Java Polymorphism

Polymorphism is derived from 2 Greek words: **poly** and **morphs**. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

That is, the same entity (method or operator or object) can perform different operations in different scenarios.

Example: Java Polymorphism

```
class Polygon {


  public void render() {
    System.out.println("Rendering Polygon...");
  }
}

class Square extends Polygon {

  // renders Square
  public void render() {
    System.out.println("Rendering Square...");
  }
}

class Circle extends Polygon {

  // renders circle
  public void render() {
    System.out.println("Rendering Circle...");
  }
}

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

    // create an object of Square
    Square s1 = new Square();
    s1.render();

    // create an object of Circle
    Circle c1 = new Circle();
    c1.render();
  }
}
```

Why Polymorphism?

Polymorphism allows us to create consistent code. In the previous example, we can also create different methods: renderSquare() and renderCircle() to render Square and Circle, respectively. This will work perfectly. However, for every shape, we need to create different methods. It will make our code inconsistent. To solve this, polymorphism in Java allows us to create a single method render() that will behave differently for different shapes.

Polymorphism in Java brings both benefits and challenges. On the positive side, it allows for the reuse of code and makes programs more flexible. By letting different objects share a common interface, developers can write code that can work with various types of objects without needing to know their specific details. This makes Java programs easier to maintain and extend over time. However, polymorphism can also add complexity to the code, making it harder to understand and debug. Additionally, there might be a slight performance overhead associated with dynamic method binding, which determines the appropriate method to execute at runtime. Despite these challenges, when used appropriately, polymorphism remains a powerful tool for building robust and adaptable Java applications.