Polymorphism in Java

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.

Polymorphism in Java

Polymorphism in Java

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.

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.

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.

Runtime Polymorphism in Java

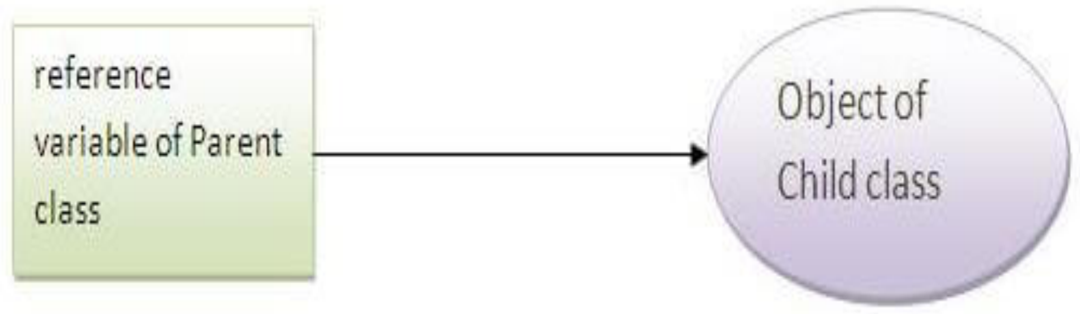
Runtime polymorphism or Dynamic Method Dispatch is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

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.

Polymorphism in Java

Upcasting :

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



Polymorphism in Java

Upcasting :

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

A a=new B();//upcasting
```

Polymorphism in Java

Example of Java Runtime Polymorphism

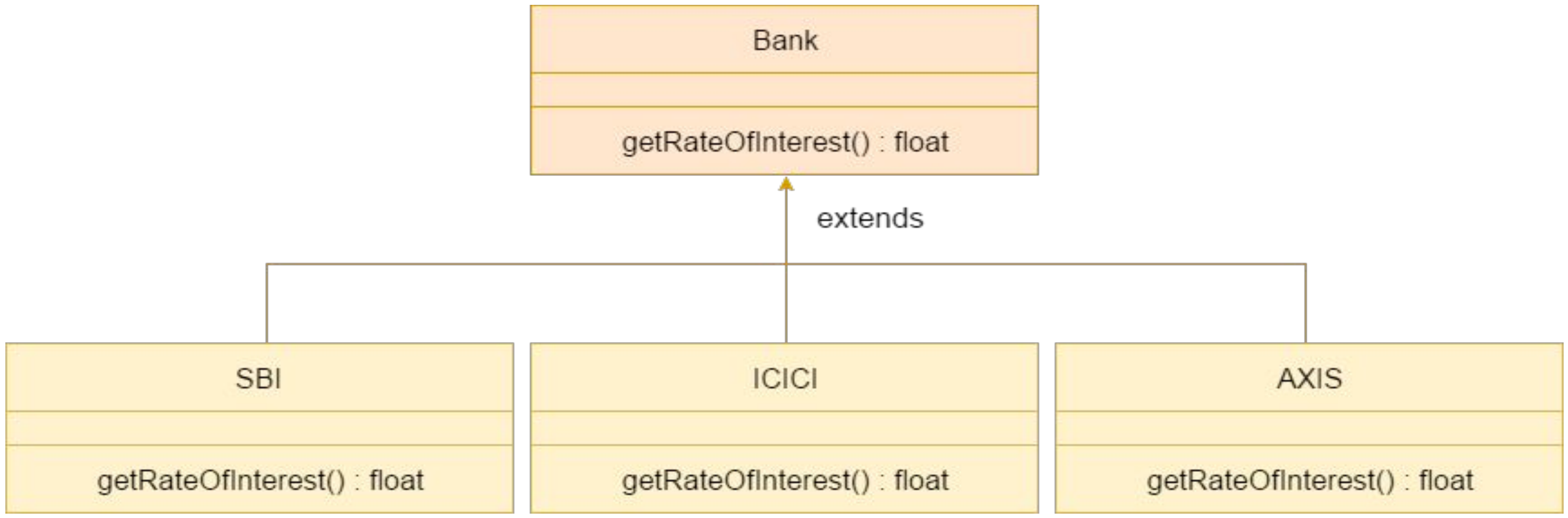
```
class Bike{
 void run(){System.out.println("running");}
}
class Splendor extends Bike{
 void run()
 {System.out.println("running safely with 60km");}

