Class

Default Constructor of GMMnager

BUILD SUCCESSFUL (total time: 0 seconds)

Method Overriding in Java Inheritance

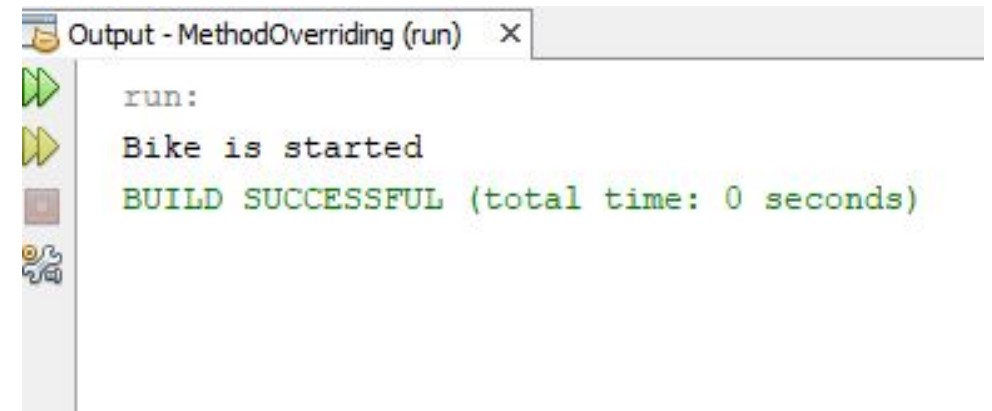
- ❖ If the same method is present in both the superclass and subclass, the method in the subclass overrides the method in the superclass
- ❖ We can use the `@Override` annotation to tell the compiler that we are overriding a method. However, the annotation is not mandatory.
- ❖ Java Overriding Rules:
 - Both the superclass and the subclass must have the **same method name**, the **same return type** and the **same parameter list**.
 - We cannot override the method declared as final and static.
 - We should always override abstract methods of the superclass

Method Overriding in Java Inheritance-Example

```
public class Vehicle {  
    public void start(){  
        System.out.println("Vehicle is started");  
    }  
}
```

```
public class Bike extends Vehicle{  
  
    @Override  
    public void start(){  
        System.out.println("Bike is started");  
    }  
}
```

```
public class MethodOverriding {  
  
    public static void main(String[] args) {  
        Bike bike = new Bike();  
        bike.start();  
    }  
}
```

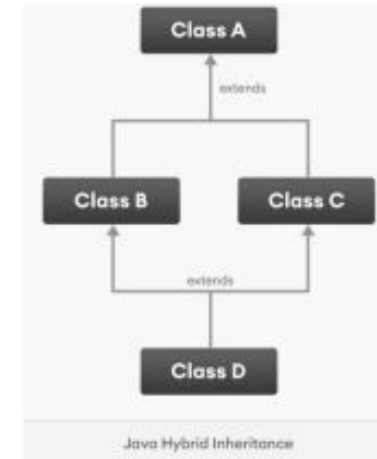
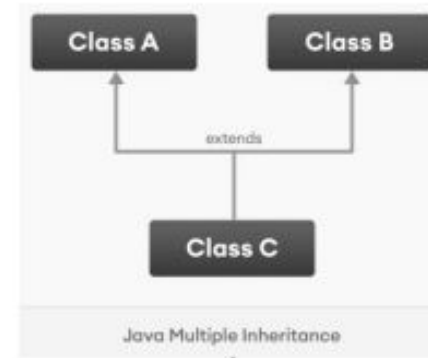
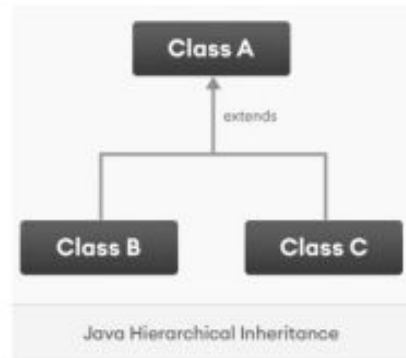
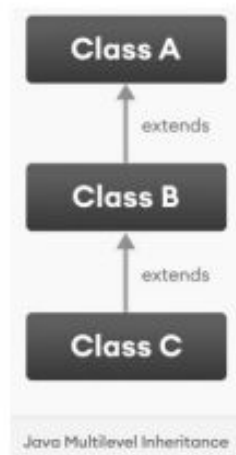


```
Output - MethodOverriding (run) X  
run:  
Bike is started  
BUILD SUCCESSFUL (total time: 0 seconds)
```

