Java by Dissection

2nd Edition

Ira Pohl

Charlie McDowell

University of California, Santa Cruz

ISBN 978-1-4116-5238-5

TABLE OF CONTENTS

PREFACE

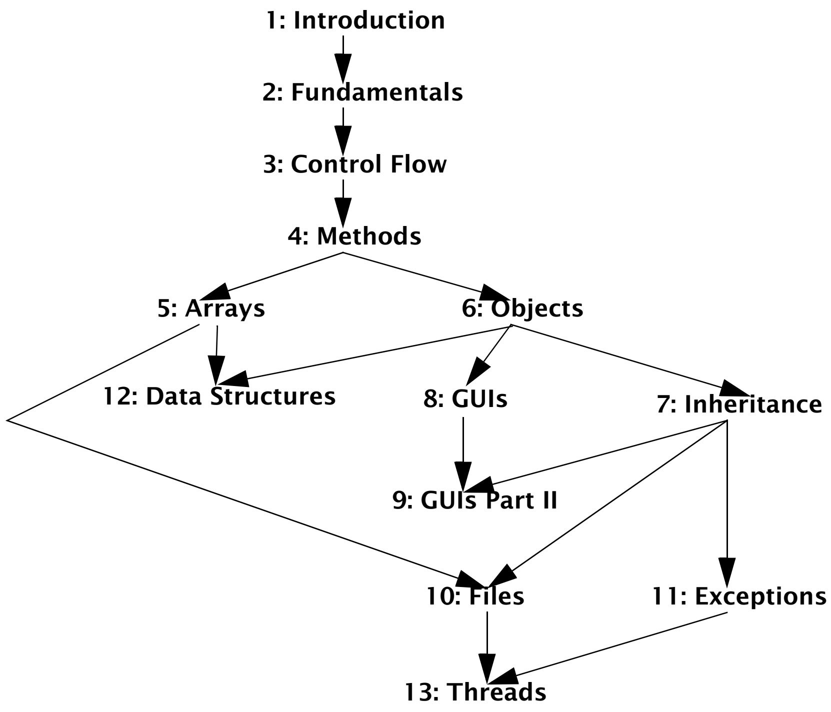
Java by Dissection:2ed is an introduction to programming in Java that assumes no prior programming experience. It thoroughly teaches modern programming techniques using Java with generics. It shows how all the basic data types and control statements are used traditionally. The book then progresses to the object-oriented features of the language and their importance to program design. As such the first half of the book can be used in a first programming course for college students. By careful choice of exercises and supplemental material it could be used for a gentle introduction to programming for computer science or engineering majors also.

The second half of the book explains in detail much that is sophisticated about Java, such as its threading, exception handling, graphical user interface (GUI), generics, and file manipulation capabilities. As such the book is suitable as the primary text in an advanced programming course, or as a supplementary text in a course on data structures, software methodology, comparative languages, or other course in which the instructor wants Java to be the language of choice.

Java, invented at Sun Microsystems in the mid-1990s, is a powerful modern successor language to C and C++. Java, like C++, adds to C the object-oriented programming concepts of *class*, *inheritance* and run-time type binding. The class mechanism provides user-defined types also called *abstract data types*. While sharing many syntactic features with C and C++, Java adds a number of improvements, including automatic memory reclamation called *garbage collection*, bounds checking on arrays, and strong typing. In addition, the standard Java libraries, called *packages* in Java, provide platform independent support for distributed programming, multi-threading and graphical user interfaces.

Although Java shares many syntactic similarities to C, unlike C++, Java is not a superset of C. This has allowed the creators of Java to make a number of syntactic improvements that make Java a much safer programming language than C. As a result, Java is much better as a first programming language.

Java by Dissection:2ed begins with a classical programming style starting with programs as a simple sequence of instructions, then adding in control flow and functional abstraction. After that comes arrays and data abstraction using classes, which can be covered in either order - arrays first, or data abstraction with classes first. Then comes the material on inheritance and graphical user interfaces. Again, the chapter on inheritance can be covered before or after the first chapter on graphical user interfaces. Finally come the advanced chapters. The following figure illustrates the flexibility in the order in which the material can be covered.



The book emphasizes working code. One or more programs particularly illustrative of the chapter's themes are analyzed by dissection, which is similar to a structured walk-through of the code. Dissection explains to the reader newly encountered programming elements and idioms.

Because Java includes a relatively easy-to-use, standard package for creating graphical user interfaces, it is possible to introduce the use of GUIs in a beginning programming book. Creating programs with GUIs is as fundamental today as being able to create nicely formatted text output. To fully understand the GUI packages in Java, it is necessary to have some understanding of OOP and inheritance. The main chapters on GUI building immediately follow the chapters on objects and inheritance. For those students interested in getting an early exposure to some of the graphical aspects of Java, we have provided a series of extended exercises at the end of the early chapters, that introduce GUIs and applets. These exercises provide templates for some simple applets, without providing complete explanations of some of the language features required.

The following summarizes the primary design features that are incorporated into this book.

Dissections. Each chapter has several important example programs. Major elements of these programs are explained by dissection. This step-by-step discussion of new programming ideas helps the reader encountering these ideas for the first time to understand them. This methodology has been proven effective in numerous programming books since its first use in *A Book on C* in 1984.

Teaching by Example. The book is a tutorial that stresses examples of working code. Right from the start the student is introduced to full working programs. Exercises are integrated with the examples to encourage experimentation. Excessive detail is avoided in explaining the larger elements of writing working code. Each chapter has several important example programs. Major elements of these programs are explained by dissection.

Object-Oriented Programming (OOP). The reader is led gradually to an understanding of the object-oriented style. Objects as data values are introduced in Chapter 2, *Program Fundamentals* . This allows an instructor partial to the objects early approach to emphasize OO from the start. Chapter 6, *Objects: Data Abstraction*, shows how the programmer can benefit in important ways from Java and object-oriented programming. Object-oriented concepts are defined, and the way in which these concepts are supported by Java is introduced. Chapter 7, *Inheritance*, develops inheritance and dynamic method dispatch, two key elements in the OOP paradigm.