 public static void main(String args[]){
  Bike b = new Splendor();//upcasting
  b.run();
 }
}
```

Polymorphism in Java

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.


Bank
getRateOfInterest() : float
extends
SBI
getRateOfInterest() : float
ICICI
getRateOfInterest() : float
AXIS
getRateOfInterest() : float

Polymorphism in Java

Java Runtime Polymorphism Example: Bank

```
lass Bank{
float getRateOfInterest(){return 0;}
}
class SBI extends Bank{
float getRateOfInterest(){return 8.4f;}
}
class ICICI extends Bank{
float getRateOfInterest(){return 7.3f;}
}
class AXIS extends Bank{
float getRateOfInterest(){return 9.7f;}
}
```

Polymorphism in Java

Java Runtime Polymorphism Example: Bank

```
class TestPolymorphism{
public static void main(String args[]){
Bank b;
b=new SBI();
System.out.println("SBI Rate of Interest:
"+b.getRateOfInterest());
b=new ICICI();
System.out.println("ICICI Rate of Interest:
"+b.getRateOfInterest());
b=new AXIS();
System.out.println("AXIS Rate of Interest:
"+b.getRateOfInterest());
}}
```

Polymorphism in Java

Java Runtime Polymorphism Example: Shape

```
class Shape{
void draw(){System.out.println("drawing...");}
}
class Rectangle extends Shape{
void draw(){System.out.println("drawing rectangle...");}
}
class Circle extends Shape{
void draw(){System.out.println("drawing circle...");}
}
class Triangle extends Shape{
void draw(){System.out.println("drawing triangle...");}
}
```

Polymorphism in Java

Java Runtime Polymorphism Example: Shape

```
class TestPolymorphism
{
     public static void main(String args[])
     {
        Shape s;
        s=new Rectangle();
        s.draw();
        s=new Circle();
        s.draw();
        s=new Triangle();
        s.draw();
     }
}
```

Polymorphism in Java

Java Runtime Polymorphism Example: Animal

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void eat(){System.out.println("eating bread...");}
}
class Cat extends Animal{
void eat(){System.out.println("eating rat...");}
}
class Lion extends Animal{
void eat(){System.out.println("eating meat...");}
}
```

Polymorphism in Java

Java Runtime Polymorphism Example: Animal

```
class TestPolymorphism
{
        public static void main(String[] args)
        {
            Animal a;
            a=new Dog();
            a.eat();
            a=new Cat();
            a.eat();
            a=new Lion();
            a.eat();
         }
}
```

Polymorphism in Java

Java Runtime Polymorphism with Data Member

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

```
class Bike{
 int speedlimit=90;
}
class Honda extends Bike{
 int speedlimit=150;
 public static void main(String args[]){
  Bike obj=new Honda();
  System.out.println(obj.speedlimit);
}
```

Polymorphism in Java

Java Runtime Polymorphism with Multilevel Inheritance

```
class Animal
{
   void eat()
    {System.out.println("eating");}
}
class Dog extends Animal
{
   void eat()
    {System.out.println("eating fruits");}
}
```

Polymorphism in Java

Java Runtime Polymorphism with Multilevel Inheritance

```
class BabyDog extends Dog
{
   void eat()
    {System.out.println("drinking milk");}
   public static void main(String args[]){
   Animal a1,a2,a3;
   a1=new Animal();
   a2=new Dog();
   a3=new BabyDog();
   a1.eat();
   a2.eat();
   a3.eat();  }}
```

Polymorphism in Java

Java Runtime Polymorphism with Multilevel Inheritance

Output:

```
eating
eating fruits
drinking Milk
```

Polymorphism in Java

Java Runtime Polymorphism with Multilevel Inheritance

```
class Animal{
void eat(){System.out.println("animal is eating...");}
}
class Dog extends Animal{
void eat(){System.out.println("dog is eating...");}
}
class BabyDog extends Dog{
public static void main(String args[]){
Animal a=new BabyDog();
a.eat();
}}
```

Output:Dog is eating

Polymorphism in Java

Static Binding and Dynamic Binding

Connecting a method call to the method body is known as binding.