Access Specifiers in Method Overriding

- ❖ The same method declared in the superclass and its subclasses can have different access specifiers. However, there is a restriction.
- ❖ We can only use those access specifiers in subclasses that provide larger access than the access specifier of the superclass.
- ❖ Suppose, a method `myClass()` in the superclass is declared protected.
- ❖ Then, the same method `myClass()` in the subclass can be either public or protected, but not private

Types of Inheritance



Java doesn't support multiple inheritance. However, we can achieve multiple inheritance using interfaces

Super Keyword in java

- ❖ To access the method of superclass from the sub class we use the super keyword.

```
class Vehicle {  
    public void start(){  
        System.out.println("Starting Vehicle!");  
    }  
}  
  
// inherit from Vehicle  
class Bike extends Vehicle {  
    @Override  
    public void start(){  
        super.start();  
        System.out.println("Starting Bike!");  
    }  
}  
  
class Main {  
    public static void main(String[] args) {  
        Bike bike = new Bike();  
        bike.start();  
    }  
}
```

```
Starting Vehicle!  
Starting Bike!
```

← Output

Super Keyword

- ❖ It is important to note that constructors in Java are not inherited.
- ❖ Hence, there is no such thing as constructor overriding in Java.
- ❖ **The parent class constructor is always called before the subclass constructor**
- ❖ However, we can call the constructor of the superclass from its subclasses using super keyword which is required **if superclass has parameterized constructor**

Super Keyword

```
class Vehicle {
    String color;
    Vehicle(String color){
        this.color = color;
    }
    public String getColor(){
        return this.color;
    }
}

class Bike extends Vehicle {
    Bike(String color){
        super(color);
        System.out.println("Bike created!");
    }
    public void start(){
        System.out.println("VROOM!");
    }
}

class Main {
    public static void main(String[] args) {
        Bike bike = new Bike("Red");
        bike.start();
        System.out.println("Bike color: "+bike.getColor());
    }
}
```

```
Bike created!
VROOM!
Bike color: Red
```

← Output

Final Keyword

❖ Java final Variable

- In Java, we cannot change the value of a final variable.
- It is similar to creating constants in C Java

❖ Java final Method

- In Java, the final method cannot be overridden by the child class
Java

❖ Java final Class

- In Java, the final class cannot be inherited by another class

Object Class In Java

- ❖ Object class is present in **java.lang package** and class acts as a root of inheritance hierarchy in any Java Program
- ❖ Every class in Java is directly or indirectly derived from the Object class.
- ❖ If a Class does not extend any other class then it is direct child class of Object and if extends other class then it is an indirectly derived.
- ❖ Therefore the Object class methods are available to all Java classes.

Some Methods In Object Class

1. toString()

- ❖ It provides string representation of an Object
- ❖ The default toString() method for class Object returns a string consisting of **the name of the class of which the object is an instance, the at-sign character '@', and the unsigned hexadecimal representation of the hash code of the object**
- ❖ It is always recommended to override toString() method to get our own String representation of Object

```
class Bike {  
  
}  
  
class Main {  
    public static void main(String[] args) {  
        Bike bike = new Bike();  
        System.out.println( bike.toString());  
    }  
}
```

Bike@5acf9800

Output

Some Methods In Object Class

2. hashCode()

