Object-Oriented Programming and Java

Object-oriented programming (OOP) is one of the bigger programming buzzwords of recent years, and you can spend years learning all about object-oriented programming methodologies and how they can make your life easier than The Old Way of programming. It all comes down to organizing your programs in ways that echo how things are put together in the real world.

Today, you'll get an overview of object-oriented programming concepts in Java and how they relate to how you structure your own programs:

- ☐ What classes and objects are, and how they relate to each other
- ☐ The two main parts of a class or object: its behaviors and its attributes
- ☐ Class inheritance and how inheritance affects the way you design your programs
- ☐ Some information about packages and interfaces

If you're already familiar with object-oriented programming, much of today's lesson will be old hat to you. You may want to skim it and go to a movie today instead. Tomorrow, you'll get into more specific details.

Thinking in Objects: An Analogy

Consider, if you will, Legos. Legos, for those who do not spend much time with children, are small plastic building blocks in various colors and sizes. They have small round bits on one side that fit into small round holes on other Legos so that they fit together snugly to create larger shapes. With different Lego bits (Lego wheels, Lego engines, Lego hinges, Lego pulleys), you can put together castles, automobiles, giant robots that swallow cities, or just about anything else you can create. Each Lego bit is a small object that fits together with other small objects in predefined ways to create other larger objects.

Here's another example. You can walk into a computer store and, with a little background and often some help, assemble an entire PC computer system from various components: a motherboard, a CPU chip, a video card, a hard disk, a keyboard, and so on. Ideally, when you finish assembling all the various self-contained units, you have a system in which all the units work together to create a larger system with which you can solve the problems you bought the computer for in the first place.

Internally, each of those components may be vastly complicated and engineered by different companies with different methods of design. But you don't need to know how the component works, what every chip on the board does, or how, when you press the A key, an "A" gets sent to your computer. As the assembler of the overall system, each component you use is a self-contained unit, and all you are interested in is how the units interact with each other. Will this video card fit into the slots on the motherboard and will this monitor work with this video card? Will each particular component speak the right commands to the other components it interacts with so that each part of the computer is understood by every other part? Once you know what the interactions are between the components and can match the interactions, putting together the overall system is easy.

What does this have to do with programming? Everything. Object-oriented programming works in exactly this same way. Using object-oriented programming, your overall program is made up of lots of different self-contained components (objects), each of which has a specific role in the program and all of which can talk to each other in predefined ways.

Objects and Classes

Object-oriented programming is modeled on how, in the real world, objects are often made up of many kinds of smaller objects. This capability of combining objects, however, is only one very general aspect of object-oriented programming. Object-oriented programming provides several other concepts and features to make creating and using objects easier and more flexible, and the most important of these features is that of classes.

NEW TERM A *class* is a template for multiple objects with similar features. Classes embody all the features of a particular set of objects.

When you write a program in an object-oriented language, you don't define actual objects. You define classes of objects.

For example, you might have a `Tree` class that describes the features of all trees (has leaves and roots, grows, creates chlorophyll). The `Tree` class serves as an abstract model for the concept of a tree—to reach out and grab, or interact with, or cut down a tree you have to have a concrete instance of that tree. Of course, once you have a tree class, you can create lots of different instances of that tree, and each different tree instance can have different features (short, tall, bushy, drops leaves in Autumn), while still behaving like and being immediately recognizable as a tree (see Figure 2.1).

NEW TERM An *instance* of a class is another word for an actual object. If classes are an abstract representation of an object, an instance is its concrete representation.

So what, precisely, is the difference between an instance and an object? Nothing, really. Object is the more general term, but both instances and objects are the concrete representation of a class. In fact, the terms instance and object are often used interchangeably in OOP language. An instance of a tree and a tree object are both the same thing.

In an example closer to the sort of things you might want to do in Java programming, you might create a class for the user interface element called a button. The `Button` class defines the features of a button (its label, its size, its appearance) and how it behaves (does it need a single click or a double click to activate it, does it change color when it's clicked, what does it do when it's activated?). Once you define the `Button` class, you can then easily create instances of that button—that is, button objects—that all take on the basic features of the button as defined by the class, but may have different appearances and behavior based on what you want that particular button to do. By creating a `Button` class, you don't have to keep rewriting the code for each individual button you want to use in your program, and you can reuse the `Button` class to create different kinds of buttons as you need them in this program and in other programs.

