Java Polymorphism

Polymorphism refers to the ability of an object to take on multiple (poly) forms (morph). In object-oriented programming, polymorphism allows objects of different classes to be treated as objects of a common superclass or interface. Through polymorphism, a single method or interface can be implemented in various ways by different classes, enabling flexibility and code reusability.

An analogy for you: imagine a musical band consisting of different musicians playing different instruments, such as guitar, drums, and keyboard. Each musician has their own unique way of playing their instrument. However, when they perform together as a band, they follow a common set of instructions, like starting, stopping, or following a particular rhythm. In this scenario, the concept of polymorphism comes into play. The musicians can be treated as objects of a common superclass (e.g., "Musician") or implement a common interface (e.g., "InstrumentPlayer"). This allows the band to work together harmoniously, even though each musician has their own way of playing their instrument.

Polymorphism is one of the four pillars of object-oriented programming, along with **encapsulation**, **inheritance**, and **abstraction**.

There are two types of polymorphism in Java:

1. **Compile-time polymorphism** (also known as **static polymorphism** or **method overloading**) is achieved through method overloading. This is when multiple methods have the same name but different method signatures. The Java compiler determines which method to invoke based on the number and type of arguments passed to the method at compile time.

2. **Run-time polymorphism** (also known as **dynamic polymorphism** or **method overriding**) is achieved through method overriding. This is when a subclass provides a specific implementation for a method already defined in its superclass. The Java runtime determines which method to invoke based on the type of object on which the method is called at runtime.

InfoWarningTip
In programmer-speak, the word **static** is used for things that don't change, and **dynamic** is used for things that do. Often, these changes are in response to user actions, but a program can be dynamic for any number of reasons. Get used to seeing these terms as two sides of the same coin, as in the above descriptions.

#Video Explanation

Polymorphism can be a fairly complicated topic, so it might be best to see someone talk about it a bit before you follow through with our examples below. Have a watch of Mosh here, and then carry on:

Open in new tab
https://youtu.be/jhDUxynEQRI

InfoWarningTip
Mosh talks about method overloading and method overriding quite a bit. We'll get into the details of both of these in lessons soon. If you get the gist of what he's saying, you're good. Don't worry about getting every single detail.

In your own words, what is polymorphism? How do you think it supports what you know about OOP?
Polymorphism allows you to write more generic and reusable code by treating objects based on their common behaviors rather than their specific types. It also forms the basis for concepts like interfaces and abstract classes, which enable designing systems that can accommodate a wide range of different classes and their interactions.

#Learn by Example

Rather than write a thousand words about polymorphism, we'd rather show you a series of examples that explains how it works in the Java language. Follow along, read the code, and push yourself to understand what's happening. If possible, explain it aloud to yourself, talking through what you see. Bonus points if you take someone from your household and explain the code below to them.

InfoWarningTip
Talking things out is a common programmer move. In fact, some programmers keep a rubber duck on their desk to talk to when nobody else is around. Read more about it in this short blog here if you like.

#Example 1:

Polymorphism in Java is when a superclass variable can refer to any object of its subclasses. The following example uses the `@Override` method to change the `makeSound()` method from the parent class. This allows us to morph that method into whatever we need it to be in its child classes (`Dog`, `Cat`, etc.). We'll be going more in-depth on `@Override` soon, but see if you can understand what the following examples are trying to do:

```
class Animal {
public void makeSound() {
System.out.println("Some animal sound");
}
}

class Dog extends Animal {
@Override
public void makeSound() {
System.out.println("Woof");
}
}

class Cat extends Animal {
@Override
public void makeSound() {
System.out.println("Meow");
}
}

public class Main {
public static void main(String[] args) {
Animal animal = new Dog();
```

```
animal.makeSound(); // Output: Woof

animal = new Cat();
animal.makeSound(); // Output: Meow
}
}
```

#Example 2:

Polymorphism in Java is when a single method can have different implementations in different classes.