Terminal input and output. Java has added a `Scanner` class that makes terminal input easy. This replaces our first edition `tio` package which is no longer needed. Also the addition of `printf()` for variable argument fromatted output is discussed.

Container Classes and Generics in Java. The text covers this added feature to Java 5.0. Containers are implemented as generics. Standard containers include Vector, LinkedList, ArrayList and PriorityQueue. These also allow Iterator access and can be used with the for-iterator statement that hides indexing.

Swing and Graphical User Interfaces. An important part of Java is its support for platform independent creation of graphical user interfaces and the web based programs called applets. InChapter 8, *Graphical User Interfaces: Part I* and Chapter 9, *Graphical User Interfaces: Part II*, we present a basic introduction to using the standard Java package Swing, for building GUIs. These chapters provide enough information to create useful and interesting applets and GUIs. A few additional GUI components are presented briefly in Appendix C, *Summary of Selected Swing Components*. For students anxious to begin writing applets, simple applets are introduced in a series of exercises beginning in Chapter 2, *Program Fundamentals*. These exercises are completely optional. The coverage of applets and GUIs in Chapter 8, *Graphical User Interfaces: Part I* and Chapter 9, *Graphical User Interfaces: Part II* does not depend upon the student having done or read the earlier applet exercises.

Threads. Multi-threaded programming is not usually discussed in a beginning text. However, some understanding of threads is essential for a true understanding of the workings of event driven GUI based programs. In addition, the authors feel that thread-based programming will become increasingly important at all levels of the programming curriculum. Threading is explained in Chapter 13, *Threads: Concurrent Programming*, and used to introduce the reader to client/server computing. This book gives a treatment suitable to the beginning programmer that has mastered the topics in the preceding chapters.

Course-Tested. This book is the basis of courses given by the authors, who have used its contents to train students in various forums since 1997. The material is course-tested, and reflects the author's considerable teaching and consulting experience.

Code examples. All the major pieces of code were tested. A consistent and proper coding style is adopted from the beginning and is one chosen by professionals in the Java community. The code is available at the Addison Wesley Longman Web site (*www.awl.com*).

Common Programming Errors. Many typical programming bugs, along with techniques for avoiding them, are described. This books explains how common Java errors are made and what must be done to correct them. The new edition also describes the use of assertions that came in with Java1.4.

Exercises. The exercises test and often advance the student's knowledge of the language. Many are intended to be done interactively while reading the text, encouraging self-paced instruction.

Web site. The examples both within the book are intended to exhibit good programming style. All examples for this book are available on the web at www.soe.ucsc.edu/~pohl/java.

New in the Second Edition. Java 5.0 came out in September of 2004 with important changes over the original language. These include: container based generics, enum types, assertions, more threading refinements, terminal IO using Scanner, and formatted IO using printf(). We have included all these important topics and where possible revised the text to take advantage of them.

Course use:

- First Course in Programming: Java by Dissection is designed to be used as a basic first programming course - similar in scope to courses that used C, Pascal, or C++. Chapters 1-6 cover such a curriculum.

- First Course in Programming - Swing emphasis: By using supplementary material at the end of each of the first chapters the instructor can take a GUI approach as opposed to the traditional terminal I/O approach.

- CS1 Course in Programming: By covering material at a more rapid pace for well prepared computer science majors the book can function as the primary text in a CS1-2 sequence. The instructor would select more advanced exercises and cover material through chapter 11.

- Second Course in Programming: The book can be used as a second or advanced course covering object-oriented programming. Chapters 2-5 can be skimmed by anyone already familiar with a procedural programming language such as C or Pascal. A programmer already familiar with OOP concepts could also skim chapters 6 and 7. Chapters 8-13 provide a mix of interesting advanced topics, not generally covered in a beginning programming course.

Acknowledgments

Our special thanks go to Debra Dolsberry and Linda Werner for their encouragement and careful reading and suggestions for improvement. Debra was especially helpful with typesetting issues. Our student Sarah Berner was an important contributor to the effectiveness of the text in its first edition and especially helpful in converting many examples and exercises over to SWING. Thanks to Uwe F. Mayer for his careful review of the second edition which caught numerous errors.

Charlie McDowell and Ira Pohl
University of California, Santa Cruz

CHAPTER 1

INTRODUCTION

Java is the first major programming language to be shaped by the World Wide Web (the Web). Java allows you to do traditional programming. Java also has many special features and libraries that allow you conveniently to write programs that can use the Web's resources. These include extensive support for graphical user interfaces, the ability to embed a Java program in a Web document, easy communication with other computers around the world, and the ability to write programs that run in parallel or on several computers at the same time.

In this chapter we give an overview of how to solve a problem on a computer. In this process, you must first construct a recipe for solving the problem. Then you must convert the recipe into a detailed set of steps that the computer can follow. Finally, you must use a programming language, such as Java, to express the steps in a form "understandable" by the computer. The Java form of the solution is then translated by a program called a compiler into the low-level operations that the computer hardware can follow directly.

We then discuss why Java is creating such a fuss in the computing world. In general terms, we explain the importance of computing on the Web and the character of the graphical user interfaces, or GUIs, that are partly behind the switch to Java.

Throughout this text we feature carefully described examples, many of which are complete programs. We often dissect them, allowing you to see in detail how each Java programming construct works. Topics introduced in this chapter are presented again in later chapters, with more detailed explanations. The code and examples are meant to convey the flavor of how to program. You should not be concerned about understanding the details of the examples. They are given in the spirit of providing an overview. If you are already familiar with what a program is and want to start immediately on the nitty-gritty of Java programming, you may skip or scan this chapter.

1.1 RECIPES

Computer programs are detailed lists of instructions for performing a specific task or for solving a particular type of problem. Programlike instruction lists, sometimes called *algorithms*, for problem solving and task performance are commonly found in everyday situations. Examples include detailed instructions for knitting a sweater, making a dress, cooking a favorite meal, registering for classes at a university, traveling from one destination to another, and using a vending machine. Examining one of these examples is instructive.

