TEACH MYSELF JAVA IN 21 DAYS PROJECT

About This Book This book

teaches you all about the Java language and how to use it to create applets and applications. By the time you get through with this book, you’ll know enough about Java to do just about anything, inside an applet or out.

**APPLETS: in java** appletes are used to provide interactive features to web application and can be executed by browser for many platforms.

They are small, portable java programs embedded in html page and can run automatically when pages are viewed.

Teach Yourself Java in 21 Days covers the Java language and its class libraries in 21 days, organized as three separate weeks. Each week covers a different broad area of developing Java applets and applications.

In the first week, you’ll learn about the Java language itself:

■ Day 1 is the basic introduction: what Java is, why it’s cool, and how to get the software. You’ll also create your first Java applications and applets.

■ On Day 2, you’ll explore basic object-oriented programming concepts as they apply to Java.

■ On Day 3, you start getting down to details with the basic Java building blocks: data types, variables, and expressions such as arithmetic and comparisons.

■ Day 4 goes into detail about how to deal with objects in Java: how to create them, how to access their variables and call their methods, and how to compare and copy them. You’ll also get your first glance at the Java class libraries.

■ On Day 5, you’ll learn more about Java with arrays, conditional statements. and loops.

■ Day 6 is the best one yet. You’ll learn how to create classes, the basic building blocks of any Java program, as well as how to put together a Java application (an application being a Java program that can run on its own without a Web browser).

■ Day 7 builds on what you learned on Day 6. On Day 7, you’ll learn more about how to create and use methods, including overriding and overloading methods and creating constructors.

Week 2 is dedicated to applets and the Java class libraries:

■ Day 8 provides the basics of applets—how they’re different from applications, how to create them, and the most important parts of an applet’s life cycle. You’ll also learn how to create HTML pages that contain Java applets. ■ On Day 9, you’ll learn about the Java classes for drawing shapes and characters to the screen—in black, white, or any other color.

■ On Day 10, you’ll start animating those shapes you learned about on Day 9, including learning what threads and their uses are.

■ Day 11 covers more detail about animation, adding bitmap images and audio to the soup.

■ Day 12 delves into interactivity—handling mouse and keyboard clicks from the user in your Java applets.

■ Day 13 is ambitious; on that day you’ll learn about using Java’s Abstract Windowing Toolkit to create a user interface in your applet including menus, buttons, checkboxes, and other elements.

■ On Day 14, you explore the last of the main Java class libraries for creating applets: windows and dialogs, networking, and a few other tidbits.

Week 3 finishes up with advanced topics, for when you start doing larger and more complex Java programs, or when you want to learn more:

■ On Day 15, you’ll learn more about the Java language’s modifiers—for abstract and final methods and classes as well as for protecting a class’s private information from the prying eyes of other classes.

Teach Yourself JAVA in 21 Days

■ Day 16 covers interfaces and packages, useful for abstracting protocols of methods to aid reuse and for the grouping and categorization of classes.

■ Day 17 covers exceptions: errors and warnings and other abnormal conditions, generated either by the system or by you in your programs.

■ Day 18 builds on the thread basics you learned on Day 10 to give a broad overview of multithreading and how to use it to allow different parts of your Java programs to run in parallel.

■ On Day 19, you’ll learn all about the input and output streams in Java’s I/O library.

■ Day 20 teaches you about native code—how to link C code into your Java programs to provide missing functionality or to gain performance.

■ Finally, on Day 21, you’ll get an overview of some of the “behind-the-scenes” technical details of how Java works: the bytecode compiler and interpreter, the techniques Java uses to ensure the integrity and security of your programs, and the Java garbage collector

*Week 1*

An Introduction to Java Programming

Platform independence

The Java compiler and the java interpreter

■ Object-Oriented Programming and Java

Objects and classes

Encapsulation

Modularity

■ Java Basics

Java statements and expressions

Variables and data types

Comparisons and logical operators

■ Working with Objects

Testing and modifying instance variables

Converting objects

■ Arrays, Conditionals, and Loops

Conditional tests

Iteration

Block statements

Creating Classes and Applications in Java

Defining constants, instance and class

variables, and methods

■ ■ More About Methods

Overloading methods

Constructor methods

Overriding methods

*CHAPTER 1 AN INTRODUCTION TO JAVA PROGRAMMING*

***An applet*** is a dynamic and interactive program that can run inside a Web page displayed by a Java-capable browser such as HotJava or Netscape 2.0.

