Polymorphism in Java

**Polymorphism in Java** is a concept by which we can perform a *single action in different ways*. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.

If you overload a static method in Java, it is the example of compile time polymorphism. Here, we will focus on runtime polymorphism in java.

Runtime Polymorphism in Java

**Runtime polymorphism** or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

Skip Ad

In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

Let's first understand the upcasting before Runtime Polymorphism.

Upcasting

If the reference variable of Parent class refers to the object of Child class, it is known as upcasting. For example:



1. **class** A{}
2. **class** B **extends** A{}
3. A a=**new** B();//upcasting

For upcasting, we can use the reference variable of class type or an interface type. For Example:

1. **interface** I{}
2. **class** A{}
3. **class** B **extends** A **implements** I{}

Here, the relationship of B class would be:

B IS-A A

B IS-A I

B IS-A Object

Since Object is the root class of all classes in Java, so we can write B IS-A Object.

Example of Java Runtime Polymorphism

In this example, we are creating two classes Bike and Splendor. Splendor class extends Bike class and overrides its run() method. We are calling the run method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, the subclass method is invoked at runtime.

Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

1. **class** Bike{
2. **void** run(){System.out.println("running");}
3. }
4. **class** Splendor **extends** Bike{
5. **void** run(){System.out.println("running safely with 60km");}
7. **public** **static** **void** main(String args[]){
8. Bike b = **new** Splendor();//upcasting
9. b.run();
10. }
11. }

Output:

running safely with 60km.

Java Runtime Polymorphism Example: Bank

Consider a scenario where Bank is a class that provides a method to get the rate of interest. However, the rate of interest may differ according to banks. For example, SBI, ICICI, and AXIS banks are providing 8.4%, 7.3%, and 9.7% rate of interest.



Note: This example is also given in method overriding but there was no upcasting.

1. **class** Bank{
2. **float** getRateOfInterest(){**return** 0;}
3. }
4. **class** SBI **extends** Bank{
5. **float** getRateOfInterest(){**return** 8.4f;}
6. }
7. **class** ICICI **extends** Bank{
8. **float** getRateOfInterest(){**return** 7.3f;}
9. }
10. **class** AXIS **extends** Bank{
11. **float** getRateOfInterest(){**return** 9.7f;}
12. }
13. **class** TestPolymorphism{
14. **public** **static** **void** main(String args[]){
15. Bank b;
16. b=**new** SBI();
17. System.out.println("SBI Rate of Interest: "+b.getRateOfInterest());
18. b=**new** ICICI();
19. System.out.println("ICICI Rate of Interest: "+b.getRateOfInterest());
20. b=**new** AXIS();
21. System.out.println("AXIS Rate of Interest: "+b.getRateOfInterest());
22. }
23. }

Output:

SBI Rate of Interest: 8.4

ICICI Rate of Interest: 7.3

AXIS Rate of Interest: 9.7

Java Runtime Polymorphism Example: Shape

1. **class** Shape{
2. **void** draw(){System.out.println("drawing...");}
3. }
4. **class** Rectangle **extends** Shape{
5. **void** draw(){System.out.println("drawing rectangle...");}
6. }
7. **class** Circle **extends** Shape{
8. **void** draw(){System.out.println("drawing circle...");}
9. }
10. **class** Triangle **extends** Shape{
11. **void** draw(){System.out.println("drawing triangle...");}
12. }
13. **class** TestPolymorphism2{
14. **public** **static** **void** main(String args[]){
15. Shape s;
16. s=**new** Rectangle();
17. s.draw();
18. s=**new** Circle();
19. s.draw();
20. s=**new** Triangle();
21. s.draw();
22. }
23. }

Output:

drawing rectangle...

drawing circle...

drawing triangle...

Java Runtime Polymorphism Example: Animal

1. **class** Animal{
2. **void** eat(){System.out.println("eating...");}
3. }
4. **class** Dog **extends** Animal{
5. **void** eat(){System.out.println("eating bread...");}
6. }
7. **class** Cat **extends** Animal{
8. **void** eat(){System.out.println("eating rat...");}
9. }
10. **class** Lion **extends** Animal{
11. **void** eat(){System.out.println("eating meat...");}
12. }
13. **class** TestPolymorphism3{
14. **public** **static** **void** main(String[] args){
15. Animal a;
16. a=**new** Dog();
17. a.eat();
18. a=**new** Cat();
19. a.eat();
20. a=**new** Lion();
21. a.eat();
22. }}

Output:

eating bread...

eating rat...

eating meat...

Java Runtime Polymorphism with Data Member

A method is overridden, not the data members, so runtime polymorphism can't be achieved by data members.

In the example given below, both the classes have a data member speedlimit. We are accessing the data member by the reference variable of Parent class which refers to the subclass object. Since we are accessing the data member which is not overridden, hence it will access the data member of the Parent class always.

Rule: Runtime polymorphism can't be achieved by data members.

1. **class** Bike{
2. **int** speedlimit=90;
3. }
4. **class** Honda3 **extends** Bike{
5. **int** speedlimit=150;
7. **public** **static** **void** main(String args[]){
8. Bike obj=**new** Honda3();
9. System.out.println(obj.speedlimit);//90
10. }

Output:

90

Java Runtime Polymorphism with Multilevel Inheritance

Let's see the simple example of Runtime Polymorphism with multilevel inheritance.

1. **class** Animal{
2. **void** eat(){System.out.println("eating");}
3. }
4. **class** Dog **extends** Animal{
5. **void** eat(){System.out.println("eating fruits");}
6. }
7. **class** BabyDog **extends** Dog{
8. **void** eat(){System.out.println("drinking milk");}
9. **public** **static** **void** main(String args[]){
10. Animal a1,a2,a3;
11. a1=**new** Animal();
12. a2=**new** Dog();
13. a3=**new** BabyDog();
14. a1.eat();
15. a2.eat();
16. a3.eat();
17. }
18. }

Output:

eating

eating fruits

drinking Milk

Try for Output

1. **class** Animal{
2. **void** eat(){System.out.println("animal is eating...");}
3. }
4. **class** Dog **extends** Animal{
5. **void** eat(){System.out.println("dog is eating...");}
6. }
7. **class** BabyDog1 **extends** Dog{
8. **public** **static** **void** main(String args[]){
9. Animal a=**new** BabyDog1();
10. a.eat();
11. }}

Output:

Dog is eating

Since, BabyDog is not overriding the eat() method, so eat() method of Dog class is invoked.