Consider this recipe for preparing a meat roast:

Sprinkle the roast with salt and pepper. Insert a meat thermometer and place in oven preheated to 150 ºC. Cook until the thermometer registers 80 ºC–85 ºC. Serve roast with gravy prepared from either meat stock or from pan drippings if there is a sufficient amount.

The recipe is typically imprecise—what does “sprinkle” mean, where is the thermometer to be inserted, and what is a “sufficient amount” of pan drippings?

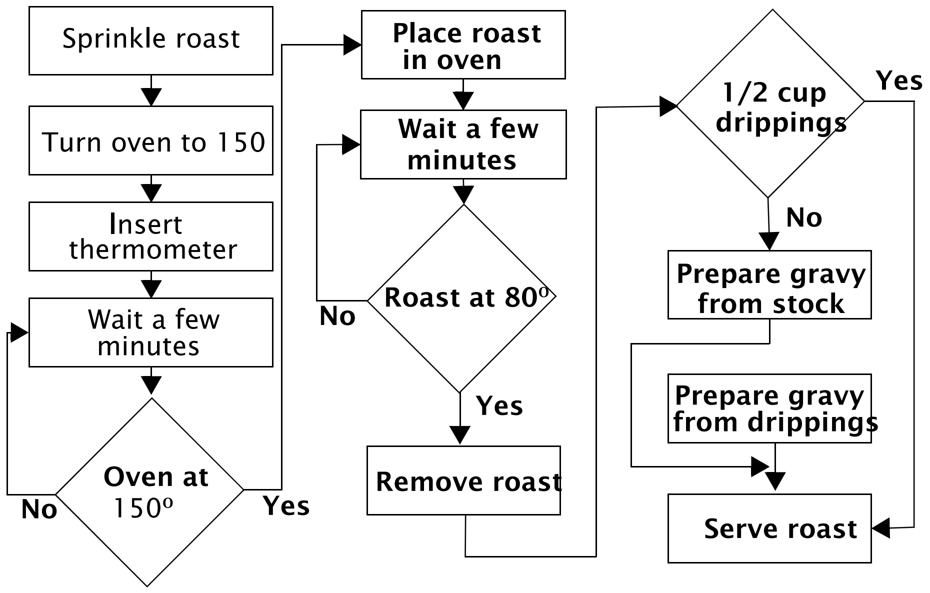
However, the recipe can be formulated more precisely as a list of instructions by taking some liberties and reading “between the lines.”

Cooking a roast

1. Sprinkle roast with 1/8 teaspoon salt and pepper.
2. Turn oven on to 150º C.
3. Insert meat thermometer into center of roast.
4. Wait a few minutes.
5. If oven does not yet register 150º C, go back to step 4.
6. Place roast in oven.
7. Wait a few minutes.
8. Check meat thermometer. If temperature is less than 80º C, go back to step 7.
9. Remove roast from oven.
10. If there is at least 1/2 cup of pan drippings, go to step 12.
11. Prepare gravy from meat stock and go to step 13.
12. Prepare gravy from pan drippings.
13. Serve roast with gravy.

These steps comprise three categories of instructions and activities—those that involve manipulating or changing the ingredients or equipment, those that just examine or test the “state” of the system, and those that transfer to the next step. Steps 1 and 6 are examples of the first category; the temperature test in step 8 and the pan dripping test in step 10 are instances of the second category; and the transfers in steps 5 and 8 (“go to step *x*”) are examples of the last category.

By using suitable graphical symbols for each of these categories, a simple two-dimensional representation of our cooking algorithm can be obtained, as shown in the following illustration.



Such a figure is called a *flowchart*. To perform the program (prepare the roast), just follow the arrows and the instructions in each box. The manipulation activities are contained in rectangles, the tests are shown in diamonds, and the transfer or flow of control is determined by the arrows. Because of their visual appeal and clarity, flowcharts are often used instead of lists of instructions for informally describing programs. Some cookbook authors even employ flowcharts extensively. In this book we use flowcharts in Chapter 3, *Statements and Control Flow*, when describing the behavior of some Java language constructs.

1.2 ALGORITHMS—BEING PRECISE

Our recipe for preparing a roast can't be executed by a computer because the individual instructions are too loosely specified. Let's consider another example—one that manipulates numbers instead of food. You need to pay for some purchase with a dollar bill and get change in dimes and pennies. The problem is to determine the correct change with the fewest pennies. Most people do this simple everyday transaction unthinkingly. But how do we precisely describe this algorithm?

In solving such a problem, trying a specific case can be useful. Let's say that you need to pay 77 cents and need change for a dollar. You can easily see that one dollar minus the 77 cents leaves you with 23 cents in change. The correct change having the fewest coins in dimes and pennies would be two dimes and three pennies. The number of dimes is the integer result of dividing 23 by 10 and discarding any fraction or remainder. The number of pennies is the remainder of dividing the 23 cents by 10. An algorithm for performing this change for a dollar is given by the following steps.

CHANGE-MAKING ALGORITHM

1. Assume that the price is written in a box labeled `price`.
2. Subtract the value of `price` from 100 and place it in a box labeled `change`.
3. Divide the value in `change` by 10, discard the remainder, and place the result in a box labeled `dimes`.
4. Take the integer remainder of `change` divided by 10 and place it in a box labeled `pennies`.
5. Print out the values in the boxes `dimes` and `pennies` with appropriate labels.
6. Halt.

This algorithm has four boxes, namely, `price`, `change`, `dimes`, and `pennies`. Let's execute this algorithm with the values given. Suppose that the price is 77 cents. Always start with the first instruction. The contents of the four boxes at various stages of execution are shown in the following table.

