Appendix A

JAVA PRIMER

In this appendix, we present a primer for people who are unfamiliar with the Java language. This introduction is intended to allow you to develop the Java skills necessary to understand the programs in this text. It is not a complete Java reference; for a more thorough coverage of Java, consult the Bibliographical Notes. We do not assume that you are familiar with object-oriented principles, such as classes, objects, and encapsulation, although some knowledge of these topics would be helpful. If you are already familiar with C or C++, the transition to Java will be smooth, because much Java syntax is based on C/C++.

A.1 Basics

In this section, we cover basics such as the structure of a basic program, control statements, methods, classes, objects, and arrays.

A.1.1 A Simple Program

A Java program consists of one or more classes. One class must contain the main() method. (A method is equivalent to a function in C/C++. ) Program execution begins in this main method. Figure A.1 illustrates a simple program that outputs a message to the terminal. The main() method must be defined as public static void main(String args[]), where String args[] are optional parameters that may be passed on the command line. Output to the
/*
The first simple program
*/
public class First {
    public static void main(String args[]) {
        // output a message to the screen
        System.out.println("Java Primer Now Brewing!");
    }
}

Figure A.1 A first Java program.

screen is done via the call to the method System.out.println(). Note that Java supports the C++ style of comments, including

    // This string is a comment

for single-line comments and

    /* And this string
    too is
    a comment */

for multiline comments.

The name of the file containing the main() method must match the class name in which it appears. In this instance, the main() method is in the class First. Therefore, the name of this file must be First.java. To compile this program using the Java Development Kit (JDK) command line compiler, enter

    javac First.java

on the command line. This will result in the generation of the file First.class. To run this program, type the following on the command line

    java First

Note that java commands and file names are case sensitive.

A.1.2 Methods
A program usually consists of other methods in addition to main(). Figure A.2 illustrates the use of a separate method to output the message Java Primer Now Brewing! In this instance, the String is passed to the static method printIt(), where it is printed. Note that both the main() and printIt() methods return void. Later examples throughout this chapter will use methods that return values. We will also explain the meaning of the keywords public and static in Section A.1.6.
public class Second
{
    public static void printIt(String message) {
        System.out.println(message);
    }

    public static void main(String args[]) {
        printIt("Java Primer Now Brewing!");
    }
}

Figure A.2 The use of methods.

A.1.3 Operators
Java uses the + - * / operators for addition, subtraction, multiplication, and division, and the & operator for the remainder. Java also provides the C++ autoincrement and autodecrement operators ++ and --. The statement ++T; is equivalent to T = T + 1. In addition, Java provides shortcuts for all the binary arithmetic operators. For example, the += operator allows the statement T = T + 5; to be written as T += 5.

The relational operators are == for equality and != for inequality. For example, the statement 4 == 4 is true, whereas 4 != 4 is false. Other relational operators include < <= >= >. Java uses && as the and operator and || as the or operator.

Unlike C++, Java does not allow operator overloading. However, you can use the + operator to append strings. For example, the statement

    System.out.println("Can you hear " + "me?");

appends me? to Can you hear.

A.1.4 Statements
In this section, we cover the basic statements for controlling the flow of a program.

A.1.4.1 The for statement
The format of the for loop is as follows:

    for (initialization; test; increment) {
        // body of for loop
    }
For example, the following for loop prints out the numbers from 1 to 25.

```java
int i;
for (i = 1; i <= 25; i++) {
    System.out.println("i = " + i);
}
```

Note that, when the body of the for loop consists of only one line, the braces of the loop can be eliminated. Also, the index of the for loop can be declared within the for statement, rather than prior to the statement. Therefore, the previous loop could have been written as follows:

```java
for (int i = 1; i <= 25; i++)
    System.out.println("i = " + i);
```

When the index is declared in the for loop, the scope of the index is only within the for statement. The index of a for loop must be an integer (int). Java data types are discussed in Section A.1.5.

### A.1.4.2 The while statement

The format of the `while` loop is as follows:

```java
while (boolean condition is true) {
    // body of loop
}
```

The format of the `do-while` loop is as follows:

```java
do {
    // body of loop
} while (boolean condition is true);
```

As is the case for the for loop, if the body of either the `while` or the `do-while` loops is only one statement, the braces can be eliminated.

