POLYMORPHISM IN JAVA

Java Polymorphism

Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance. Let's us inherit attributes and methods from another class. **Polymorphism** uses those methods to perform different tasks. This allows us to perform a single action in different ways.

For example, think of a superclass called Animal that has a method called animalSound(). Subclasses of Animals could be Buffaloes, Cats, Dogs, Birds - And they also have their own implementation of an animal sound (the Bufallo mou, and the cat meows, etc.):

Example

```
class Animal {
  public void animalSound() {
    System.out.println("The animal makes a sound");
  }
}


class Bufallo extends Animal {
  public void animalSound() {
    System.out.println("The Bufallo says: mou mou");
  }
}


class Dog extends Animal {
  public void animalSound() {
    System.out.println("The dog says: bow wow");
  }
}
```

Remember from the Inheritance chapter that we use the extends keyword to inherit from a class.

Now we can create Bufallo and Dog objects and call the animalSound() method on both of them:

Example

```
class Animal {
  public void animalSound() {
    System.out.println("The animal makes a sound");
  }
}

class Bufallo extends Animal {
  public void animalSound() {
    System.out.println("The Bufallo says: mou mou");
  }
}

class Dog extends Animal {
  public void animalSound() {
    System.out.println("The dog says: bow wow");
  }
}

class Main {
  public static void main(String[] args) {
    Animal myAnimal = new Animal();  // Create a Animal object
    Animal myBufallo = new Bufallo();  // Create a Bufallo object
    Animal myDog = new Dog();  // Create a Dog object
    myAnimal.animalSound();
    myBufallo.animalSound();
```

```
    myDog.animalSound();
  }
}
```

OUTPUT

The animal makes a sound

The Bufallo says: mou mou

The dog says: bow wow

Why And When To Use "Inheritance" and "Polymorphism"?

- It is useful for code reusability: reuse attributes and methods of an existing class when you create a new class.

Here's another example of polymorphism in Java, this time using method overloading:

```
class Calculator {
    // Method overloading: multiple methods with the same name but different parameters
    int add(int a, int b) {
        return a + b;
    }

    double add(double a, double b) {
        return a + b;
    }
}

public class PolymorphismExample {
    public static void main(String[] args) {
        Calculator calculator = new Calculator();
 // Calling the add method with different parameter types
```

```
        int sum1 = calculator.add(5, 3); // Calls the int version of add method
        double sum2 = calculator.add(4.5, 2.5); // Calls the double version of add method

        System.out.println("Sum of integers: " + sum1); // Output: Sum of integers: 8
        System.out.println("Sum of doubles: " + sum2); // Output: Sum of doubles: 7.0
    }
}
```

In this example:

- We have a `Calculator` class with two overloaded `add` methods: one that takes two integers and one that takes two doubles.

- Both methods have the same name (`add`), but they have different parameter types.

- In the `main()` method, we create an instance of the `Calculator` class.

- We call the `add` method with different types of arguments (integers and doubles).

- At compile time, the compiler determines which version of the `add` method to call based on the parameter types passed to it. This is an example of compile-time polymorphism, or method overloading.

Polymorphism is a fundamental concept in object-oriented programming that allows objects of different classes to be treated as objects of a common superclass. It provides flexibility, extensibility, and simplification of code. However, like any programming concept, it has its advantages and disadvantages.

Advantages of Polymorphism:

1. **Code Reusability**: Polymorphism enables the reuse of code by allowing different classes to share a common interface or superclass. This reduces redundancy and promotes modular design.

2. **Flexibility**: Polymorphism allows for the creation of generic code that can operate on objects of different types. This flexibility enables easier adaptation to changing requirements and promotes code extensibility.

3. **Simplicity**: Polymorphism simplifies code by abstracting away specific implementation details. It promotes a high-level view of the system, making the code easier to understand and maintain.

4. **Enhanced Modularity**: Polymorphism encourages modular design by promoting loose coupling between classes. Changes to one part of the codebase are less likely to affect other parts, leading to better maintainability and scalability.

5. **Dynamic Binding**: Polymorphism enables dynamic method invocation, where the specific implementation of a method is determined at runtime based on the actual type of the object. This allows for more flexible and adaptable code.

Disadvantages of Polymorphism:

1. **Runtime Overhead**: Dynamic method invocation in polymorphism can incur some runtime overhead compared to static method invocation. This overhead may impact performance in time-critical applications.

2. **Complexity**: Polymorphism, especially when used extensively, can introduce complexity to the codebase. Understanding the behavior of polymorphic code may require additional effort, especially for developers unfamiliar with the code.

3. **Potential Pitfalls**: Inheritance hierarchies that rely heavily on polymorphism may lead to fragile base class problems or the violation of the Liskov Substitution Principle (LSP). Incorrect use of polymorphism can result in unexpected behavior or bugs.

4. **Debugging Challenges**: Polymorphic code may be more challenging to debug, especially when dealing with complex inheritance hierarchies or overridden methods. Tracking the flow of execution and understanding which method is invoked at runtime can be challenging.

5. **Performance Considerations**: While polymorphism can enhance code flexibility, it may not always be the most performant solution, especially in performance-critical applications. Careful consideration is needed to balance flexibility with performance requirements.

Conclusion

In summary, while polymorphism offers numerous advantages such as code reusability, flexibility, and simplicity, it also comes with potential drawbacks such as runtime overhead, complexity, and debugging challenges. Like any programming concept, it should be used judiciously, considering the specific requirements and constraints of the project.

Practice Tasks

1. Create a **Parent** class as the base class with a method **displayParentInfo**. Then, create a **Child** class inheriting from the **Parent** class with a method **displayChildInfo**.

1. **Parent** Class:
    - Method: **displayParentInfo** that prints "This is the parent class."
2. **Child** Class (Inherits from **Parent**):
    - Method: **displayChildInfo** that prints "This is the child class."
3. Demonstrate the usage of inheritance by creating an instance of the **Child** class and calling both **displayParentInfo** and **displayChildInfo** methods.

2. Create a **Student** class with private attributes **name**, **age**, and **studentID**. Use encapsulation to provide public methods for setting and getting these attributes.

1. **Student** Class:
    - Private attributes: **name** (String), **age** (int), and **studentID** (String).
    - Public methods to set (**setName**, **setAge**, **setStudentID**) and get (**getName**, **getAge**, **getStudentID**) these attributes.
2. Demonstrate the usage of the **Student** class by creating an instance, setting attributes using setter methods, and then getting and displaying the attributes using getter methods.

3. Create a **Shape** class as the base class with a method **calculateArea**. Then, create two subclasses: **Rectangle** and **Circle**. Each subclass should override the **calculateArea** method to calculate the area of the respective shape.

1. **Shape** Class:
    - Method: **calculateArea** that returns 0.0 (to be overridden by subclasses).
2. **Rectangle** Class (Subclass of **Shape**):
    - Override **calculateArea** method to calculate the area of a rectangle (length * width).
3. **Circle** Class (Subclass of **Shape**):
    - Override **calculateArea** method to calculate the area of a circle (π * radius^2).
4. Demonstrate polymorphism by creating instances of both **Rectangle** and **Circle** classes, calling the **calculateArea** method on each, and printing the calculated areas.