Box	Step 1	Step 2	Step 3	Step 4	Step 5
PRICE	77	77	77	77	77
CHANGE		23	23	23	23
DIMES			2	2	2
PENNIES				3	3

To execute step 1, place the first number, 77, in the box `price`. At the end of instruction 2, the result of subtracting 77 from 100 is 23, which is placed in the box `change`. Each step of the algorithm performs a small part of the computation. By step 5, the correct values are all in their respective boxes and are printed out. Study the example until you're convinced that this algorithm will work correctly for any price under $1.00. A good way to do so is to act the part of a computer following the recipe. Following a set of instructions in this way, formulated as a computer program, is called *hand simulation* or *bench testing*. It is a good way to find errors in an algorithm or program. In computer parlance these errors are called *bugs,* and finding and removing them is called *debugging*.

We executed the change-making algorithm by acting as an agent, mechanically following a list of instructions. The execution of a set of instructions by an agent is called a *computation*. Usually the agent is a computer; in that case, the set of instructions is a computer program. In the remainder of this book, unless explicitly stated otherwise, we use *program* to mean *computer program*.

The algorithm for making change has several important features that are characteristic of all algorithms.

Characteristics of Algorithms

- The sequence of instructions will terminate.
- The instructions are precise. Each instruction is unambiguous and subject to only one interpretation.
- The instructions are simple to perform. Each instruction is within the capabilities of the executing agent and can be carried out exactly in a finite amount of time; such instructions are called *effective*.
- There are inputs and outputs. An algorithm has one or more outputs ("answers") that depend on the particular input data. The input to the change-making algorithm is the price of the item purchased. The output is the number of dimes and pennies.

Our description of the change-making algorithm, although relatively precise, is not written in any formal programming language. Such informal notations for algorithms are called *pseudocode*, whereas real code is something suitable for a computer. Where appropriate we use pseudocode to describe algorithms. Doing so allows us to explain an algorithm or computation to you without all the necessary detail needed by a computer.

The term *algorithm* has a long, involved history, originally stemming from the name of a well-known Arabic mathematician of the ninth century, Abu Jafar Muhammed Musa Al-Khwarizmi. It later became associated with arithmetic processes and then, more particularly, with Euclid's algorithm for computing the greatest common divisor of two integers. Since the development of computers, the word has taken on a more precise meaning that defines a real or abstract computer as the ultimate executing agent—any terminating computation by a computer is an algorithm, and any algorithm can be programmed for a computer.

1.3 IMPLEMENTING OUR ALGORITHM IN JAVA

In this section we implement our change-making algorithm in the Java programming language. You need not worry about following the Java details at this point; we cover all of them fully in the next two chapters. We specifically revisit this example in Section 2.10.1, *An Integer Arithmetic Example: Change*, on page 31. For now, simply note the similarity between the following Java program and the informal algorithm presented earlier. You not only have to be able to formulate a recipe and make it algorithmic, but you finally have to express it in a computer language.

```
// MakeChange.java - change in dimes and pennies
import java.util.*; // use Scanner for input

class MakeChange {
  public static void main (String[] args) {
    int price, change, dimes, pennies;
    Scanner scan = new Scanner(System.in);
    System.out.println("type price (0 to 100):");
    price = scan.nextInt();
    change = 100 - price;      //how much change
    dimes = change / 10;       //number of dimes
    pennies = change % 10;     //number of pennies
    System.out.print("The change is :");
    System.out.print(dimes);
    System.out.print(" dimes ");
    System.out.print(pennies);
    System.out.print(" pennies.\n");
  }
}
```

Dissection of the *MakeChange* Program

- `java.util.*; // use Scanner for input`

A package is a library or collection of previously written program parts that you can use. This line tells the Java compiler that the program `MakeChange` uses information from the package `java.util`. It allows you to use the `Scanner` class, which we explain shortly.

- `int price, change, dimes, pennies;`

This program declares four integer variables. These hold the values to be manipulated.

- `Scanner scan = new Scanner(System.in);`

This declares a `Scanner` variable `scan` that is tied to `System.in`. This lets `scan` provide input by typing to the terminal.

- `System.out.println("type price(0 to 100):");`

This line is used to *prompt* you to type the price. Whenever a program is expecting a user to do something, it should print out a prompt telling the user what to do. The part in quotes appears on the user's screen when the program is run.

```
● price = scan.nextInt();
```

The `scan.nextInt()` is used to obtain the input from the keyboard. The value read is stored in the variable `price`. The symbol `=` is called the *assignment operator.* Read the first line as “`price` is *assigned the value* obtained from the typing an integer on your keyboard. At this point you must type in an integer price. For example, you would type 77 and then hit Enter.

```
● change = 100 - price;       //how much change
```

This line computes the amount of change.

```
  dimes = change / 10;        //number of dimes

● pennies = change % 10;      //number of pennies
```

The number of dimes is the integer or whole part of the result of dividing `change` by 10. The symbol `/`, when used with two integers, computes the whole (nonfraction) part of the division. The number of pennies is the integer remainder of `change` divided by 10. The symbol `%` is the integer remainder or modulo operator in Java. So if `change` is 23, the integer divide of 23/10 is 2 and the integer remainder or modulo of 23 % 10 is 3.

```
● System.out.print("The change is : ");
  System.out.print(dimes);
  System.out.print(" dimes ");
  System.out.print(pennies);
  System.out.print(" pennies.\n");
```

In this example the `System.out.print()` statements cause the values between the parentheses to be printed on the computer console. The first one just prints out the characters between the quotation marks. The second one converts the value in `dimes` to the sequence of digits and prints those digits. The other print statements are similar. For an input value of 77, the output would be

```
The change is : 2 dimes 3 pennies
```

The `\n` in the last print statement indicates that a newline should be sent to the console, ending the line of output.

1.4 WHY JAVA?

A variety of programming languages are available for expressing programs, but the most useful ones are suitable for both machine and human consumption. In this book we cover programming by using one such language—the programming language Java.

Java is a relatively new programming language, first introduced in 1995. In this book a language is needed that allows algorithms to be easily understood, designed, analyzed, and implemented as computer programs. The following is an excerpt from the original paper introducing Java to the world.

