Excerpt from

Introduction to Java

The following is a sample excerpt from a study unit converted into the Adobe Acrobat format. A sample online exam is available for this excerpt.

The study unit defines and explains object-oriented programming. You'll learn how to use the exciting object-oriented Java language to create programs, interfaces, and applications. You'll also learn the history of the language and be introduced to some conventions that will be used throughout your study of Java.

After reading through the material in this excerpt, feel free to take the sample exam based on this material.

INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING IN JAVA

Welcome to the exciting world of programming in Java. You're about to learn one of the most widely used computer programming languages. The Java platform is a new way of computing that's designed around the power of networks. It's designed based on the idea that the same software should run on many different kinds of computers, consumer electronics, and other devices.

The History of Java

In the early 1990s, Sun Microsystems established the Green Team. The team's task was to study many common electronic objects—such as stereos, televisions, microwave ovens, remote controls, even automobile computers—and try to establish a single programming language that could be used to operate all of these devices.

Since they wanted to emulate real-world objects, it was obvious that they should use an object-oriented language, such as C++. The drawbacks to using a complex language, such as C++, were the size and processor requirements needed to run such a system. So, the team began working on a new language with the core functionality of C++, but with a much smaller library of functions, allowing it to be run on a much smaller computer.

Sun's development group named this language *Oak*, after a tree one of the team members had outside his window.

They then began an unsuccessful attempt to sell the idea of a portable operating system to various electronics manufacturers. The manufacturers weren't particularly excited by the prospect of making their products compatible with other manufacturers' components, and the language went nowhere.

At about the same time, a company called Oak Technologies decided they didn't want to share their trademarked name with Sun's new product, and they required that Sun change the name of the new language. Popular legend has it that the team decided on the name *Java* after one of their many trips to the coffee shop.

So, there was a new programming language and a new name, but what to do with a small, powerful language that could run on almost anything?

Java is an exciting programming language that allows us to write programs that can be embedded into Internet Web pages (applets), programs that can be run on any Java-enabled computer (applications), and programs that might be used either as an applet or an application.

Users can create interactive pages for Web-based businesses. Online customized catalogs, questionnaires, order forms, e-mail requests, and customer lists are just some of the possibilities. But there's far more to Java than business applications. Programmers can create games, animations, and much more. The ideas are limited only by your imagination.

Important Terms and Concepts

Unlike Java, most current structured programming languages are *compiled* languages, meaning that their raw source code is converted into "machine language" at design time. This process makes the program run very quickly on the user's machine. However, using this "machine language" means that the completed program can be used on only one type of computer system.

It also means that if programmers wanted to make a change to any part of the program, they would have to recompile the entire program to implement the changes.

Java, on the other hand, is an *interpreted* language. This means that the programmer writes the source code on a local computer, and then runs it through a special type of *compiler*.

The Java compiler (known as *javac*) converts the source code not into finished, platform-specific machine language, but instead into binary strings called *bytecodes*. The user then runs the bytecodes through a program known as an *interpreter,* which translates the bytecodes into a form that can be run on the client machine. One of Java's strongest points is that it can be interpreted on the user's computer (Figure 1).

