# Intro to OOP: A closer look at inheritance and polymorphism

Inheritance and polymorphism are among the most powerful features of object-oriented programming. See how Java approaches multiple inheritance and polymorphism in this introductory article on OOP.

**By Jeff Hanson**

In ”Intro to OOP: Understanding classes and objects,” we discussed the benefits of inheritance and polymorphism. We explored the concept of extending a base class to define specific subclasses that inherit the desired behaviors and properties of the base class while overriding those that don't fit. This approach helps reduce code redundancy and possible bug redundancy.

Now we will look further into inheritance by examining multiple inheritance and how Java handles it. We will also look at polymorphism through Java’s dynamic binding.

# More about inheritance

Some object-oriented languages offer a feature known as "multiple inheritance," which can be valuable when an object needs to inherit behaviors or properties from more than one base class. Multiple inheritance can also prove to be tricky in certain situations. For example, let's say we wanted to define a base class, `Animal`, and two subclasses of `Animal`, `LandAnimal` and `WaterAnimal`. Now, we want to define a class encapsulating a frog. A frog is amphibious, so, we naturally would want to define the class `Frog` as inheriting from `LandAnimal` and `WaterAnimal`. This would allow `Frog` to inherit the needed behaviors and properties from both `LandAnimal` and `WaterAnimal`.

This seems simple enough at first glance; however, let's add a property to `Animal` called, `LivingEnvironment`, returned using the method `getLivingEnvironment`. We’ll assume that `LandAnimal` and `WaterAnimal` both override this method to provide their own specific implementations. `LandAnimal` would return Land as its `LivingEnvironment` property, while `WaterAnimal` would return Water as its `LivingEnvironment` property. Now, when we implement Frog as a subclass of both `LandAnimal` and `WaterAnimal` and we want to get the `LivingEnvironment` for `Frog`, we run into a snag: Does the `getLivingEnvironment` method of `Frog` return `Land` or `Water`? The answer depends on how the compiler handles multiple inheritance.

Java does not support multiple inheritance, as defined in the previous paragraph. It does, however, allow an object to take on multiple personalities using a feature known as an "interface." The following example demonstrates a possible definition of an interface defining `LandAnimal`:

```
public interface LandAnimal
{
    public int getNumberOfLegs();
    public boolean hasATail();
}
```

A class that uses an interface adds the phrase "`implements <Interface Name>`" to the definition of the class, substituting the actual name of the interface for `<Interface Name>`." For example, in Java, we would declare our `Frog` class in the following manner:

```
public class Frog extends Animal implements LandAnimal, WaterAnimal
```

An interface defines no actual functionality; instead, it acts as a contract to users of an object that implements the interface. The interface guarantees that the object implements the methods defined by the interface. Also, an object that implements an interface can, at runtime, be cast to the interface type. For example, using the Frog definition above, and assuming that `LandAnimal` defines a method called `getNumberOfLegs` and `WaterAnimal` defines a method called `hasGills`, an instance of `Frog` could be cast at runtime to a `LandAnimal` or `WaterAnimal` like this:

```
Frog  aFrog = new Frog();
int  legCount = ((LandAnimal)aFrog).getNumberOfLegs();
Boolean  gillFlag = ((WaterAnimal)aFrog).hasGills();
```

Notice how the `Frog` instance can be cast to act as a `LandAnimal` object even though no actual `LandAnimal` object was created. This allows us to refer to an object by any one of its "personalities" at runtime and is known as "dynamic binding" or "runtime binding."

# More about polymorphism

Polymorphism in Java is made possible by dynamic binding, which refers to the mechanism used by Java to select a method or object to invoke at runtime. Overloading defines a special kind of polymorphism, and in Java, it's demonstrated when two or more methods of a class have the same name but have different parameter lists or "signatures." A method's signature refers to the method's name and the number and types of its parameters. Every method of a particular class has a unique signature relevant to the class. A class can have multiple methods with the same name as long as the parameter list for each method is unique. For example, we could define two methods in our `Animal` class with the name `getHello` using one method to retrieve the sound the animal normally makes and the other method to retrieve the sound the animal makes when scared or comforted. We will give each method a unique signature as follows:

```
public String getHello();
public String getHello(int mood);
```