In his science-fiction novel *The Rolling Stones*, Robert A. Heinlein comments:

> Every technology goes through three stages: first a crudely simple and quite unsatisfactory gadget; second, an enormously complicated group of gadgets designed to overcome the shortcomings of the original and achieving thereby somewhat satisfactory performance through extremely complex compromise; third, a final proper design therefrom.

Heinlein's comment could well describe the evolution of many programming languages. Java presents a new viewpoint in the evolution of programming languages—creation of a small and simple language that's still sufficiently comprehensive to address a wide variety of software application development. Although Java is superficially similar to C and C++, Java gained its simplicity from the systematic removal of features from its predecessors.

We agree with its creators that Java has, to a large extent, lived up to the promise of a small, simple, yet comprehensive programming language. As such, Java is both an excellent programming language for real program development and a good first programming language.

Possibly even more important for Java's initial success are the features that make it appropriate for development of programs distributed over the Internet. These programs, called *applets*, execute inside Internet browsers such as Firefox, Mozilla, Netscape, and Internet Explorer.

1.5 NETWORK COMPUTING AND THE WEB

Much of modern computing is done in a connected environment. That is, computers are connected to other computers in a network. This connection allows the computers to exchange information or to "talk to each other." Java was developed with network computing as the normal environment. It has many features and libraries of parts of programs that promote networking. Earlier languages and systems viewed a computer as a lone instrument doing significant computations, the results of which would largely be output to a screen or printer. By promoting networking whereby computers are connected and pass results to each other dynamically—and cooperate on large-scale computations—Java became the principal language for computing on networks.

The largest network is the global network called the *Internet*. Using the Internet, researchers at the European Particle Physics Laboratory (CERN) developed a way to share information via a formatting language called *hyper-text markup-language*, or *HTML*. The computers exchanged HTML documents by means of a protocol called *hyper-text transfer protocol*, or *HTTP*. A *protocol* is like a language that computers use to "talk" to each other. HTML allows one electronic document to link to another electronic document on a different computer. The program used to view HTML documents and follow the links is called a *browser*. With the click of a button, a user could move from one document on one computer to another, related document, on another computer. Because of these links, the resulting web of documents was dubbed the *World Wide Web* or simply *the Web*. Today, many people equate the Web with the Internet, but the Web is only one application of the technology known as the Internet.

Java programs, called *applets*, can be written to automatically load and execute by following a link in the Web. This ability to embed Java programs in HTML documents was a primary factor in the early success of Java. With Java applets, Web documents are no longer static text, images, or video segments. Web documents can provide all the interaction of any program. We discuss graphical user interfaces and applets in Chapter 8, *Graphical User Interfaces: Part I.*

If you're interested in an early exposure to applets, we also provide templates for simple applets in the exercises at the end of Chapters 2 through 6.

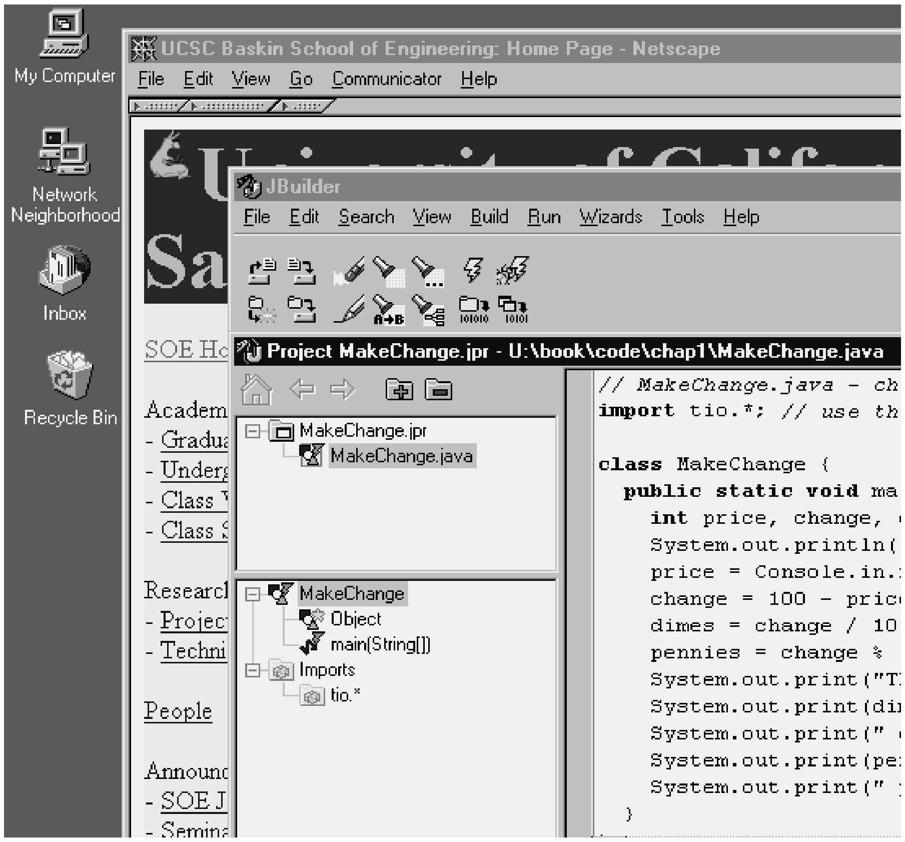
From the World Wide Web (www) and HTTP, you can begin to make some sense of the ubiquitous Internet addresses such as *http://www.company.com*. This type of address is called a *universal resource locator*, or *URL*. These are addresses embedded in HTML documents, linking one document to another.

Client-server computing is an essential new element in computers that manage communications and computations. The server, typically a fast computer with a very large amount of hard disk storage, gives out information to clients upon request. An example would be a server giving out stock market quotes to clients that request it. The network support in Java makes implementing client–server programming solutions easy. In Chapter 9, *Graphical User Interfaces: Part II*, we show you how to implement a simple Java server program that can connect and respond to requests from multiple clients on other computers.