### A.1.4.3 The if statement

The format of the `if` statement is as follows:

```java
if (boolean condition is true) {
    // statement(s)
}
else if (boolean condition is true) {
    // other statement(s)
}
else {
    // even other statement(s)
}
```
If there is only a single statement to be performed if the Boolean expression is true, then the braces can be eliminated.

### A.1.5 Data Types

Java provides two different varieties of data types: primitive and reference (or nonprimitive). Primitive data types include boolean, char, byte, short, int, long, float, and double. Reference data types include objects and arrays.

### A.1.6 Classes and Objects

Objects are at the heart of Java programming. An object consists of data and methods that access (and possibly manipulate) the data. An object also has a state, which consists of the values for the object's data at a particular time. Objects are modeled with classes. Figure A.3 is a class for a Point. The data

```java
class Point
{
    // constructor to initialize the object
    public Point(int i, int j)
    {
        xCoord = i;
        yCoord = j;
    }

    // exchange the values of xCoord and yCoord
    public void swap()
    {
        int temp;
        temp = xCoord;
        xCoord = yCoord;
        yCoord = temp;
    }

    public void printIt()
    {
        System.out.println("X coordinate = " + xCoord);
        System.out.println("Y coordinate = " + yCoord);
    }

    // class data
    private int xCoord; // X coordinate
    private int yCoord; // Y coordinate
}
```

Figure A.3 A point.
public Point() {
    xCoord = 0;
    yCoord = 0;
}

Figure A.4 Default constructor for the Point class.

for this class are the x and y coordinates for the point. An object—which is an instance of this class—can be created with the new statement. For example, the statement

Point ptA = new Point(0,15);

creates an object with the name ptA of type Point. Furthermore, the state of this object is initialized to x = 0 and y = 15. We initialize objects using a constructor. A constructor looks like an ordinary method except that it has no return value and it must have the same name as the name of the class. The constructor is called when the object is created with the new statement. In the statement Point ptA = new Point(0,15);, the values 0 and 15 are passed as parameters to the constructor, where they are assigned to the data xCoord and yCoord.

With the Point class written the way it is, we could not create a point using the statement

Point ptA = new Point();

because there is no constructor that has an empty parameter list. If we wanted to allow the creation of such an object, we have to create a matching constructor. For example, we could add the constructor shown in Figure A.4 to the class Point. Doing so allows us to create an object using the statement Point ptA = new Point();. In this case, the x and y coordinates for the point are initialized to 0.

Having two constructors in the Point class leads to an interesting issue. In some ways, a constructor behaves like an ordinary method. Two constructors for this class essentially means two methods with the same name. This feature of object-oriented languages is known as method overloading. A class may contain one or more methods with the same name, as long as the data types in the parameter list are different.

The Point class has two other methods: swap() and printIt(). Note that these methods are not declared as static; unlike the printIt() method in Figure A.2. These methods can be invoked with the statements

ptA.swap();
ptA.printIt();

The difference between static and instance (nonstatic) methods is that static methods do not require being associated with an object, and we can call them by merely invoking the name of the method. Instance methods can be invoked
only by being associated with an instance of an object. That is, they can be called with only the notation object-name.method. Also, an object containing the instance methods must be instantiated with new before its instance methods can be called. In this example, we must first create ptA to call the instance methods swap() and printIt().

Also notice that we declare the methods and data belonging to the class Point using either the public or private modifiers. A public declaration allows the method or data to be accessed from outside the class. Private declaration allows the data or methods to be accessed from only within the class. Typically, we design classes by declaring class data as private. Thus, the data for the class are accessible through only those methods that are defined within the class. Declaring data as private prevents the data from being manipulated from outside the class. In Figure A.3, only the swap() and printIt() methods can access the class data. These methods are defined as public, meaning that they can be called by code outside the class. Therefore, the designer of a class can determine how the class data are accessed and manipulated by restricting access to private class data through public methods. (In Section A.4, we look at the protected modifier.)