- ❖ For every object, JVM generates a unique number which is hashCode.
- ❖ It returns distinct integers for distinct objects.
- ❖ **A common misconception about this method is that hashCode() method returns the address of object, which is not correct.**
- ❖ It convert the internal address of object to an integer by using an algorithm
- ❖ Override of hashCode() method needs to be done such that for every object we generate a unique number.
- ❖ For example, for a Student class we can return roll no. of student from hashCode() method as it is unique.

```
class Bike {  
  
}  
  
class Main {  
    public static void main(String[] args) {  
        Bike bike = new Bike();  
        System.out.println( bike.hashCode());  
    }  
}
```

1523554304

Output

Some Methods In Object Class

3. equals(Object Obj)

- ❖ Compares the given object to “this” object (the object on which the method is called).
- ❖ It gives a generic way to compare objects for equality.
- ❖ It is recommended to override equals(Object obj) method to get our own equality condition on Objects.
- ❖ It is generally necessary to override the hashCode() method whenever this method is overridden, so as to maintain the general contract for the hashCode method, which states that equal objects must have equal hash codes.

```
class Main {  
    public static void main(String[] args) {  
        Bike bike1 = new Bike();  
        Bike bike2 = new Bike();  
  
        System.out.println(bike1.equals(bike2));  
  
        String college1 = "Himalaya";  
        String college2 = "Himalaya";  
  
        System.out.println(college1.equals(college2));  
    }  
}
```

false
true
↑
Output

Some Methods In Object Class

4. getClass()

- ❖ Returns the class object of “this” object and used to get actual runtime class of the object
- ❖ As it is final so we don't override it.

5. finalize()

- ❖ This method is called just before an object is garbage collected.
- ❖ It is called by the Garbage Collector on an object when garbage collector determines that there are no more references to the object.
- ❖ We should override finalize() method to dispose system resources, perform clean-up activities and minimize memory leaks

```
class Main {  
    public static void main(String[] args) {  
        Bike bike1 = new Bike();  
        Bike bike2 = new Bike();  
  
        System.out.println(bike1.equals(bike2));  
  
        String college1 = "Himalaya";  
        String college2 = "Himalaya";  
  
        System.out.println(college1.equals(college2));  
    }  
}
```

false
true
↑
Output

Some Methods In Object Class

6. clone()
 - It returns a new object that is exactly the same as this object
7. The remaining three methods wait(), notify() notifyAll() are related to Concurrency

Java Abstract Class and Method-Example (Theory already discussed in chapter 3)

```
abstract class Bike{
    Bike(){
        System.out.println("bike is created");
    }
    //abstract method
    abstract void run();
    // regular method
    void changeGear(){
        System.out.println("gear changed");
    }
}

//Creating a Child class which inherits Abstract class
class Honda extends Bike{
    @Override
    void run(){
        System.out.println("running safely..");
    }
}
```

```
class Main{
    public static void main(String args[]){
        Bike obj = new Honda();
        obj.run();
        obj.changeGear();
    }
}
```

bike is created
running safely..
gear changed

Output

Java Interface-Example (Theory already discussed in chapter 3)

```
interface Bike{
    void burnFuel();
}

class Yamaha implements Bike{
    @Override
    public void burnFuel() {
        System.out.println("FI system burns Fuel");
    }
}

class Bajaj implements Bike{
    @Override
    public void burnFuel() {
        System.out.println("Carburator system burns Fuel");
    }
}
```

```
class Main{
    public static void main(String args[]){
        Yamaha r15 = new Yamaha();
        Bajaj ns = new Bajaj();

        r15.burnFuel();
        ns.burnFuel();
    }
}
```

FI system burns Fuel
Carburator system burns Fuel

Output

Java Interface- Implementing Multiple Interfaces

❖ In Java, a class can also implement multiple Interfaces

```
interface A {  
    // members of A  
}  
  
interface B {  
    // members of B  
}  
  
class C implements A, B {  
    // abstract members of A  
    // abstract members of B  
}
```

Java Packages

- ❖ A java package is a group of similar types of classes, interfaces and sub-packages
- ❖ Advantage of Java Package
 - Java package is used to categorize the classes and interfaces so that they can be easily maintained.
 - Java package provides access protection.
 - Java package removes naming collision.
- ❖ Categories of Packages
 - Built-in Packages (packages from the Java API)
 - User-defined Packages (create your own packages)