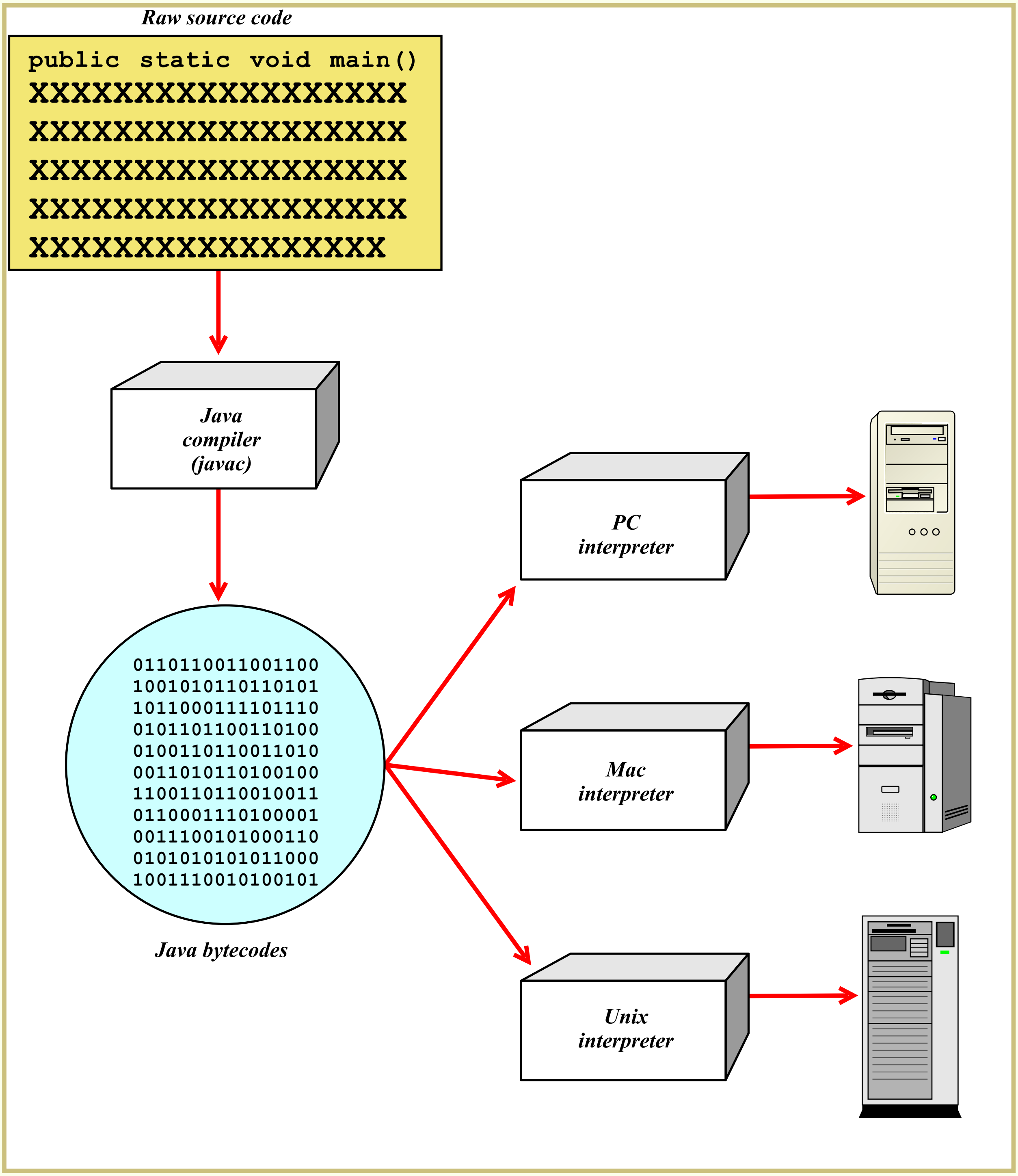

FIGURE 1—As an interpreted language, Java can be run on any system for which a Java interpreter program is available.

Java allows programmers to produce programs that run on any host computer with a Java Virtual Machine (JVM). The JVM reads the bytecodes produced by the java compiler and provides an execution environment for these bytecodes. The JVM provides the interface to the host computer's native operating system, releasing the java program from the details of the specific host computer. A program written on a DOS-based PC platform may be run on a Macintosh, a PC running OS/2 or Windows, a Sun Solaris workstation, or even on a mainframe computer under UNIX.

Since the World Wide Web and Internet consist of computers of many different types, you can see how Java's versatility would be useful. With Java, a program needs to be written only once before everyone else on the Internet can can start using it, regardless of the type of operating systems involved.

A Look at Programming

Java is an object-oriented programming language. What is object-oriented programming, and how is it different from other kinds of programming? Currently, there are two major styles of programming. The traditional style of programming used in languages such as BASIC, FORTRAN, COBOL, and C is known as procedural, or structural, programming. Small-talk, C++, and Java are object-oriented languages. Before learning more about Java and object-oriented languages, let's take a look at how traditional programming is done.

Procedural Programming

Many command-line languages use procedural programming. In these languages, each line of the program tells the computer to do something: open a file, get an input, perform an action, write an output, display the output on the screen, and so on. The program is simply a set of instructions to be carried out one at a time by the computer. This might be an ideal way to write a small program where there are few variables to consider, or where there aren't many steps to complete.

In the real world, however, programs are often called on to perform a number of complex tasks. It would be very difficult for a programmer to keep track of more than perhaps a few hundred lines of coding. However, some larger tasks within a program might each require hundreds of lines per task. For this reason, programs are broken down into sections, with the complex tasks standing on their own. Depending on the language being used, these sections are called *subprograms, procedures,* or *subroutines*.

In procedural programming, the subprograms can be combined into groups called *modules*. For example, let's look at a program that calculates payroll for a business. The core program might request an employee name and an entry for how many hours that employee worked during a given period. This request could be called an *input module*.

Internally, the program can then compare the hours to a table to see how many hours are to be paid at regular time, and how many should be paid at an overtime rate. (This could be called a *rates module*.) The program can then calculate the gross pay based on those calculations (*gross module*).

Next, a *pretax module* can subtract any special withholdings from the gross pay before the taxes are calculated. The *tax module* will then calculate any taxes based on the adjusted gross pay. Yet another module (the *net module*) will tally all of the withholdings and subtract them from the gross pay.

There might be a *year-to-date module* that takes these totals and sends them to a database where all of the figures for the entire year are held. Finally, the finished pay statement is sent to the screen, where the operator can then send it to yet another module, which will lay out the numbers in a particular order and print the paycheck.

Although this might seem like a complicated process, it's really a very simple version of what actually happens. In fact, there may be hundreds of modules in a typical program.