Note that there are instances where data may be declared as public and methods as private. Data constants may be declared as public if access from outside the class is desired. We declare constants using the keywords static final. For example, a class consisting of mathematical functions may publicly declare the constant pi as

public static final double PI = 3.14159;

Also, if a method is to be used only within the class, it is appropriate to define that method as private.

Objects are treated by reference in Java. When the program creates an instance of an object, that instance is assigned a reference to the object. Figure A.5 shows two objects of class Point: ptA and ptB. These objects are unique; each has its own state. However, the statement

Point ptC = ptB;

assigns ptC a reference to object ptB. In essence, both ptB and ptC now refer to the same object, as shown in Figure A.6. The statement

ptC.swap();

alters the value of this object, calling the swap() method, which exchanges the values of xCoord and yCoord. Calling the printIt() method for ptB and ptC illustrates the change in the state for the object, showing that now the x coordinate is -25 and the y coordinate is -15.

Whenever objects are passed as parameters to methods, they are passed by reference. Thus, the local parameter is a reference to the argument being passed. If the method alters the state of the local parameter, it also changes the state of
public class Third
{
    public static void main(String args[]) {
        // create 2 points and initialize them
        Point ptA = new Point(5, 10);
        Point ptB = new Point(-15, -25);

        // output their values
        ptA.printIt();
        ptB.printIt();

        // now exchange values for first point
        ptA.swap();
        ptA.printIt();

        // ptC is a reference to ptB
        Point ptC = ptB;
        // exchange the values for ptC
        ptC.swap();

        // output the values
        ptB.printIt();
        ptC.printIt();
    }
}

Figure A.5  Objects as references.

the argument that it was passed. In Figure A.7, the object ptD is passed to the
method change(), where the swap() method is called. When they return from
change(), the values of the x and y parameters for ptD are exchanged.

The Java keyword this provides an object a reference to itself. The this
reference is useful when an object needs to pass a reference of itself to another
object.

ptA
xCoord=5
yCoord=10

ptB
xCoord=-15
yCoord=-25

ptC

pt C is a reference to pt B.

Figure A.6  ptC is a reference to ptB.
public static void change(Point tmp) {
    tmp.swap();
}

Point ptD = new Point(0,1);
change(ptD);
ptD.printIt();

Figure A.7  Object parameters are passed by reference.

A.1.7 Arrays

Arrays are reference data types in Java; thus, they are created just like any other type of object. The syntax for creating an array of 10 bytes is

    byte[] numbers = new byte[10];

Arrays can also be created and initialized in the same step. The following code fragment initializes an array to the first five prime numbers.

    int[] primeNums = {3, 5, 7, 11, 13};

Note that neither is the size of the array specified nor is the new statement used when an array is initialized.

When we create an array of objects, we must first allocate the array with the new statement and we must allocate each object using new. The statement

    Point[] pts = new Point[5];

creates an array to hold five references to Point objects. The statements

    for (int i = 0; i < 5; i++) {
        pts[i] = new Point(i,i);
        pts[i].printIt();
    }

create the five objects, assigning a reference to each object to the appropriate array element.

Java performs default array initialization. Primitive numeric data types are initialized to zero, and arrays of objects are initialized to null.

A.1.8 Packages

Many of the core classes for Java are gathered into packages. We do not teach you how to create packages, but we do show how to use the different packages that constitute the core Java application programming interface (API). A pack-
age is simply a set of related classes. The naming convention for the core Java packages is `java.<package name>.<class name>`. The standard package for the core API is `java.lang`. Many of the more commonly used classes belong to this package, including `String` and `Thread`. Therefore, the fully qualified names for these classes are `java.lang.String` and `java.lang.Thread`. Classes that belong to `java.lang` are so common that we can refer to them by their class name, omitting the preceding package name. However, there are many other classes that belong to packages other than `java.lang`, and they do not follow this rule. In these instances, the fully qualified package name must be used. For example, the `Vector` class belongs to the package `java.util` (we look at vectors in Section A.3). To use a `Vector`, therefore, we must use the following syntax

```java
class Vector items = new java.util.Vector();
```

This requirement obviously can lead to cumbersome syntax. The less verbose solution is for the program to declare that it is using a class in a certain package by issuing the following statement at the top (before any class definitions.)

```java
import java.util.Vector;
```

Now, we could declare the `Vector` using the easier style:

```java
Vector items = new Vector();
```

However, the most common style people use when they are using packages is

```java
import java.util.*;
```

This approach allows the program to use any class in the package `java.util`. The JavaSoft web site (http://www.jassoft.com) includes a complete listing of all packages in the core API.