Java Packages-Built-in Packages

- ❖ The Java API is a library of prewritten classes, that are free to use, included in the Java Development Environment.
- ❖ The library contains components for managing input, database programming, and much more
- ❖ The library is divided into packages and classes. Meaning we can either import a single class (along with its methods and attributes), or a whole package that contain all the classes that belong to the specified package.

```
import package.name.Class;    // Import a single class
import package.name.*;       // Import the whole package
```

- ❖ Example: `import java.util.Scanner;`

In the example above, java.util is a package, while Scanner is a class of the java.util package

- ❖ Example: `import java.util.*;` will import all the classes in the java.util package

Java Packages- User-Defined Packages

- ❖ To create a package, use the package keyword
- ❖ While creating a project, we can also define package name.

Java Polymorphism

- ❖ The word polymorphism is a combination of two words i.e. **poly** and **morphs**.
- ❖ The word poly means **many** and morphs means **different forms**.
- ❖ In short, a mechanism by which we can perform a single action in different ways.
- ❖ Real life example
 - A person in a shop is a customer, in an office, he is an employee, in the home he is husband/ father/son, in a party he is guest. So, the same person possesses different roles in different places. It is called polymorphism.

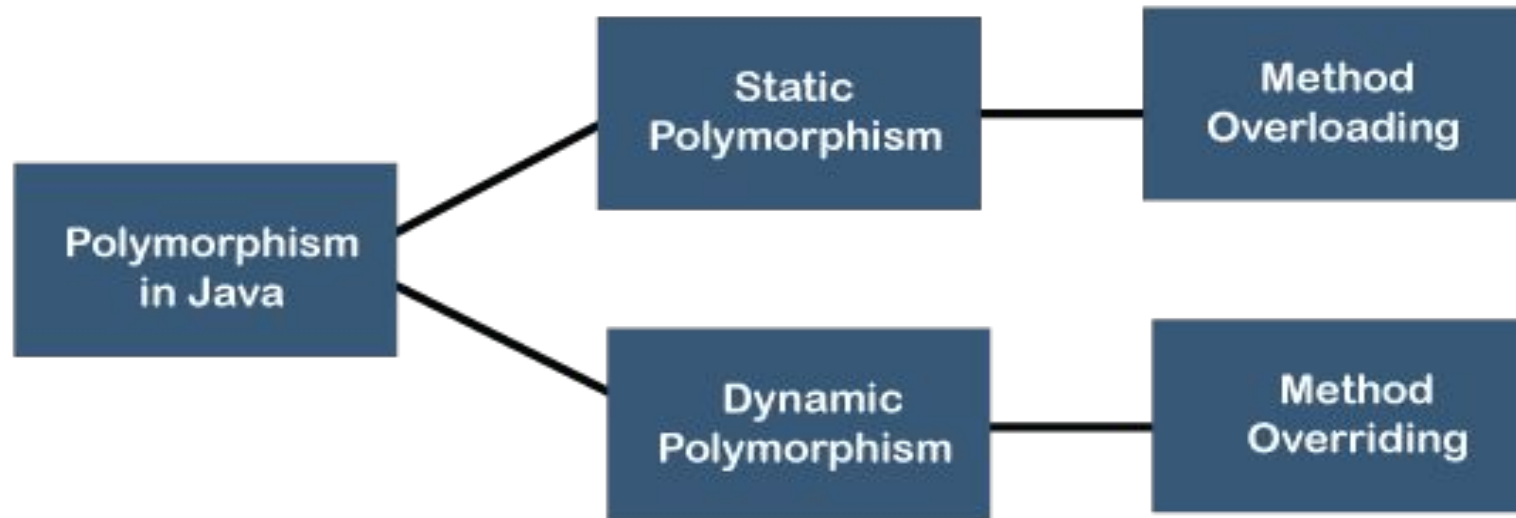
Java Polymorphism

❖ We can achieve polymorphism in Java using the following ways:

1. Method Overriding
2. Method Overloading
3. Operator Overloading
 - Some operators in Java behave differently with different operands.
 - For example, + operator is overloaded to perform numeric addition as well as string concatenation
 - In languages like C++, we can define operators to work differently for different operands.
 - However, Java doesn't support user-defined operator overloading.

