Intro to OOP: Understanding classes and objects

If you're not familiar with object-oriented programming, some of the concepts can be hard to understand, especially if you're a longtime procedural language programmer. Follow along as we take a look at two key OOP concepts: classes and objects.

By Jeff Hanson

Last time, in ”Transitioning into object-oriented programming using Java,” we learned that for a programming language to be truly object-oriented, it should support information hiding/encapsulation, polymorphism, inheritance, and dynamic binding. In addition, we learned that Java fully supports these features, and that because Java is an interpreted language that runs inside a virtual machine, programs written using Java can be executed on any operating system that supports a Java Virtual Machine (JVM). We also learned that objects are software-programming entities that represent real-life objects and that they are defined by their state and behavior. Finally, we learned that everything in Java is an object, except primitive data types.

Because so much of this programming style involves objects and classes, we will now look at each of these further.

More about objects

One of the keys to working with objects is knowing how to identify them when you look at system-analysis documents or design documents. Since objects generally represent people, places, or things, a basic method of identifying objects is to pick out the nouns used in a sentence. Here’s a simple example. In the sentence "A customer can have more than one bank account," we identify two objects, customer and account. In the sentence, "The cat meows," we identify one object, cat.

More about classes

Previously, we learned that a class is an entity that defines how an object will behave and what the object will contain when the object is constructed or instantiated. In a discussion about animals, we might say, "Dogs bark, cats meow, and ducks quack." Identifying the objects in the sentence, we end up with dogs, cats, and ducks. Bark, meow, and quack are behaviors performed by our objects.

To implement the objects, we could create three objects named `Dog`, `Cat`, and `Duck`. To implement the behaviors, we could create a method for each of these objects that represents the sound each animal makes, and we could call the method speak or, if we use our

imagination, we could call the method `sayHello`. In the context of a program

To illustrate these concepts, let's modify the `HelloWorld` application we created in our previous article by introducing these three new objects and giving each of them a `sayHello` method, as shown here:

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

class Dog
{
        public void sayHello()
        {
                System.out.println("Bark");
        }
}

class Cat
{
        public void sayHello()
        {
                System.out.println("Meow");
        }
}

class Duck
{
        public void sayHello()
        {
               System.out.println("Quack");
        }
}
```

After compiling and running our application, the output will be as follows:

```
Bark
Meow
Quack
```

Looking at our application, we immediately notice a couple of things: Each object represents an animal, and each object implements a common method, `sayHello`. Let's assume that we want to give our objects more functionality and add behaviors and attributes that represent each animal that the objects represent. For example, we could add a method to determine whether each animal is a mammal, and we could add a method to determine whether each animal is

carnivorous. We could do this by adding each method to every object or we could call into play two of the most powerful features of OOP: inheritance and polymorphism.

Since all of the objects represent animals, we'll create a class, known as a "base" or "super" class, called `Animal`. We can then allow all of our objects to inherit common features from the `Animal` class and force each object to define only the features that are different from the `Animal` class.

Java uses the keyword extends to indicate that one class inherits from another. Let's rework our application by applying inheritance and polymorphism to take advantage of code reuse and to enable each object to implement only the features that are not common with the base class, `Animal`:

```
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()
                           +", is carnivorous: " +animal1.isCarnivorous()
                           +", is a mammal: " +animal1.isAMammal());
       System.out.println("A cat says " +animal2.getHello()
                           +", is carnivorous: " +animal2.isCarnivorous()
                           +", is a mammal: " +animal2.isAMammal());
       System.out.println("A duck says " +animal3.getHello()
                          +", is carnivorous: " +animal3.isCarnivorous()
                           +", is a mammal: " +animal3.isAMammal());
   }
}

abstract class Animal
{
        public boolean isAMammal()
        {
                return(true);
        }

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

        abstract public String getHello();
}

class Dog extends Animal
{
        public String getHello()
        {
                return("Bark");
        }
}
```

```
class Cat extends Animal
{
        public String getHello()
        {
                return("Meow");
        }
}

class Duck extends Animal
{
         public boolean isAMammal()
        {
                return(false);
        }

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

        public String getHello()
        {
                return("Quack");
        }
}
```

After compiling and running our application, the output will be as follows:

```
A dog says Bark, is carnivorous: true, is a mammal: true
A cat says Meow, is carnivorous: true, is a mammal: true
A duck says Quack, is carnivorous: false, is a mammal: false
```

Looking at our example, we see that we declared a new class called `Animal` that defines three methods: `isAMammal`, `isCarnivorous`, and `getHello`. We also notice that we added the extends `Animal` phrase to each of our existing objects' declarations. This tells the compiler that these objects are descendants of the `Animal` class.

Since the methods `isAMammal` and `isCarnivorous` both return true, `Dog` and `Cat` do not need to reimplement or "override" these two methods. A duck, however, is neither a mammal nor carnivorous, so our `Duck` class needs to override these two methods and return the correct value. All of our objects "say hello" in a different manner, so, they all need to override the `getHello` method. Since the manner in which each animal says hello differs, we declare the `getHello` method in our base class, Animal, as abstract, and we do not give the method a body. This forces each descendant of `Animal` to override the `getHello` method and define it as needed for the particular animal.

Because we have declared the `getHello` method abstract, we cannot directly instantiate an instance of `Animal`. Thus, we need to declare the `Animal` class as abstract. We did this by placing the keyword abstract at the beginning of the `Animal` class definition. The ability of descendant classes to override methods of their base class is known as polymorphism. Polymorphism enables descendant classes to use the methods of the base class or override them if the base

class implementation is not sufficient. This promotes code reuse, enabling swifter code implementation, and it isolates bugs for easier code maintenance.

Summary

In this article, we learned how to identify potential objects. We also learned how to use inheritance and polymorphism to speed up our code implementation time and to isolate bugs, which makes our code maintenance tasks much easier. Next time, we'll expand on the concepts of polymorphism and inheritance and start our discussion of dynamic binding.