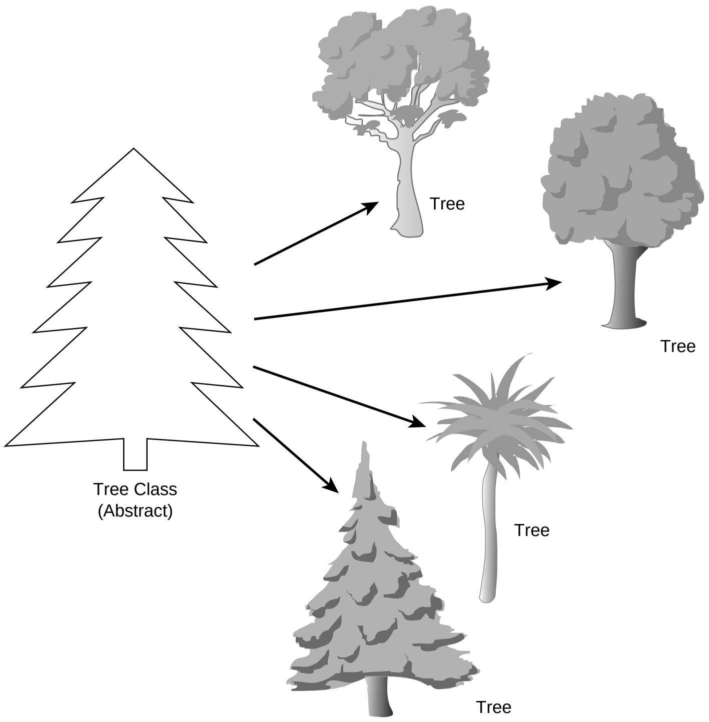

Figure 2.1.
The tree class and tree instances.

Tip: If you're used to programming in C, you can think of a class as sort of creating a new composite data type by using `struct` and `typedef`. Classes, however, can provide much more than just a collection of data, as you'll discover in the rest of today's lesson.

When you write a Java program, you design and construct a set of classes. Then, when your program runs, instances of those classes are created and discarded as needed. Your task, as a Java programmer, is to create the right set of classes to accomplish what your program needs to accomplish.

Fortunately, you don't have to start from the very beginning: the Java environment comes with a library of classes that implement a lot of the basic behavior you need—not only for basic programming tasks (classes to provide basic math functions, arrays, strings, and so on), but also for graphics and networking behavior. In many cases, the Java class libraries may be enough so that all you have to do in your Java program is create a single class that uses the standard class libraries. For complicated Java programs, you may have to create a whole set of classes with defined interactions between them.

NEW TERM A *class library* is a set of classes.

Behavior and Attributes

Every class you write in Java is generally made up of two components: attributes and behavior. In this section, you'll learn about each one as it applies to a thoeretical class called `Motorcycle`. To finish up this section, you'll create the Java code to implement a representation of a motorcycle.

Attributes

Attributes are the individual things that differentiate one object from another and determine the appearance, state, or other qualities of that object. Let's create a theoretical class called `Motorcycle`. The attributes of a motorcycle might include the following:

- *Color:* red, green, silver, brown
- *Style:* cruiser, sport bike, standard
- *Make:* Honda, BMW, Bultaco

Attributes of an object can also include information about its state; for example, you could have features for engine condition (off or on) or current gear selected.

Attributes are defined by variables; in fact, you can consider them analogous to global variables for the entire object. Because each instance of a class can have different values for its variables, each variable is called an instance variable.

NEW TERM *Instance variables* define the attributes of an object. The class defines the kind of attribute, and each instance stores its own value for that attribute.

Each attribute, as the term is used here, has a single corresponding instance variable; changing the value of a variable changes the attribute of that object. Instance variables may be set when an object is created and stay constant throughout the life of the object, or they may be able to change at will as the program runs.