Java Polymorphism

- ❖ There are two types of polymorphism in Java:
 - Static Polymorphism (Compile Time Polymorphism): can be achieved using method overloading
 - Dynamic Polymorphism (Run Time Polymorphism): can be achieved using method overriding



Java Polymorphism Example

```
class Vehicle{
    public void wheels(){};
}

class Bike extends Vehicle{
    @Override
    public void wheels(){
        System.out.println("Vehicle BIKE has two wheels");
    }
}

class Car extends Vehicle{
    @Override
    public void wheels(){
        System.out.println("Vehicle CAR has two wheels");
    }
}
```

```
class Main{
    public static void main(String[] args){
        Bike honda = new Bike();
        Car bmw = new Car();

        Vehicle[] vehicles = {honda,bmw};
        for(Vehicle vehicle:vehicles){
            vehicle.wheels();
        }
    }
}
```

Vehicle behaves as bike

Vehicle behaves as car

Vehicle BIKE has two wheels
Vehicle CAR has two wheels

Output

Dynamic Polymorphism

- ❖ Dynamic Polymorphism in OOPs is the mechanism by which multiple methods can be defined with same name and signature in the superclass and subclass.
- ❖ The call to an overridden method are resolved at run time.
- ❖ We can create a **reference variable of super class** that **refer to a sub class object** as

SuperClassName referenceVariable = new SubClassName();

Example: Vehicle vehicle = new Bike();

where Vehicle is super class and Bike is sub class

Dynamic Polymorphism-Example

```
public class Vehicle {  
    public void wheels() {  
        System.out.println("Vehicles have wheel");  
    }  
}
```

```
public class Bike extends Vehicle {  
    @Override  
    public void wheels() {  
        System.out.println("Vehicle BIKE have 2 wheels");  
    }  
}
```

```
public class Car extends Vehicle {  
    @Override  
    public void wheels() {  
        System.out.println("Vehicle CAR have 4 wheels");  
    }  
}
```

```
import java.util.Scanner;  
public class Dynamicpolymorphism {  
    public static void main(String[] args) {  
        Vehicle vehicle;  
        System.out.println("Enter 1 for Bike and 2 for Car");  
        Scanner sc = new Scanner(System.in);  
        int choice = sc.nextInt();  
        if(choice == 1){  
            vehicle = new Bike();  
        }else if(choice == 2){  
            vehicle = new Car();  
        }else{  
            vehicle = new Vehicle();  
        }  
        vehicle.wheels();  
    }  
}
```

Dynamic Polymorphism-Example

Output of Earlier Program

```
Output - Dynamicpolymorphism (run) X
run:
Enter 1 for Bike and 2 for Car
1
Vehicle BIKE have 2 wheels
BUILD SUCCESSFUL (total time: 3 seconds)
```

```
Output - Dynamicpolymorphism (run) X
run:
Enter 1 for Bike and 2 for Car
2
Vehicle CAR have 4 wheels
BUILD SUCCESSFUL (total time: 2 seconds)
```

```
Output - Dynamicpolymorphism (run) X
run:
Enter 1 for Bike and 2 for Car
3
Vehicles have wheel
BUILD SUCCESSFUL (total time: 2 seconds)
```

Binding in Java

- ❖ Binding is a mechanism creating link between method call and method actual implementation.
- ❖ As per the polymorphism concept in Java , object can have many different forms.
- ❖ Object forms can be resolved at compile time and run time.
- ❖ If linking between method call and method implementation is resolved at compile time then we call it **static binding/ early binding**
- ❖ If linking between method call and method implementation is resolved at run time then it **dynamic binding/late binding**.