## A.2 Exception Handling

An exception is an occurrence of an unusual event during the operation of a program. Java allows a program to catch exceptions and to handle them in a graceful manner. When an exception occurs within a method, that method can throw the exception to the code that called it. For example, the API for the `wait()` method for the class `Object` appears as

```java
public final void wait() throws InterruptedException;
```

Thus, calling the `wait()` could result in an `InterruptedException` occurring. Exceptions can be handled by being caught. The syntax for catching and handling exceptions is as follows:
try {
    // call some method(s) that
    // may result in an exception.
} catch (theException e) {
    // now handle the exception
} finally {
    // perform this whether the
    // exception occurred or not.
}

All code that may cause an exception is enclosed within a try block. If an exception that is specified in the catch statement occurs, the exception is caught and handled within the catch block. (When an exception is thrown, any following code in the try block is not executed.) An optional finally clause allows the program to run code whether or not the exception occurred. The program shown in Figure A.8 illustrates the use of these statements. This program generates randomly either 0 or 1 using the static method random()

```java
public class TestExcept {
    public static void main(String args[]) {
        int num, recip;

        // generate a random number 0 or 1
        // random() returns a double, type-cast to int
        num = (int) (Math.random() * 2);

        try {
            recip = 1 / num;
            System.out.println("The reciprocal is "+ recip);
        } catch (ArithmeticException e) {
            // output the exception message
            System.out.println(e);
        } finally {
            System.out.println("The number was " + num);
        }
    }
}

Figure A.8 Testing exceptions.
from the class \texttt{java.lang.Math}, and produces the reciprocal of this number. If this random number is 0, an arithmetic exception for divide by 0 is thrown. This exception is handled by outputting the exception message. The \texttt{finally} clause prints out the random number that was generated.

If an exception is not caught, it propagates up through the call stack until it reaches the \texttt{main()} method. For example, one such exception is an array out of bounds (\texttt{java.lang.ArrayIndexOutOfBoundsException}). The following code example illustrates how such an exception could occur:

\begin{verbatim}
int[] nums = {5,10,15,20,25};

for (int i = 0; i < 6; i++)
    System.out.println(nums[i]);
\end{verbatim}

Since this exception is not caught within a try-catch block, it propagates all the way up through the original call to \texttt{main()}, where it causes program termination and an error message.

### A.3 Inheritance

Java is considered a pure object-oriented language. A feature of such languages is that they are able to extend the functionality of an existing class. In object-oriented terms, this extension is inheritance. In general, it works as follows.

\begin{verbatim}
public class Person
{
    public Person(String n, int a, String s) {
        name = n;
        age = a;
        ssn = s;
    }

    public void printPerson() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        System.out.println("Social Security: " + ssn);
    }

    private String name;
    private int age;
    private String ssn;
}
\end{verbatim}

\textbf{Figure A.9} A person.
A.3 Inheritance

Assume that there exists a class that has almost all the functionality that is needed for a particular application. Rather than developing from scratch a new class that contains all the necessary functionality, we can use inheritance to add to the existing class whatever is needed. For example, assume that an application must keep track of student records. A student is distinguished by her name, age, social-security number, major, and grade-point average (GPA). If another class already exists that represents people using name, age, and social-security number, we can create a student class by extending it, adding a major and GPA. The class being inherited from is the base class; the extended class is the derived class.

The code for the Person class is shown in Figure A.9. The class Student is shown in Figure A.10. What is new in this class definition is the keyword extends. The Java syntax for a derived class to inherit from a base class is

```
derived-class extends base-class
```

Also note that the constructor in the Student class has a call to super(n, a, s). When an object of class Student is created, the constructor for the base class is not normally called. By invoking the super() method, the derived class can call the constructor in the base class.