Now, let's put into practice some of the concepts we have talked about by modifying our example program:

```
public class HelloWorld
{
    public static void main(String[] args)
    {
        Dog   animal1 = new Dog();
        Cat   animal2 = new Cat();
        Duck   animal3 = new Duck();

        System.out.println("A dog says " +animal1.getHello()
                                          +", when scared says: "
+animal1.getHello(Animal.SCARED)
                                          +", is carnivorous: "
+animal1.isCarnivorous()
                                         +", is a mammal: " +animal1.isAMammal());
        System.out.println("A cat says " +animal2.getHello()
                                          +", when comforted says: "
+animal2.getHello(Animal.COMFORTED)
                                         +", is carnivorous: "
+animal2.isCarnivorous()
                                          +", is a mammal: " +animal2.isAMammal());
        System.out.println("A duck says " +animal3.getHello()
                                           +", when scared says: "
+animal3.getHello(Animal.SCARED)
                                           +", is carnivorous: "
+animal3.isCarnivorous()
                                           +", is a mammal: " +animal3.isAMammal());
    }
}

abstract class Animal
{
    public static final int   SCARED = 1;
    public static final int   COMFORTED = 2;

    public boolean isAMammal()
    {
            return(true);
    }

    public boolean isCarnivorous()
    {
            return(true);
    }

    abstract public String getHello();
    abstract public String getHello(int mood);
}

interface LandAnimal
{
    public int getNumberOfLegs();
    public boolean hasATail();
}

interface WaterAnimal
{
    public boolean hasGills();
```

```
    public boolean laysEggs();
}

class Dog extends Animal implements LandAnimal
{
    // methods that override superclass's implementation
    public String getHello()
    {
        return("Bark");
    }

    public String getHello(int mood)
    {
        switch (mood) {
             case SCARED:
                     return("Growl");
             case COMFORTED:
                     return("");
        }

    return("Bark");
    }

    // Implementation of LandAnimal interface
    public int getNumberOfLegs()
    {
        return(4);
    }

    public boolean hasATail()
    {
        return(true);
    }
}

class Cat extends Animal implements LandAnimal
{
    // methods that override superclass's implementation
    public String getHello()
    {
        return("Meow");
    }

    public String getHello(int mood)
    {
        switch (mood) {
             case SCARED:
                 return("Hiss");
             case COMFORTED:
                 return("Purr");
        }

        return("Meow");
    }

    // Implementation of LandAnimal interface
    public int getNumberOfLegs()
    {
        return(4);
    }

    public boolean hasATail()
    {
```

```
        return(true);
    }
}

class Duck extends Animal implements LandAnimal, WaterAnimal
{
    // methods that override superclass's implementation
    public String getHello()
    {
        return("Quack");
    }

    public String getHello(int mood)
    {
        switch (mood) {
             case SCARED:
                 return("Quack, Quack, Quack");
             case COMFORTED:
                 return("");
        }

        return("Quack");
    }

    public boolean isAMammal()
    {
        return(false);
    }

    public boolean isCarnivorous()
    {
        return(false);
    }

    // Implementation of WaterAnimal interface
    public boolean hasGills()
    {
        return(false);
    }

    public boolean laysEggs()
    {
        return(true);
    }

    // Implementation of LandAnimal interface
    public int getNumberOfLegs()
    {
        return(2);
    }

    public boolean hasATail()
    {
        return(false);
    }
}
```

The output from our program when executed should appear as follows:

```
A dog says Bark, when scared says: Growl, is carnivorous: true, is a mammal: true
A cat says Meow, when comforted says: Purr, is carnivorous: true, is a mammal: true
A duck says Quack, when scared says: Quack, Quack, Quack, is carnivorous: false, is
a mammal: false
```

# Summary

The combination of inheritance, polymorphism, and interfaces provides a powerful set of programming tools, allowing us to reuse code, isolate occurrences of bugs, and take advantage of runtime/dynamic binding. In our next article, we will discuss how to control the exposure of our methods and properties using Java's scope/visibility rules.