1.6 HUMAN–COMPUTER INTERACTION AND THE GUI

In the early days of computing, most interaction between a computer and a human being was in the form of typed input to the computer and printed output to the person. Most human–computer interaction today is done with a *graphical user interface* (*GUI*—pronounced gooey). The user has a keyboard, usually similar to a standard typewriter keyboard, a pointing device, usually a mouse, and a display device capable of displaying both text and graphics.

The user's display is usually divided into viewing areas called *windows*. Associated with the windows are menus of commands relating to manipulation of the data displayed in the window. The display (screen) may also contain icons, or small graphical images, whose visual appearance is designed to signify their functions. For example, a picture of a trash can represents deleting a file, and an arrowhead is used to scroll a viewing screen. The following screen shot of a typical desktop shows several windows, icons, and a row of menus across the top of each window.



```
// MakeChange.java - ch
import tio.*; // use th

class MakeChange {
  public static void ma
    int price, change,
    System.out.println(
    price = Console.in.
    change = 100 - pric
    dimes = change / 10
    pennies = change %
    System.out.print("T
    System.out.print(di
    System.out.print("
    System.out.print(pe
    System.out.print("
  }
```

For many years, the task of creating a graphical user interface for a new application was extremely time-consuming and difficult. The programming languages and operating systems in use did not make building such interfaces easy. These interfaces were usually built from libraries of program parts that were suitable only for one particular operating system. If a company building a new program wanted to have its program run on computers with different operating systems, it would have to create multiple versions of the programs.

Java provides a solution. Software companies now sell programs that make building graphical user interfaces relatively easy. Java includes a library of program parts called *Swing* for building graphical user interfaces. The program parts in Swing can be used on any computer that can run Java programs. Thus a programmer can now write one version of a program and have it run on many different kinds of computers. Although in-depth coverage of Swing is beyond the scope of this book, Swing is simple enough to use that even beginning programmers can begin to build programs with graphical user interfaces. We discuss Swing further in Chapter 8, *Graphical User Interfaces: Part I* and Chapter 9, *Graphical User Interfaces: Part II.*

By the end of this book, you will be writing your own sophisticated applets and regular Java programs with GUIs. If you've used a Java-enabled browser, such as Netscape or Explorer, you've probably unknowingly made use of an applet. If not, you can try out a simple applet calculator right now by going to *www.soe.ucsc.edu/~charlie/java/jbd/MiniCalcApplet.html*. This is the same applet presented in Section 8.8, *Applets*, on page 248.

SUMMARY

- Informally, an algorithm is a list of instructions for performing a specific task or for solving a particular type of problem. Algorithmlike specifications for problem solving and task performance are commonly found in everyday situations. A recipe is a kind of algorithm.

- One graphical representation of an algorithm is a flow chart. To perform the algorithm, the user just follows the arrows and the instructions in each box. The manipulation activities are contained in rectangles, the tests are shown in diamonds, and the transfer or flow of control is determined by the arrows. Because of their visual appeal and clarity, flow charts are often used instead of lists of instructions for informally describing algorithms.

- Algorithms that can be executed on computers are called programs. Programs are written in special languages called programming languages, such as Java, which we use in this book.

- Modern computing is done in a connected environment. A computer is connected to other computers in a network. This connection allows the computers to exchange information or to "talk to each other." Java was developed with network computing as the normal environment. The largest network is the global network called the Internet. This network is used to exchange documents throughout the world with a common format called HTML. This format allows links to be followed across the Internet. Because of these links, the resulting web of documents was named the World Wide Web or simply the Web.

- It is possible to write Java programs called applets, that are automatically loaded and executed as the result of following a link in the Web. This ability to embed Java programs in HTML documents is a primary factor in the success of Java. With Java applets, Web documents are no longer static text, images, or video segments. Web documents can provide all the interactions of any program.

REVIEW QUESTIONS

1. What is an algorithm?

2. A graphical description of a computation using boxes and arrows is called a ________________.

3. Informal, but relatively precise descriptions of algorithms are called ________________.

4. What is bench testing? Why is it important?

5. What did `java.util` refer to in the `MakeChange.java` program?

6. What was `System.out.print()` used for in the Java program?

7. What is HTML? What is HTTP? What is URL?

8. What is a browser? Name a commonly used browser.

9. What are the primary components of a typical human–computer interface today?

10. The information provided by the user to a program when it executes is called the _____________. The answers provided by the program are called the _____________.

11. What is Swing?

Exercises

1. Write a flow chart for the following recipe for cooking baked beans (taken from *The Natural Foods Cookbook*, Nitty Gritty Productions, Concord, California, 1972).

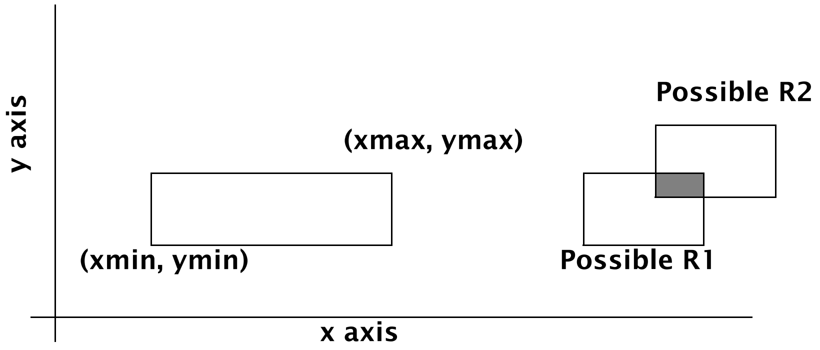
> Bring apple juice and water to a boil and add beans so slowly that boiling doesn't stop. Reduce heat after beans are in water and simmer 2 to 2 and a half hours or until beans are almost tender. Drain beans, reserving liquid, and add other ingredients to beans. Place in oiled baking dish and bake covered 2 to 3 hours in 2508 oven. Uncover for the last hour of cooking. If beans become dry, add a little of the reserved bean water. About 15 minutes before removing from the oven, add the fresh diced tomato.

2. Let m and n be positive integers. Draw a flowchart for the following algorithm for computing the quotient q and remainder r of dividing m by n.