```java
public class Student extends Person {

    public Student(String n, int a, String s,
                    String m, double g) {
        // call the constructor in the
        // parent (Person) class
        super(n, a, s);
        major = m;
        GPA = g;
    }

    public void studentInfo() {
        // call this method in the parent class
        printPerson();

        System.out.println("Major: " + major);
        System.out.println("GPA: " + GPA);
    }

    private String major;
    private double GPA;
}
```

Figure A.10 A student.
Java makes extensive use of inheritance throughout the core APL. For example, many of the classes in the packages java.awt (used for creating graphical programs) and java.io (provides I/O facilities) are part of an inheritance hierarchy. In addition, one of the ways of creating a thread (described in Chapter 5) is to define a class that extends the Thread class.

By default, all Java classes are derived from a common class called Object. We can exploit this feature by using the Vector class from the package java.util. A vector is a collection class: It contains a collection of objects. A vector behaves like a dynamic array: It grows and shrinks as needed. The method

```java
addElement(Object obj)
```

of the Vector class inserts the Object obj at the end of the vector. Since all classes are derived from Object, this allows us to insert any class into a vector. The following statements illustrate how to insert three separate strings into a vector:

```java
String first = "The first element";
String second = "The second element";
String third = "The third element";

// insert the strings into the vector
Vector items = new Vector();
items.addElement(first);
items.addElement(second);
items.addElement(third);
```

Note that, although these objects are String objects being inserted into the vector, they could be objects of any class.

Retrieving objects from a vector works similarly. The `elementAt()` method returns the Object at a specified index in the vector, as shown in the following statements.

```java
String element;
element = (String) items.elementAt(0);
System.out.println(element);
element = (String) items.elementAt(1);
System.out.println(element);
element = (String) items.elementAt(2);
System.out.println(element);
```

Since the return type for this method is Object, the value being returned must be typecast to its appropriate data type. The ability to create collection classes—such as vectors, hashtables, stacks, and queues—that contain the type Object allows a programmer to provide a generic data structure that works with all objects.
A.4 Interfaces and Abstract Classes

Another feature of pure object-oriented languages is the ability to define the behavior of an object without specifying how that behavior is to be implemented. Java gives us this ability through interfaces and abstract classes.

In Java, an interface is similar to a class definition, except that it only contains a list of methods and their parameters while omitting the implementation of each method. Furthermore, interfaces may not contain any class data apart from constants. Figure A.11 provides an example of an interface for a shape. This interface merely specifies two methods that are common to shapes—the

```java
public interface Shape
{
    public double area();
    public double circumference();
}

public static final double PI = 3.14159;
```

Figure A.11 Shape interface.

```java
public class Circle implements Shape
{
    // initialize the radius of the circle
    public Circle(double r) {
        radius = r;
    }

    // calculate the area of a circle
    public double area() {
        return PI * radius * radius;
    }

    // calculate the circumference of a circle
    public double circumference() {
        return 2 * PI * radius;
    }

    private double radius;
}
```

Figure A.12 Circle implementation of Shape interface.
public class Rectangle implements Shape {
    public Rectangle(double h, double w) {
        height = h;
        width = w;
    }

    // calculate the area of a rectangle
    public double area() {
        return height * width;
    }

    // calculate the circumference of a rectangle
    public double circumference() {
        return 2 * (height + width);
    }

    private double height;
    private double width;
}

Figure A.13  Rectangle implementation of Shape interface.

area and circumference for the shape. Note that these methods are not defined; it is up to the class that implements this interface to define the body of these methods.

    public class TestShapes {
        public static void display(Shape figure) {
            System.out.println("The area is " + figure.area());
            System.out.println("The circumference is " + figure.circumference());
        }

        public static void main(String args[]) {
            Shape figOne = new Circle(3.5);
            display(figOne);

            figOne = new Rectangle(3,4);
            display(figOne);
        }
    }

    Figure A.14  Program to test the Circle and Rectangle shapes.
Two possible shapes that could implement this interface are a circle and rectangle. The implementation of the Shape interface for a circle is shown in Figure A.12 (on page 775).

When a class implements an interface, it must define the methods as specified by that interface. In this case, the Circle class must implement the area() and circumference() methods. The implementation of the interface for a rectangle is shown in Figure A.13. Since the area and circumference are determined differently for each Shape, the implementation of these methods is left up to classes that implement the interface.

Figure A.14 illustrates how we can use these classes. The interesting feature of this program is that the object reference figOne is declared as type Shape and is first used as a Circle object and next as a Rectangle. We are allowed to do this because the Circle and Rectangle classes both implement the Shape interface. Thus, they are both shapes. Thus, any object declared as type Shape CAN be an instance of any class that implements the Shape interface. Polymorphism is the ability to take on more than one form; in this example, figOne is considered a polymorphic reference, because it is first an instance of a Circle and then an instance of a Rectangle.

The methods shown in the Shape interface are considered to be abstract, in that they are not defined. In fact, we could have put the keyword abstract as