The Limitations of Structured Programming

When programmers start to deal with many hundreds of modules in a program, problems will inevitably creep in. First, the program itself becomes far too complex for just one person to write and maintain. So, there are often different programmers programming different modules, and they might not all be making use of the existing modules in quite the same way.

For example, Programmer A uses a value from the input module to arrive at some intermediate value. Programmer B, working in a different module, needs the same value that Programmer A's module provides; however, Programmer B doesn't realize that this value has already been calculated, and so B recalculates it within his or her module. Now there's a problem. The same variable, possibly calculated two different ways, now exists in the program in two different places at the same time. Programmer C now calls for the variable in another module. The computer does what it's told and sends *both* answers to the third module at the same time. Programmer C might expect the variable calculation as defined in Programmer A's module but inadvertently get the calculation from Programmer B's module, which can introduce subtle errors that are difficult to correct.

The type of data that was requested is referred to as *global data,* because it's available to every part of the program all of the time. The *data crashes* occur because each module has unrestricted access to the data from every other module within the program.

The Real World

The second problem with procedural programming is that it doesn't apply well to all real-world applications. In the real world, we have objects. An object is something tangible, such as a kitten or a motorcycle. Kittens and motorcycles are each much more complicated than a single program module. Remember that each module can complete only one task, in a particular order. Anyone who has ever owned a kitten can tell you that one task, in one particular order, is *not* the way a kitten works!

In the real world, objects have both *characteristics* and *behaviors*. The kitten has characteristics such as fur color, eye color, number of legs, and so on. It has behaviors such as running, jumping, purring, sleeping, and so on.

In the world of programming, each behavior is a response to an external request or command. If we command the motorcycle to run by turning the ignition switch, the engine will start. Pushing the horn button or the headlight switch will call another behavior, but have no effect on the *run* command. The engine-starting behavior never sees the data from the horn button or the headlight switch.

Object-Oriented Programming

The major characteristics of object-oriented programming are *encapsulation, polymorphism,* and *inheritance.* Let's take a look at each of these characteristics now.

Encapsulation. We can gather variables and methods together inside of a class. Instance variables and methods can be added, deleted, or changed, but as long as the functions provided by the object remain the same, any code that uses the object can continue to use it without being rewritten.

- *Data hiding.* All of the code for a class might be hidden from the view of the end user with, no loss of functionality.
- *Abstraction.* A class can be written to hold nothing but a pure design, partial implementations, or instance states; or a class might be written where only the subclasses are expected to be used.

Polymorphism. Polymorphism refers to the fact that different objects respond to the same message differently.

Inheritance. The following are traits of inheritance.

- A new class might be defined in terms of an existing class.
- The new class may have access to all the members and methods of the base class.
- The new class can add new or more detailed methods and state variables.
- Program code might be reused again and again.
- Classes are arranged in a hierarchical order.

The Language of Java

Object-oriented programming has a language of its own. Let's look at some very basic concepts of object-oriented programming, and at the same time examine some of the important terms that you'll use when programming in Java.

What Is a Class?

There are several ways to describe a *class*.

- A class is the definition of an object.
- A class is the blueprint for the object.
- A class is *not* the object itself.
- There can be many objects from a single class.
- An object is an *instance* of a class.

➤tip **You'll notice that in the description below, class names are presented in a different typeface. Later in your material, you'll see that classes within a computer program, as well as other programming elements, such as statements, are also written in this special font.**

The creation of an object from a class is called *instantiation.* For example, let's say we have a class called `Feline`. In this case, `puff`, `sylvester`, or `garfield` might be instances of the class `Feline`. Also, `puff`, `sylvester`, and `garfield` are concrete instances of the blueprint for data and behavior defined by class `Feline`.

A class defines attributes and behavior. Instances of a class all have the same attributes. The attributes may or may not have different values. Class `Feline` may have an attribute called *color.* The color attribute of object `puff` may have a value of *yellow* while object `garfield` may have a color attribute whose value is *orange.* The term *instance variable* refers to attributes that belong to an object, which is an instance of a class.

➤tip

You'll notice that the classes in the example begin with an uppercase letter. This is typical in Java programming, though not necessary to create a workable program.

An Overview of Inheritance

Inheritance is one of the major features of object-oriented programming. Inheritance allows a subclass to be defined in terms of a more general superclass. For example, consider a superclass named `WheeledVehicle`. Superclass `WheeledVehicle` has attributes such as *manufacturer* and *vehicle ID*. We can define a subclass of `WheeledVehicle` called `Truck`. The subclass `Truck` "inherits" the attributes *manufacturer* and *vehicle ID* from the superclass `WheeledVehicle`. Subclass `Truck` may add new attributes such as *tractor type* and *freight capacity*. Instances of subclass `Truck` have all the attributes of subclass `Truck` and the attributes of Superclass `WheeledVehicle`. All classes within a Java program are arranged in a strict order from the top down. This is just a brief overview of inheritance; it will be examined in detail later in your program.