***The HotJava browser*** is a World Wide Web browser used to view Web pages, follow links, and submit forms. It can also download and play applets on the reader’s system.

***WHAT IS JAVA***

***Java*** is an object-oriented programming language developed by Sun Microsystems, a company best known for its high-end Unix workstations. Modeled after C++, the Java language was designed to be small, simple, and portable across platforms and operating systems, both at the source and at the binary level.

Java is often mentioned in the same breath as HotJava, a World Wide Web browser from Sun like Netscape or Mosaic. What makes HotJava different from most other browsers is that, in addition to all its basic Web features, it can also download and play applets on the reader’s system. Applets appear in a Web page much in the same way as images do, but unlike images, applets are dynamic and interactive. ***Applets*** can be used to create animations, figures, or areas that can respond to input from the reader, games, or other interactive effects on the same Web pages among the text and graphics.

Although HotJava was the first World Wide Web browser to be able to play Java applets, Java support is rapidly becoming available in other browsers. Netscape 2.0 provides support for Java applets, and other browser developers have also announced support for Java in forthcoming products.

***The hot java browser***



To create an applet, you write it in the Java language, compile it using a Java compiler, and refer to that applet in your HTML Web pages. You put the resulting HTML and Java files on a Web site much in the same way that you make ordinary HTML and image files available. Then, when someone using the HotJava browser (or other Java-aware browser) views your page with the embedded applet, that browser downloads the applet to the local system and executes it, and then the reader can view and interact with your applet in all its glory (readers using other browsers won’t see anything

The important thing to understand about Java is that you can do so much more with it besides create applets. Java was written as a full-fledged programming language in which you can accomplish the same sorts of tasks and solve the same sorts of problems that you can in other programming languages, such as C or C++. HotJava itself, including all the networking, display, and user interface elements, is written in Java.

***The java’s past present and future***

The Java language was developed at Sun Microsystems in 1991 as part of a research project to develop software for consumer electronics devices—television sets, VCRs, toasters, and the other sorts of machines you can buy at any department store. Java’s goals at that time were to be small, fast, efficient, and easily portable to a wide range of hardware devices. It is those same goals that made Java an ideal (perfect, model) language for distributing executable programs via the World Wide Web, and also a general-purpose programming language for developing programs that are easily usable and portable across different platforms.

The Java language was used in several projects within Sun, but did not get very much commercial attention until it was paired with HotJava. HotJava was written in 1994 in a matter of months, both as a vehicle for downloading and running applets and also as an example of the sort of complex application that can be written in Java.

At the time this book is being written, Sun has released the beta version of the Java Developer’s Kit (JDK), which includes tools for developing Java applets and applications on Sun systems running Solaris 2.3 or higher for Windows NT and for Windows 95.

JDK: is a tool box that include tools for developing java applets and application

Note that because the JDK is currently in beta, it is still subject to change between now and when it is officially released. Applets and applications you write using the JDK and using the examples in this book may require some changes to work with future versions of the JDK. However, because the Java language has been around for several years and has been used for several projects, the language itself is quite stable(un changing, constant) and robust (strong, healthy)and most likely will not change excessively(exceptionally). Keep this beta status in mind as you read through this book and as you develop your own Java programs.

Support for playing Java programs is a little more confusing at the moment. Sun’s HotJava is not currently included with the Beta JDK; the only available version of HotJava is an older alpha version, and, tragically, applets written for the alpha version of Java do not work with the beta JDK, and vice versa. By the time you read this, Sun may have released a newer version of HotJava which will enable you to view applets.

The JDK does include an application called applet viewer that allows you to test your Java applets as you write them. If an applet works in the applet viewer, it should work with any Java-capable browser.

