Polymorphism

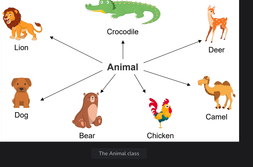
Get familiar with the concept of polymorphism and its types with implementation.

We'll cover the following

Introduction to polymorphism

The word **polymorphism** is a combination of two Greek words, "poly" meaning many, and "morph" meaning forms. In programming, polymorphism is a phenomenon that allows an object to have several different forms and behaviors.

For example, take the `Animal` class. There are many different animals, e.g., lion, deer, dog, and crocodile, etc. So, they are all animals, but their properties are different. The animal class can have a method, `makeNoise`. Its implementation should be different for a lion, deer, or any other animal as they all have different noises. This is called polymorphism.



The Animal class

Types of polymorphism

There are two types of polymorphism: dynamic polymorphism and static polymorphism, as shown in the figure below.

Types of polymorphism

Dynamic polymorphism

Dynamic polymorphism is the mechanism that defines the methods with the same name, return type, and parameters in the base class and derived classes. Hence, the call to an overridden method is decided at runtime. That is why dynamic polymorphism is also known as **runtime polymorphism**. It is achieved by method overriding.

Method overriding

In object-oriented programming, if a subclass provides a specific implementation of a method that had already been defined in one of its parent classes, it is known as **method overriding**.

Suppose we have a parent class, `Animal`, with its subclass, `Lion`. Below is the implementation of two functions with the same name in each class to check method overriding behavior.

```
class Animal {
    public void printAnimal() {
        System.out.print("I am from the Animal class\n");
    }
    void printAnimalTwo() {
        System.out.print("I am from the Animal class\n");
    }
}

class Lion extends Animal {
    // method overriding
    public void printAnimal() {
        System.out.print("I am from the Lion class\n");
    }
}

public class main {
    public static void main(String[] args) {
        Animal animal
        Lion lion = new Lion();
        animal = lion;

        animal.printAnimal();
        animal.printAnimalTwo();
    }
}
```

Method overriding example

Static polymorphism

Static polymorphism is also known as compile-time polymorphism, and it is achieved by method overloading or operator overloading.

Method overloading

Methods are said to be **overloaded** if a class has more than one method with the same name, but either the number of arguments is different, or the type of arguments is different. We have implemented method overloading using two functions with the same name but with different numbers of arguments. You can see this in the implementation below.

```
class Sum {
    public int addition(int a, int b) {
        return a + b;
    }

    public int addition(int a, int b, int c) {
        return a + b + c;
    }
}

public class main {
    public static void main(String[] args) {
        Sum sum = new Sum();
    System.out.print(sum.addition(14, 35));
    System.out.print("\n");
    System.out.print(sum.addition(31, 34, 43));
    }
}
```

Method overloading example

```
class Area {
    public double calculateArea(double length, double breadth) {
        return length * breadth;
    }

    public double calculateArea(double side) {
        return side * side;
    }
}

public class main {
    public static void main(String[] args) {
    Area area = new Area();
    System.out.print("Area of rectangle = " + area.calculateArea(3, 4));
    System.out.print("\n");
    System.out.print("Area of square = " + area.calculateArea(6));
    }
}
```

Method overloading example

Operator overloading

Operators can be overloaded to operate in a certain user-defined way. Its corresponding method is invoked to perform its predefined function whenever an operator is used. For example, when the `+` operator is called, it invokes the special function, `add`, but this operator acts differently for different data types. The `+` operator adds the numbers when it is used between two `int` data types and merges two strings when used between `string` data types.

Let's look at the implementation below, where we've overloaded the `+` operator to add complex numbers instead of simply adding two real numbers.

```
using System;

class ComplexNumber {
    float real;
    float imaginary;
    // Constructor
    public ComplexNumber(float real, float imaginary) {
      this.real = real;
      this.imaginary = imaginary;
    }

    public override string ToString() {
        return(String.Format("( {0} + {1} i )", real, imaginary));
    }

    // Overloading function for +
    public static ComplexNumber operator+(ComplexNumber c1, ComplexNumber c2) {
        return(new ComplexNumber(c1.real + c2.real, c1.imaginary + c2.imaginary));
    }
}

class MainClass {
    public static void Main() {
        ComplexNumber c1 = new ComplexNumber(11, 5);
        ComplexNumber c2 = new ComplexNumber(3, 6);
        // display results
        Console.WriteLine(c1 + c2);
    }
}
```

Operator overloading example

> **Note:** Java and JavaScript do not support operator overloading.

Dynamic polymorphism vs. static polymorphism

The table below provides a highlight of the differences between dynamic and static polymorphism:

Static Polymorphism	Dynamic Polymorphism
Polymorphism that is resolved during compile-time is known as static polymorphism.	Polymorphism that is resolved during runtime is known as dynamic polymorphism.
Method overloading is used in static polymorphism.	Method overriding is used in dynamic polymorphism.
Mostly used to increase the readability of the code.	Mostly used to have a separate implementation for a method if is already defined in the base class.
Arguments must be different in the case of overloading.	Arguments must be the same in the case of overriding.
Return type of the method does not matter.	Return type of the method must be the same.
Private and sealed methods can be overloaded.	Private and sealed methods cannot be overridden.
Gives better performance because the binding is being done at compile-time.	Gives worse performance because the binding is being done at runtime.

Let's test our OOP concepts in the next lesson.