    1. Set q to 0.
    2. If $m < n$, then stop. q is the quotient and m is the remainder.
    3. Replace m by $m - n$.
    4. Increment q by 1.
    5. Go to step 2.

3. Bench test the algorithm in the previous exercise. Use $m = 4$ and $n = 2$ as values.

4. Let a rectangle that is aligned with the coordinate axes be represented by the coordinates of its lower left and upper right corners, (*xmin*, *ymin*) and (*xmax*, *ymax*), respectively, as shown in the following illustration. Given two such rectangles, *R1* and *R2*, devise an algorithm that finds the rectangle, if any, that is common to both *R1* and *R2*. The input data are the eight real numbers representing the coordinates of the rectangles' corners. The illustration shows two rectangles that have a common rectangle, shown in grey.



5. Suppose that a line segment is represented by its two endpoints, *PE* = (*xe*, *ye*) and *PP* = (*xp*, *yp*). For two line segments *L1* and *L2*, devise an algorithm that computes the coordinates of their point of intersection, if any. (Two line segments intersect if they have only one point in common.)

6. Make a list of all the automatic interactions that you have with computers—for example, utility bills, automatic mailing lists, class scheduling, and so on.

7. Give a detailed step-by-step procedure, also called an algorithm, for
   - traveling from your place of residence to school or work,
   - manually dividing one number by another (long division),
   - taking an unordered set of numbers and putting them in ascending sequence (i.e., "sorting" the numbers in ascending sequence), or
   - playing, and never losing, the game of Tic-Tac-Toe.

CHAPTER 2

PROGRAM FUNDAMENTALS

In this chapter we introduce programming fundamentals in Java. We write a Java program by using different elements of the Java language. The most basic elements are the tokens of the language. Tokens are the words that make up the sentences of the program. Programming is comparable to writing. To write an essay we write words that make up sentences, sentences that make up paragraphs, and paragraphs that make up essays. In this chapter we concentrate on how to use the "words" and combine them into the program equivalent of useful sentences and paragraphs.

2.1 "HELLO, WORLD!" IN JAVA

A simple first Java program is the classic "Hello, world!" program, so named because it prints the message `Hello, world!` on the computer's screen.

```
/* HelloWorld.java
 * Purpose:
 *   The classic "Hello, world!" program.
 *   It prints a message to the screen.

 * Author: Jane Programmer
 *       as derived from Kernighan and Richie
 */

class HelloWorld {
  public static void main (String[] args) {
    System.out.println("Hello, world!");
  }
}
```

Dissection of the *HelloWorld* Program

-
  ```
  /* HelloWorld.java
   * Purpose:
   ...
   */
  ```

Everything between a `/*` and a `*/` is a comment. Comments are ignored by the Java compiler and are inserted to make the program more understandable to the human reader. Every program should begin with a comment such as the one in this example. In our examples, the name of the file appears in the comment. This example indicates that it is from the file *HelloWorld.java*. Other things to include in program comments are the function or purpose of the program, the author of the program, and a revision history with dates, indicating major modifications to the program.

- `class HelloWorld {`

The word `class` is a keyword preceding the name of the *class*. A *keyword* has a predefined special purpose. A *class* is a named collection of data and instructions. The name of the class being defined in this example is `HelloWorld.` The left brace "{" begins the definition of a class. A matching right brace "}" is needed to end the class definition. Forgetting to match braces is a common error.

- `public static void main (String[] args) {`

This line declares that the class `HelloWorld` contains a *method* with the name `main`. A *method* is a named group of instructions within a class. In this example, only one method, named `main`, is defined for the class `HelloWorld`. In Chapter 4, *Methods: Functional Abstraction*, we create classes that contain several methods in a single class. When we use the name of a method, we add parentheses at the end to remind you that it is a method. This convention comes from actual Java code, wherein the name of a method is always followed by parentheses. The method defined by this line is thus referred to as `main()`.

There are two kinds of Java programs: *stand-alone applications* and *applets.* The method `main()` appears in every stand-alone Java program indicating where program execution will begin. Later we use a different line that serves a similar purpose for applets. An explanation of the words `public`, `static`, and `void` in this line is left until later.

-
  ```
  {
    System.out.println("Hello, world!");
  }
  ```

The entire body of the method `main()`, the real instructions to the computer, appears between the braces. In this example, just one instruction prints the desired message. You can memorize this instruction and use it as an incantation to get something printed out on your computer screen. What gets printed is between the quotation marks.

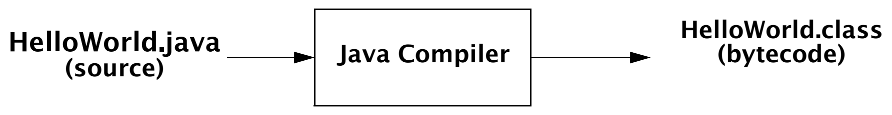
2.2 COMPILING AND RUNNING YOUR JAVA PROGRAM

You will be dealing with two different representations of your programs: the part you write and a form more suitable for the computer to use when it finally runs your program. The text you write to give the computer instructions is called the *source code* or simply the *source*. This source code will be compiled by the Java *compiler* into a form more suitable as instructions to the computer called *object code*. The source code form of the program is represented in the language Java that you will be learning. Informally, you can think of the source code as the *raw* form of the program in contrast to the object code, which is the *cooked* or compiled form. In Java, all source code file names have the suffix *.java*, such as *HelloWorld.java*. The result of correctly compiling *HelloWorld.java* is the object code *HelloWorld.class*. In some situations, the name of the file without the *.java* and the name of the class defined in the file must be the same. Although this requirement does not apply to the programs in the first part of this book, we follow that practice even when it isn't required and suggest that you do the same.