***JAVA IS PLATFORM-INDEPENDENT***

Platform independence is one of the most significant advantages that Java has over other programming languages, particularly for systems that need to work on many different platforms. Java is platform-independent at both the source and the binary level.

***Platform-independence*** is a program’s capability of moving easily from one computer system to another.

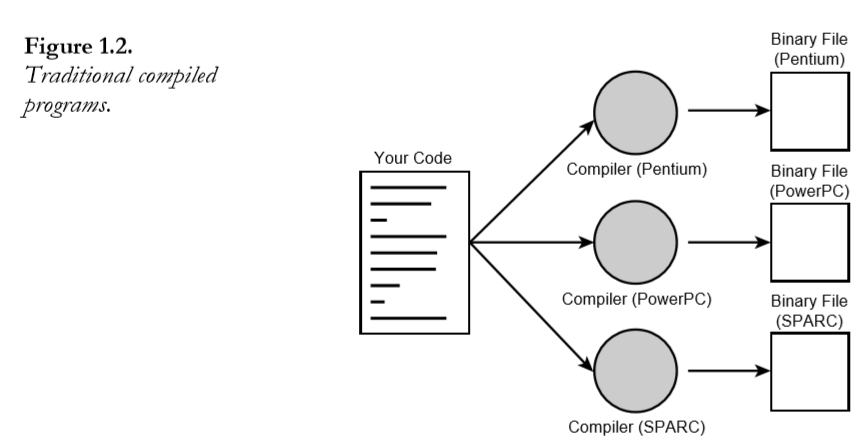
At the source level, Java’s primitive data types have consistent sizes across all development platforms. Java’s foundation class libraries make it easy to write code that can be moved from platform to platform without the need to rewrite it to work with that platform.

Platform-independence doesn’t stop at the source level, however. Java binary files are also platform-independent and can run on multiple problems without the need to recompile the source. How does this work? Java binary files are actually in a form called bytecodes.

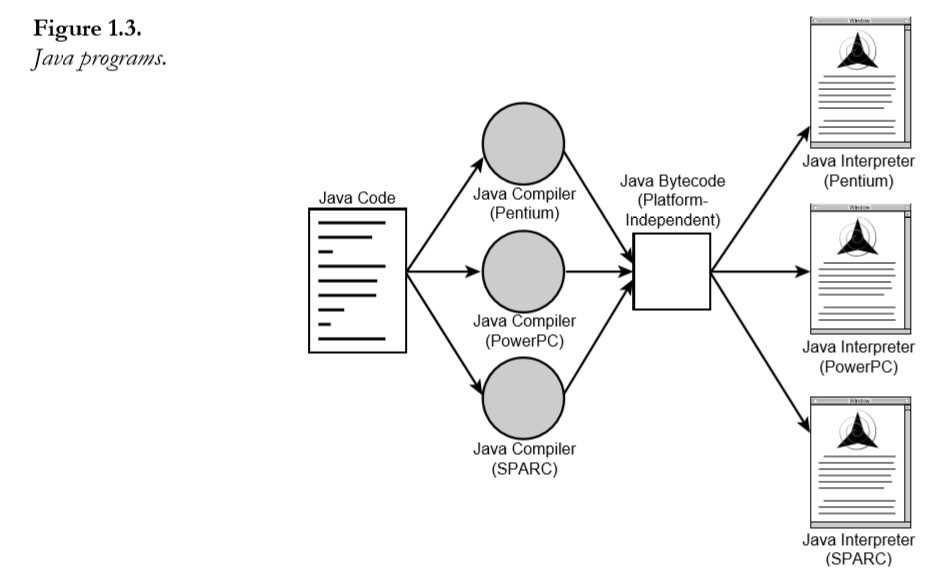
***Bytecodes*** are a set of instructions that looks a lot like some machine codes, but that is not specific to any one processor.