There are two types of binding

- **Static binding (also known as early binding).**
- **Dynamic binding (also known as late binding).**

Polymorphism in Java

Understanding Type :

Let's understand the type of instance.

1) variables have a type :

Each variable has a type, it may be primitive and non-primitive.

```
    int data=30;
```

Here data variable is a type of int.

Polymorphism in Java

Understanding Type :

2) References have a type :

```
class Dog{
 public static void main(String args[])
{

    Dog d1;//Here d1 is a type of Dog
 }
}
```

Polymorphism in Java

Understanding Type :

3) Objects have a type :

An object is an instance of particular java class,but it is also an instance of its superclass.

```
class Animal{}
class Dog extends Animal{
 public static void main(String args[]){
  Dog d1=new Dog();
 }
}
```

Here d1 is an instance of Dog class, but it is also an instance of Animal.

Polymorphism in Java

static binding :

When type of the object is determined at compiled time(by the compiler), it is known as static binding.

If there is any private, final or static method in a class, there is static binding.

```
class Dog{
 private void eat()
  {System.out.println("dog is eating...");}

 public static void main(String args[]){
  Dog d1=new Dog();
  d1.eat();  }}
```

Polymorphism in Java

Dynamic binding :

When type of the object is determined at run-time, it is known as dynamic binding.

```
class Animal{
 void eat()
 {System.out.println("animal is eating...");}
}
class Dog extends Animal{
 void eat(){System.out.println("dog is eating...");}

 public static void main(String args[]){
  Animal a=new Dog();
  a.eat();  }}
```

Polymorphism in Java

Dynamic binding :

In the previous example object type cannot be determined by the compiler, because the instance of Dog is also an instance of Animal.So compiler doesn't know its type, only its base type.

Polymorphism in Java

Java instanceof operator :

The java instanceof operator is used to test whether the object is an instance of the specified type (class or subclass or interface).

The instanceof in java is also known as type comparison operator because it compares the instance with type. It returns either true or false. If we apply the instanceof operator with any variable that has null value, it returns false.

Polymorphism in Java

Java instanceof operator :

```
class Simple
{
    public static void main(String args[])
    {
        Simple s=new Simple();
        System.out.println(s instanceof Simple);//true
    }
}
```

Polymorphism in Java

Java instanceof operator :

```
class Animal{}
class Dog extends Animal
{
  //Dog inherits Animal

    public static void main(String args[])
     {
        Dog d=new Dog();
        System.out.println(d instanceof Animal);//true
     }
}
```

Polymorphism in Java

instanceof in java with a variable that have null value

```
class Dog
{
    public static void main(String args[])
    {
        Dog d=null;
        System.out.println(d instanceof Dog);//false
    }
}
```

Polymorphism in Java

Downcasting with java instanceof operator :

When Subclass type refers to the object of Parent class, it is known as downcasting. If we perform it directly, compiler gives Compilation error. If you perform it by typecasting, ClassCastException is thrown at runtime. But if we use instanceof operator, downcasting is possible.

```
    Dog d=new Animal();//Compilation error
```

If we perform downcasting by typecasting, ClassCastException is thrown at runtime.

```
    Dog d=(Dog)new Animal();
    //Compiles successfully but ClassCastException is thrown at
runtime
```

Polymorphism in Java

Possibility of downcasting with instanceof :

```
class Animal { }
class Dog extends Animal
{
  static void method(Animal a) {
    if(a instanceof Dog){
      Dog d=(Dog)a;//downcasting
      System.out.println("ok downcasting performed");
    }
}
  public static void main (String [] args) {
    Animal a=new Dog();
    Dog.method(a);  }   }
```

Polymorphism in Java

Downcasting without the use of java instanceof

```
class Animal { }
class Dog extends Animal
{
       static void method(Animal a)
       {
             Dog d=(Dog)a;//downcasting
             System.out.println("ok downcasting performed");
        }
     public static void main (String [] args)
     {
        Animal a=new Dog();
        Dog.method(a);
      }}
```