In addition to instance variables, there are also class variables, which apply to the class itself and to all its instances. Unlike instance variables, whose values are stored in the instance, class variables' values are stored in the class itself. You'll learn about class variables later on this week; you'll learn more specifics about instance variables tomorrow.

Behavior

A class's behavior determines what instances of that class do when their internal state changes or when that instance is asked to do something by another class or object. Behavior is the way objects can do anything to themselves or have anything done to them. For example, to go back to the theoretical `Motorcycle` class, here are some behaviors that the `Motorcycle` class might have:

- Start the engine
- Stop the engine
- Speed up
- Change gear
- Stall

To define an object's behavior, you create methods, which look and behave just like functions in other languages, but are defined inside a class. Java does not have functions defined outside classes (as C++ does).

NEW TERM *Methods* are functions defined inside classes that operate on instances of those classes.

Methods don't always affect only a single object; objects communicate with each other using methods as well. A class or object can call methods in another class or object to communicate changes in the environment or to ask that object to change its state.

Just as there are instance and class variables, there are also instance and class methods. Instance methods (which are so common they're usually just called methods) apply and operate on an instance; class methods apply and operate on a class (or on other objects). You'll learn more about class methods later on this week.

Creating a Class

Up to this point, today's lesson has been pretty theoretical. In this section, you'll create a working example of the `Motorcycle` class so that you can see how instance variables and methods are defined in a class. You'll also create a Java application that creates a new instance of the `Motorcycle` class and shows its instance variables.

Note: I'm not going to go into a lot of detail about the actual syntax of this example here. Don't worry too much about it if you're not really sure what's going on; it will become clear to you later on this week. All you really need to worry about in this example is understanding the basic parts of this class definition.

Ready? Let's start with a basic class definition. Open up that editor and enter the following:

```
class Motorcycle {

}
```

Congratulations! You've now created a class. Of course, it doesn't do very much at the moment, but that's a Java class at its very simplest.

First, let's create some instance variables for this class—three of them, to be specific. Just below the first line, add the following three lines:

```
String make;
String color;
boolean engineState;
```

Here, you've created three instance variables: two, `make` and `color`, can contain `String` objects (`String` is part of that standard class library mentioned earlier). The third, `engineState`, is a boolean that refers to whether the engine is off or on.

Technical Note: `boolean` in Java is a real data type that can have the value `true` or `false`. Unlike C, booleans are not numbers. You'll hear about this again tomorrow so you won't forget.

Now let's add some behavior (methods) to the class. There are all kinds of things a motorcycle can do, but to keep things short, let's add just one method—a method that starts the engine. Add the following lines below the instance variables in your class definition:

```
void startEngine() {
    if (engineState == true)
        System.out.println("The engine is already on.");
    else {
        engineState = true;
        System.out.println("The engine is now on.");
    }
}
```

The `startEngine` method tests to see whether the engine is already running (in the line `engineState == true`) and, if it is, merely prints a message to that effect. If the engine isn't already running, it changes the state of the engine to `true` and then prints a message.

With your methods and variables in place, save the program to a file called Motorcycle.java (remember, you should always name your Java files the same names as the class they define). Here's what your program should look like so far:

```
class Motorcycle {

    String make;
    String color;
    boolean engineState;

    void startEngine() {
        if (engineState == true)
            System.out.println("The engine is already on.");
        else {
            engineState = true;
            System.out.println("The engine is now on.");
        }
    }
}
```

✓ **Tip:** The indentation of each part of the class isn't important to the Java compiler. Using some form of indentation, however, makes your class definition easier for you and for other people to read. The indentation used here, with instance variables and methods indented from the class definition, is the style used throughout this book. The Java class libraries use a similar indentation. You can choose any indentation style that you like.

Before you compile this class, let's add one more method. The `showAtts` method prints the current values of the instance variables in an instance of your `Motorcycle` class. Here's what it looks like:

```
void showAtts() {
    System.out.println("This motorcycle is a "
        + color + " " + make);
    if (engineState == true)
        System.out.println("The engine is on.");
    else System.out.println("The engine is off.");
}
```

The `showAtts` method prints two lines to the screen: the `make` and `color` of the motorcycle object, and whether or not the engine is on or off.