Normally, when you compile a program written in C or in most other languages, the compiler translates your program into machine codes or processor instructions. Those instructions are specific to the processor your computer is running—so, for example, if you compile your code on a Pentium system, the resulting program will run only on other Pentium systems. If you want to use the same program on another system, you have to go back to your original source, get a compiler for that system, and recompile your code. Figure 1.2 shows the result of this system: multiple executable programs for multiple systems.

Things are different when you write code in Java. The Java development environment has two parts***: a Java compiler*** and a ***Java interpreter***. The Java compiler takes your Java program and instead of generating machine codes from your source files, it generates bytecodes.



To run a Java program, you run a program called a bytecode interpreter, which in turn executes your Java program. You can either run the interpreter by itself, or—for applets— there is a bytecode interpreter built into HotJava and other Java-capable browsers that runs the applet for you.



Why go through all the trouble of adding this extra layer of the bytecode interpreter? Having your Java programs in bytecode form means that instead of being specific to any one system, your programs can be run on any platform and any operating or window system as long as the Java interpreter is available. This capability of a single binary file to be executable across platforms is crucial to what enables applets to work, because the World Wide Web itself is also platform independent. Just as HTML files can be read on any platform, so applets can be executed on any platform that is a Java-capable browser.

The disadvantage of using bytecodes is in execution speed. Because system-specific programs run directly on the hardware for which they are compiled, they run significantly faster than Java bytecodes, which must be processed by the interpreter. For many Java programs, the speed may not be an issue. If you write programs that require more execution speed than the Java interpreter can provide, you have several solutions available to you, including being able to link native code into your Java program or using tools to convert your Java bytecodes into native code. Note that by using any of these solutions, you lose the portability that Java bytecodes provide.

***Java Is Object-Oriented***

To some, object-oriented programming (OOP) technique is merely a way of organizing programs, and it can be accomplished using any language. Working with a real object-oriented language and programming environment, however, enables you to take full advantage of object oriented methodology and its capabilities of creating flexible, modular programs and reusing code.

Many of Java’s object-oriented concepts are inherited from C++, the language on which it is based, but it borrows many concepts from other object-oriented languages as well. Like most object-oriented programming languages, Java includes a set of class libraries that provide basic data types, system input and output capabilities, and other utility functions. These basic classes are part of the Java development kit, which also has classes to support networking, common Internet protocols, and user interface toolkit functions. Because these class libraries are written in Java, they are portable across platforms as all Java applications are.

***Java Is Easy to Learn***

In addition to its portability and object-orientation, one of Java’s initial design goals was to be small and simple, and therefore easier to write, easier to compile, easier to debug, and, best of all, easy to learn. Keeping the language small also makes it more robust because there are fewer chances for programmers to make difficult-to-find mistakes. Despite its size and simple design, however, Java still has a great deal of power and flexibility

**GETTING STARTED WITH PROGRAMMING IN JAVA**

Enough background! Let’s finish off this day by creating two real Java programs: a stand-alone Java application and an applet that you can view in either in the applet viewer (part of the JDK) or in a Java-capable browser. Although both these programs are extremely simple, they will give you an idea of what a Java program looks like and how to compile and run it.

**APPLETS AND APPLICATIONS**

Java applications fall into two main groups: ***applets*** and ***applications.***

***Applets***, as you have learned, are Java programs that are downloaded over the World Wide Web and executed by a Web browser on the reader’s machine. Applets depend on a Java-capable browser in order to run (although they can also be viewed using a tool called the applet viewer

***Java applications*** are more general programs written in the Java language. Java applications don’t require a browser to run, and in fact, Java can be used to create most other kinds of applications that you would normally use a more conventional programming language to create. HotJava itself is a Java application.

A single Java program can be an applet or an application or both, depending on how you write that program and the capabilities that program uses.

***Creating a Java Application***

Let’s start by creating a simple Java application: the classic Hello World example that all language books use to begin.

**Listing 1.1. Your first Java application.**

package com.codewithamedee;  
  
public class Main {  
  
 public static void main(String[] args) {  
 System. *out*. println("hello world");  
 }  
}

This program has two main parts:

■ All the program is enclosed in a class definition—here, a class called HelloWorld.