In the instance depicted below, the `Shape` class is the base class and `Rectangle` and `Circle` are derived classes that inherit from `Shape`. The `draw()` method is overridden in both `Rectangle` and `Circle` classes. The polymorphism is demonstrated in the `main` method, where a variable of the base class `Shape` can reference an object of the derived classes `Rectangle` and `Circle`. When the `draw()` method is called on these objects, the appropriate overridden method is called, and the correct output is displayed. Try copying/pasting this code into your IDE and following along with what it does:

```
class Shape {
void draw() {
System.out.println("Drawing a shape");
}
}

class Rectangle extends Shape {
@Override
void draw() {
System.out.println("Drawing a rectangle");
}
}

class Circle extends Shape {
@Override
void draw() {
System.out.println("Drawing a circle");
}
}

public class Main {
public static void main(String[] args) {
Shape s1 = new Shape();
Shape s2 = new Rectangle();
Shape s3 = new Circle();
```

```
s1.draw();
s2.draw();
s3.draw();
}
}
```

#Example 3:

In the example below, the `Vehicle` class is an abstract class that defines the `start()` method as abstract, which means its implementation is left to the derived classes. It also defines a non-abstract method `stop()` used by all derived classes. The `Car`, `Bike`, and `Bus` classes are derived classes that inherit from the `Vehicle` class and provide their own implementation for the `start()` method. The `Driver` class has a `drive` method that takes a `Vehicle` object as a parameter and calls the `start()` and `stop()` methods on it.

The polymorphism is demonstrated in the `main` method, where a `Driver` object is created and the `drive` method is called on it with objects of the derived classes `Car`, `Bike` and `Bus` as arguments. The `drive` method calls the `start()` and `stop()` methods on the passed object, which are overridden in the respective derived classes, and the appropriate output is displayed.

```
abstract class Vehicle {
abstract void start();
void stop() {
System.out.println("Stopping the vehicle");
}
}

class Car extends Vehicle {
@Override
void start() {
System.out.println("Starting the car engine");
}
}

class Bike extends Vehicle {
@Override
void start() {
System.out.println("Kick starting the bike");
}
}
```

```
class Bus extends Vehicle {
@Override
void start() {
System.out.println("Turning on the ignition of the bus");
}
}

class Driver {
void drive(Vehicle v) {
v.start();
v.stop();
}
}

public class Main {
public static void main(String[] args) {
Driver d = new Driver();
d.drive(new Car());
d.drive(new Bike());
d.drive(new Bus());
}
}
```

#Example 4:

Another instance of polymorphism in Java is through interface implementation. An interface defines a set of methods a class must implement, but the class can provide its own unique implementation.

```
interface Shape {
public void draw();
}

class Circle implements Shape {
@Override
public void draw() {
System.out.println("Drawing a circle");
}
}

class Square implements Shape {
@Override
public void draw() {
System.out.println("Drawing a square");
}
}
```

In the illustrated examples above, the same method or variable can take on different forms depending on the object it is associated with. That's polymorphism in a nutshell.

#Conclusion

Here is a quick summary of the takeaways from this lesson:

1. Polymorphism is a fundamental concept in object-oriented programming and Java, allowing objects to take many forms.
2. Polymorphism allows a single variable, function, or object to refer to multiple data types or behaviour.
3. It is achieved through **method overriding**, **interface implementation**, or an **abstract class** or **interface as a type**.
4. Polymorphism makes our code more flexible and reusable, enabling a single piece of code to handle multiple objects or a single variable to refer to multiple data types.
5. Polymorphism also makes our code more extensible, allowing new classes to be created and used in existing code without modifying the pre-existing code.
6. In Java, polymorphism is a key feature of the object-oriented programming paradigm and is essential to creating robust, maintainable and flexible code.

#Knowledge Check

Of the following, select all that are ways to accomplish polymorphism in Java.

using an abstract class

method overriding

creating a class

interface implementation

```
class Cat extends Animal {
@[blank]
public void makeSound() {
System.out.println("Meow");
}
}
```

In the code snippet above, what word should go in the to indicate our intention of overriding the makeSound() method inherited from the Animal class?

In the code below, select all instances of the superclass. Note: do not select the entirety of the superclass's declaration. Instead, select the name given to the superclass.

```
class Animal {
public void makeSound() {
System.out.println("Some animal sound");
}
}

class Dog extends Animal {
@Override
public void makeSound() {
System.out.println("Bark");
}
}
```

Back
Unsubmit
Next