Save that file again and compile it using `javac`:

```
javac Motorcycle.java
```

Note: After this point, I'm going to assume you know how to compile and run Java programs. I won't repeat this information after this.

What happens if you now use the Java interpreter to run this compiled class? Try it. Java assumes that this class is an application and looks for a `main` method. This is just a class, however, so it doesn't have a `main` method. The Java interpreter (`java`) gives you an error like this one:

```
In class Motorcycle: void main(String argv[]) is not defined
```

To do something with the Motorcycle class—for example, to create instances of that class and play with them—you're going to need to create a Java application that uses this class or add a `main` method to this one. For simplicity's sake, let's do the latter. Listing 2.1 shows the `main()` method you'll add to the `Motorcycle` class (you'll go over what this does in a bit).

Type

Listing 2.1. The `main()` method for `Motorcycle.java`.

```
public static void main (String args[]) {
   Motorcycle m = new Motorcycle();
   m.make = "Yamaha RZ350";
   m.color = "yellow";
   System.out.println("Calling showAtts...");
   m.showAtts();
   System.out.println("--------");
   System.out.println("Starting engine...");
   m.startEngine();
   System.out.println("--------");
   System.out.println("Calling showAtts...");
   m.showAtts();
   System.out.println("--------");
   System.out.println("Starting engine...");
   m.startEngine();
}
```

With the `main()` method, the `Motorcycle` class is now an application, and you can compile it again and this time it'll run. Here's how the output should look:

Output

```
Calling showAtts...
This motorcycle is a yellow Yamaha RZ350
The engine is off.
--------
```

```
Starting engine...
The engine is now on.
--------
Calling showAtts...
This motorcycle is a yellow Yamaha RZ350
The engine is on.
--------
Starting engine...
The engine is already on.
```

Analysis The contents of the `main()` method are all going to look very new to you, so let's go through it line by line so that you at least have a basic idea of what it does (you'll get details about the specifics of all of this tomorrow and the day after).

The first line declares the `main()` method. The `main()` method always looks like this; you'll learn the specifics of each part later this week.

Line 2, `Motorcycle m = new Motorcycle()`, creates a new instance of the `Motorcycle` class and stores a reference to it in the variable `m`. Remember, you don't usually operate directly on classes in your Java programs; instead, you create objects from those classes and then modify and call methods in those objects.

Lines 3 and 4 set the instance variables for this motorcycle object: the make is now a `Yamaha RZ350` (a very pretty motorcycle from the mid-1980s), and the color is `yellow`.

Lines 5 and 6 call the `showAtts()` method, defined in your motorcycle object. (Actually, only 6 does; 5 just prints a message that you're about to call this method.) The new motorcycle object then prints out the values of its instance variables—the `make` and `color` as you set in the previous lines—and shows that the engine is off.

Line 7 prints a divider line to the screen; this is just for prettier output.

Line 9 calls the `startEngine()` method in the motorcycle object to start the engine. The engine should now be on.

Line 12 prints the values of the instance variables again. This time, the report should say the engine is now on.

Line 15 tries to start the engine again, just for fun. Because the engine is already on, this should print the error message.

Inheritance, Interfaces, and Packages

Now that you have a basic grasp of classes, objects, methods, variables, and how to put it all together in a Java program, it's time to confuse you again. Inheritance, interfaces, and packages are all mechanisms for organizing classes and class behaviors. The Java class libraries use all these concepts, and the best class libraries you write for your own programs will also use these concepts.

Inheritance

Inheritance is one of the most crucial concepts in object-oriented programming, and it has a very direct effect on how you design and write your Java classes. Inheritance is a powerful mechanism that means when you write a class you only have to specify how that class is different from some other class, while also giving you dynamic access to the information contained in those other classes.

NEW TERM With inheritance, all classes—those you write, those from other class libraries that you use, and those from the standard utility classes as well—are arranged in a strict hierarchy (see Figure 2.2).

Each class has a superclass (the class above it in the hierarchy), and each class can have one or more subclasses (classes below that class in the hierarchy). Classes further down in the hierarchy are said to inherit from classes further up in the hierarchy.