■ The body of the program (here, just the one line) is contained in a routine called main(). In Java applications, as in a C or C++ program, main() is the first routine(function) that is run when the program is executed.

Once you finish typing the program, save the file. Conventionally, Java source files are named the same name as the class they define, with an extension of .java. This file should therefore be called HelloWorld.java.

Now, let’s compile the source file using the Java compiler. In Sun’s JDK, the Java compiler is called javac.

To compile your Java program, Make sure the javac program is in your execution path and type javac followed by the name of your source file:

javac HelloWorld.java

*CHAPTER 2OBJECT-ORIENTED PROGRAMMING AND JAV*

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

**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

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

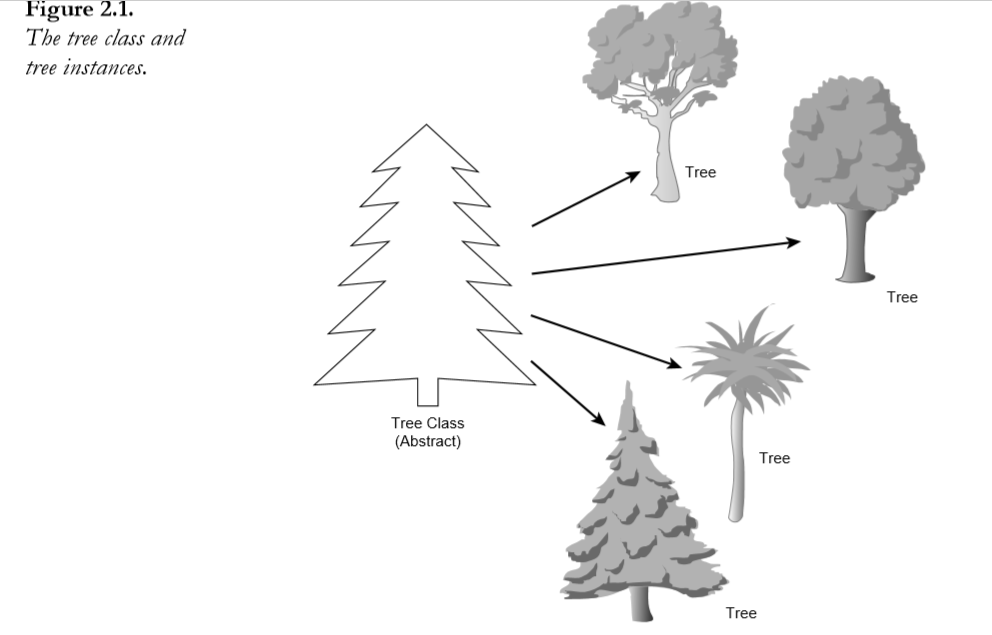
When you write a program in an object-oriented language, you don’t define actual objects. You define classes of objects.

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

*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.



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.

***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.

***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.

***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

***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).

***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.

package com.codewithamedee;  
  
public class Motocycle {  
   
}

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:

package com.codewithamedee;  
  
public class Motocycle {  
 public String makes;  
 public String color;  
 public 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.

***boolean*** in Java is a real data type that can have the value true or false. Unlike C, booleans are not numbers.

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:

package com.codewithamedee;  
  
public class Motocycle {  
 public String makes;  
 public String color;  
 public Boolean EngineState;  
 void StartEngine(){  
 if (EngineState == true)  
 System. *out* .println("The engine is arleady 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:

package com.codewithamedee;  
  
public class Motocycle {  
 public String makes;  
 public String color;  
 public Boolean EngineState;  
 void StartEngine(){  
 if (EngineState == true)  
 System. *out* .println("The engine is aleady on");  
 else{  
 EngineState = true;  
 System . *out*. println("The engine is now on");  
 }  
 void ShowAtts(){  
 System. *out*. println("This motocycle is a" + color + " " + makes);  
 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.

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).

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

package com.codewithamedee;  
public class Main {  
 public static void main(String[] args) {  
 Motocycle m = new Motocycle();  
 m.makes = "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("calling 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: 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