Chapter 2: Class Hierarchy

Class Hierarchy

Java is an object-oriented language. Part of what that means is that in Java, every new data type we create and every new program we write are classes. We have been using classes for quite a while now, so it's time we dig a little deeper into how class hierarchies work in Java.

When I say a *class hierarchy*, I'm referring to how every class in Java depends on, or *inherits* from, at least one other class. We call the class that inherits a *subclass* or a *child class*. The class that is inherited *from* is called a *superclass*, *parent class*, or *base class*.

Each of those classes, in turn, can depend on other classes above them in the hierarchy. Ultimately, every class traces its way back to Java's Object class. Every data type in Java except the primitive types (int, float, bool, and so on) inherits from the Object class.

What does it mean for a class to *inherit* from another class? It means that the child class contains all the characteristics, including the data fields and methods, of its parent. Since the parent also inherits from its parent, the bottom child class in a hierarchy inherits all the data and methods of all its *ancestors* above it in the hierarchy.

Before we get too far into this discussion and your eyes glaze over, let's look at a small example of a class hierarchy to help explain these concepts.

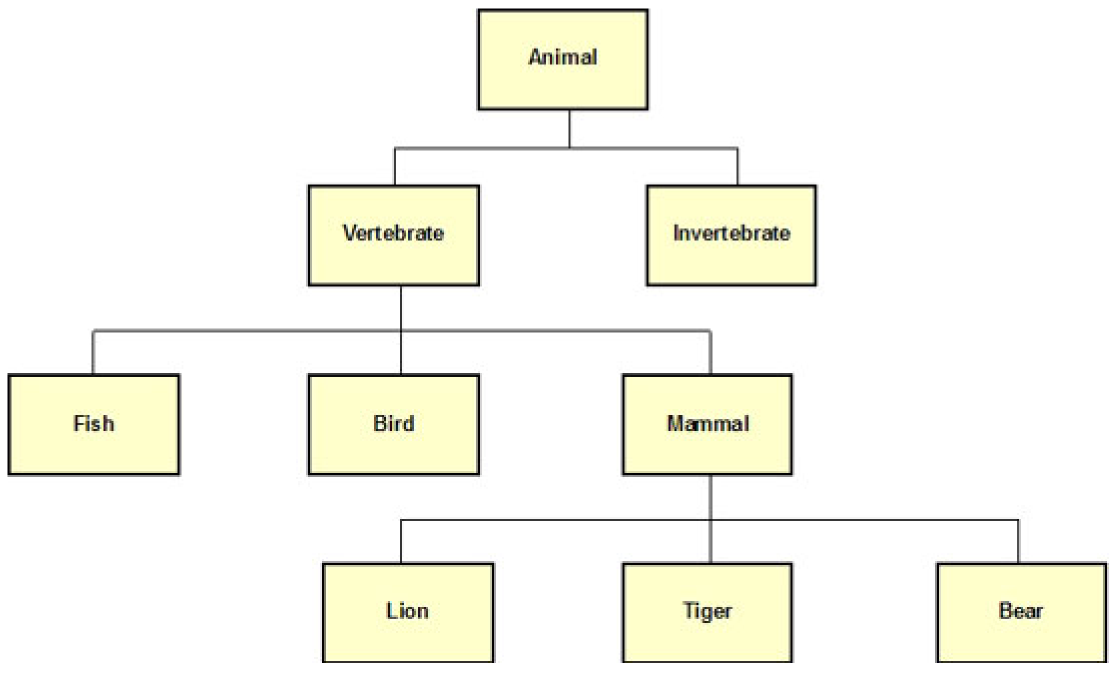
Suppose we want to set up a class hierarchy that describes some animals for an application that we could use in a biology class or at a zoo. First, we'll set up a base class named Animal. We'll put data and methods in that class that apply to *all* animals. For example, all animals move, and all animals eat. We'll put those actions in our Animal class and call them move() and eat(). (No surprises there!)

Let's also set up a diagram of our hierarchy so we have a visual representation to refer to. We'll use rectangles to represent our classes. Here's our Animal class:

A single class

Animal is too generic to represent all the animals we'll want to work with in our application, though. To take care of that, let's create some subclasses of the Animal class. As subclasses, they'll inherit the characteristics of their parent class, and then we'll add more specific data and methods that distinguish them from each other. For example, logical subclasses of Animal would be Vertebrate and Invertebrate. We could add the major characteristic that makes them different: an internal skeleton (or the lack thereof).

Under Vertebrate, we'll add more subclasses like Fish, Bird, and Mammal, with their unique characteristics: gills for Fish, feathers for Bird, and hair for Mammal. We'll add three more subclasses for Mammal just to fill out our example: Lion, Tiger, and Bear. (Oh, my!) Our example hierarchy looks like this:



A class hierarchy

Each subclass inherits all the characteristics (fields and methods) of its superclass, but it also has its own unique characteristics. For example, bears move differently than lions and tigers, since they sometimes walk on their hind feet. Bears hibernate; lions and tigers don't. Bears eat differently, too—they eat fruit and vegetables as well as meat. They are all mammals, though, and inherit all the Mammal characteristics.

Subclasses have another relationship with their parent classes that I call the "is a" relationship. In other words, an object of the subclass "is a" parent class object, too. A Bear is a Mammal, a Mammal is a Vertebrate, and so on. Whenever a program expects an object of type Mammal, an object of type Bear (or Lion or Tiger) will work, too, since they are Mammals.

By now you may be asking yourself, "What on earth does all this have to do with Java programming?" Let's talk about that.

The Java API Hierarchy

You probably already know about the Java API Specification. (If you don't, it's at Sun's Web site. See this lesson's Supplementary Material for a link.) The Java API Specification is a list of all the packages and classes that are part of Java. It also includes descriptions of their public fields and methods.

The connection to this lesson is that the API spells out the class hierarchy that comes with Java. It's a huge one! To get a better idea of how it works, we'll start with a quick and familiar example.

First, let's say we want to find out how to use the indexOf() method of the String class. We go to the API, scroll down until we see String in the lower-left pane, and click it. Documentation for the class will appear in the main pane.

We can immediately see that String is part of the java.lang package. (Its full name is java.lang.String.) We also see at the top of the pane that String inherits directly from the Object class. Just below that, we see its declaration:

```
public final class String

extends Object

implements Serializable, Comparable<String>, CharSequence
```

Under that are several paragraphs describing the class, followed by short descriptions of its field, its constructors (15 of them if I counted right), and its methods (more than 60 of them). Among the methods are the descriptions of the four versions of indexOf(). If I want more detail on one of them, I can click its name to jump to a longer description.

Are you getting a better idea of how class hierarchies work in Java? If you have any further questions about the String class hierarchy, let me know in the Discussion area.

Now, let's move on to our main topic!