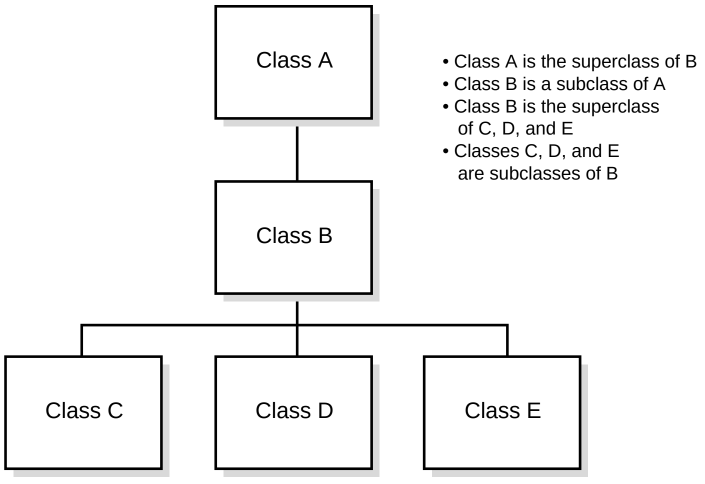
Class A
Class B
Class C
Class D
Class E
• Class A is the superclass of B
• Class B is a subclass of A
• Class B is the superclass of C, D, and E
• Classes C, D, and E are subclasses of B

Figure 2.2.
A class hierarchy.

Subclasses inherit all the methods and variables from their superclasses—that is, in any particular class, if the superclass defines behavior that your class needs, you don't have to redefine it or copy that code from some other class. Your class automatically gets that behavior from its superclass, that superclass gets behavior from its superclass, and so on all the way up the hierarchy. Your class becomes a combination of all the features of the classes above it in the hierarchy.

At the top of the Java class hierarchy is the class `Object`; all classes inherit from this one superclass. `Object` is the most general class in the hierarchy; it defines behavior specific to all objects in the Java class hierarchy. Each class farther down in the hierarchy adds more information and becomes more tailored to a specific purpose. In this way, you can think of a class hierarchy as defining very abstract concepts at the top of the hierarchy and those ideas becoming more concrete the farther down the chain of superclasses you go.

Most of the time when you write new Java classes, you'll want to create a class that has all the information some other class has, plus some extra information. For example, you may want a version of a `Button` with its own built-in label. To get all the `Button` information, all you have to do is define your class to inherit from `Button`. Your class will automatically get all the behavior defined in `Button` (and in `Button`'s superclasses), so all you have to worry about are the things that make your class different from `Button` itself. This mechanism for defining new classes as the differences between them and their superclasses is called *subclassing*.

NEW TERM *Subclassing* involves creating a new class that inherits from some other class in the class hierarchy. Using subclassing, you only need to define the differences between your class and its parent; the additional behavior is all available to your class through inheritance.

What if your class defines entirely new behavior, and isn't really a subclass of another class? Your class can also inherit directly from `Object`, which still allows it to fit neatly into the Java class hierarchy. In fact, if you create a class definition that doesn't indicate its superclass in the first line, Java automatically assumes you're inheriting from `Object`. The `Motorcycle` class you created in the previous section inherited from `Object`.

Creating a Class Hierarchy

If you're creating a larger set of classes, it makes sense for your classes not only to inherit from the existing class hierarchy, but also to make up a hierarchy themselves. This may take some planning beforehand when you're trying to figure out how to organize your Java code, but the advantages are significant once it's done:

- When you develop your classes in a hierarchy, you can factor out information common to multiple classes in superclasses, and then reuse that superclass's information over and over again. Each subclass gets that common information from its superclass.
- Changing (or inserting) a class further up in the hierarchy automatically changes the behavior of the lower classes—no need to change or recompile any of the lower classes, because they get the new information through inheritance and not by copying any of the code.

For example, let's go back to that `Motorcycle` class, and pretend you created a Java program to implement all the features of a motorcycle. It's done, it works, and everything is fine. Now, your next task is to create a Java class called `Car`.

`Car` and `Motorcycle` have many similar features—both are vehicles driven by engines. Both have transmissions and headlamps and speedometers. So, your first impulse may be to open up your `Motorcycle` class file and copy over a lot of the information you already defined into the new class `Car`.