```java
public abstract class Employee
{
    public Employee(String n, String t, double s) {
        name = n;
        title = t;
        salary = s;
    }

    public void printInfo() {
        System.out.println("Name: "+name);
        System.out.println("Title: "+title);
        System.out.println("Salary: "+salary);
    }

    public abstract void computeRaise();
    private String name;
    private String title;
    protected double salary;
}
```

Figure A.15  Abstract class for an employee.
public class Manager extends Employee {
    public Manager(String n, String t, double s) {
        super(n, t, s);
    }

    public void computeRaise() {
        salary += salary * .05 + BONUS;
    }

    private static final double BONUS = 2500;
}

Figure A.16 A manager.

follows, when we defined the interface:

    public abstract double area();
    public abstract double circumference();

The use of the keyword abstract is optional in an interface. If it is absent, it is implied.

Another special type of class is an abstract class. An abstract class is similar to an interface, but it may contain both abstract methods and defined methods. In an interface, all methods must be abstract. Figure A.15, shown on page 777, illustrates an abstract class to model an employee.

To use an abstract class, we extend the class using inheritance, as described in Section A.3. Figure A.16 provides an example of a class that extends Employee for managers. This class must implement the abstract method com-

public class Developer extends Employee {
    public Developer(String n, String t, double s, int np) {
        super(n, t, s);
        numOfPrograms = np;
    }

    public void computeRaise() {
        salary += salary * .05 + numOfPrograms * 25;
    }

    private int numOfPrograms;
}

Figure A.17 A developer.
Employee[] worker = new Employee[3];

worker[0] = new Manager("Pat", "Supervisor", 30000);
worker[1] = new Developer("Tom", "Java Tech", 28000, 20);
worker[2] = new Developer("Jay", "Java Intern", 26000, 8);

for (int i = 0; i < 3; i++) {
    worker[i].computeRaise();
    worker[i].printInfo();
}

Figure A.18 Use of the Manager and Developer classes.

puteRaise(). We calculate the raise for managers by giving them a 5% salary increase in addition to a bonus. Figure A.17 illustrates a class for developers. Program developers are given a salary raise of 3% plus $25 for every program that they write. Figure A.18 shows an example that uses the Manager and Developer classes.

In Section A.1.6, we discussed what the private and public modifiers are and how we can use them to control access to data and methods from outside a class definition. However, the salary data in the Employee abstract class have been declared as protected. The protected modifier allows data to be accessed from within the class in which they are defined, in addition to any classes that are derived from that class. This scheme prevents access to data from code outside the class definition, but allows subclasses to access the data.

Neither interfaces nor abstract classes can be instantiated. Objects can be created only from classes that either implement the interface or extend the abstract class.

In addition to using inheritance, the core Java API also makes heavy use of interfaces and abstract classes. For example, event handling (for mouse clicks, key presses, and so on) in the package java.awt.event is done through implementation of interfaces. Also, a second way of creating threads (described in Section 5.6) implements the Runnable interface.

A.5 Applications and Applets

A Java program can be either a standalone application or an applet. The programs that we have shown thus far in this primer are standalone applications. An applet is a Java program that runs embedded within a web page. Because an applet is part of a web page, running an applet requires using either a web browser or the software appletviewer that comes with the Java Development Kit (JDK). Because this text deals primarily with applications, this primer has focused on the development of standalone applications. We cover the structure of applets briefly in this section.