There are many names and forms for the machine representation of a program after it has been processed somewhat by the computer. A common first step in processing source code is to compile it, which means to translate it into a form more suitable for the computer. For most common programming languages, this compiled form is called the *machine code, object code,* or *binary form*. When you buy a piece of software you usually get *binaries* or an *executable image*. For Java, this form is a bit different and is called *Java Bytecode*. These bytecodes are the same whether you are running on a Macintosh, on an Intel machine with Microsoft Windows, or on a machine from Sun running Unix. This sameness is an important advantage of Java programs, as they can be written to be platform independent, which is generally not true for most other programming languages, such as C or COBOL.

In Java, all bytecode files must have a name that ends in *.class* such as *HelloWorld.class.* The word *class* is used because, in Java, programs are broken into chunks called classes, much like a chapter is broken into sections. The following diagram illustrates the compilation process and the conventions just described.

The Compilation Process



There are two principal methods for compiling and running Java programs. One method uses an Integrated Development Environment (IDE), of which there are many for Java. The actual details of compiling for each of the IDEs vary slightly but the basic steps are the same.

Java via the IDE

1. Use the editor that comes with the IDE to create the source file. These editors generally help with syntax by using special fonts or colors for keywords and helping you match braces, and so on.
2. Create a project and add your source file to the project.
3. Select Run from a menu. The IDE will automatically determine that the source file needs to be compiled and compile it before trying to run the program.

The other principal method is the command line approach. In it you run the compiler yourself from the command line of either a Command Prompt in Windows or Unix terminal or shell program. For both Unix and Windows the steps are the same.

Java via the Command Line

1. Use your favorite text editor to create the source file. Save the source into a file ending with the extension *.java*.
2. Compile the program with the command `javac` followed by the source file name—for example, `javac HelloWorld.java`.
3. If the program compiles without errors, run the program with the command `java` followed by the name of the class. Do not append *.class*, as in *HelloWorld.class*; use only the class name— for example, `java HelloWorld`.

The last two steps are shown as follows for the program `HelloWorld`.

```
os-prompt>javac HelloWorld.java
os-prompt>java HelloWorld
Hello, world!
os-prompt>
```

2.3 LEXICAL ELEMENTS

Java has rules about how its language elements can be put together to form a complete program. In this section we begin by looking at the various words and symbols, called *lexical elements*, that are used to construct Java programs.

The most fundamental element in the structure of a program is a single character that can be displayed on a computer screen or typed at a computer keyboard. Prior to Java, most programming languages, such as C and C++, used the ASCII character set. It provides for 127 different characters, which is enough to represent all the characters on the conventional English language keyboard. This set may seem like a lot, but when you consider all the human languages in the world and the various symbols they use, it is inadequate. Because Java was designed to be used throughout the world, not just in English-speaking countries, Java developers adopted the Unicode character set. It provides for more than 64,000 different characters.

When a Java compiler first begins to analyze a Java program, it groups the individual characters into larger lexical elements, usually called *tokens*. Some tokens—such as the plus sign, +, which is the Java symbol used to add two numbers—are only one character long. Other tokens—such as the keywords `class` and `public`—are many characters long. These basic tokens are then combined into larger language forms, such as expressions. An example of an expression comprising three tokens is `x+y`.

There are five types of tokens: keywords, identifiers, literals, operators, and punctuation. White space and comments are two additional lexical elements that are discarded early in the compilation process.

2.3.1 WHITE SPACE

White space in Java refers to the space character, which you get when you strike the space bar on the keyboard; the tab character, which is actually one character, although it may appear as several spaces on your screen; and the newline character which you get when you hit the Return or Enter key on the keyboard. White space is used primarily to make the program look nice and also serves to separate adjacent tokens that are not separated by any other punctuation and would otherwise be considered a single, longer token. For example, white space is used to separate the following three tokens:

```
public static void
```

in the `HelloWorld` program. In such situations, where one white space character is required, any number of white space characters can be used. For example, we could have put each of the words `public`, `static`, and `void` on separate lines or put lots of spaces between them as in

```
   public      static      void      main(...
```

Except for string literals, which we discuss shortly, any number of adjacent white space characters—even mixing tab, space, and newline characters—is the same as just one white space character as far as the structure and meaning of the program are concerned. Stated another way, if you can legally put in one space, you can put in as many spaces, tabs, and newlines as you want. You can't put white space in the middle of a *keyword* or *identifier*, such as a *variable* or class name. We discuss keywords, identifiers, and variables later in this chapter.

2.3.2 COMMENTS

Comments are very important in writing good code and are too often neglected. The purpose of a comment is to provide additional information to the person reading a program. It serves as a concise form of program documentation. As far as the computer is concerned, the comment does not result in a token; it separates other tokens or is ignored completely when it isn't needed to separate tokens. Java has three ways to specify comments.

A *single line* comment begins with `//` and causes the rest of the line—all characters to the next newline—to be treated as a comment and ignored by the compiler. It is called a single line comment because the comment can't be longer than a single line. By definition, the comment ends at the end of the line containing the `//`.

A *multiline* comment can extend across several lines in a program. The beginning of the comment is marked with `/*` and the end of the comment is marked with `*/`. Everything between the marks and the marks themselves is a comment and is ignored. Here is the multiline comment from our first Java program.

```
/* HelloWorld.java
 * Purpose:
 *   This is the classic "Hello, world!" program.
 *   It simply prints a message to the screen.
 * Author:
 *    Jane Programmer
 *       as derived from Kernighan and Richie
 */
```

The single asterisks on the intermediate lines are not required and are used merely to accent the extent of the comment. These comments are also called *block comments.*

The third style of comment is a minor variation on the multiline comment. The beginning marker has an additional asterisk; that is, the beginning of the comment is marked with `/**` and the end of the comment is marked with `*/`. These comments are identical to the multiline comment except that they are recognized by a special program called *javadoc* that automatically extracts such comments and produces documentation for the program organized as an HTML document. See Section 4.13, *Programming Style*, on page 111, for more about *javadoc*.

2.3.3 KEYWORDS

Keywords, also known as *reserved words*, have a predefined special purpose and can't be used for any but that purpose. Each of the 47 keywords in Java has a special meaning to the Java compiler. A keyword must be separated from other keywords or identifiers by white space, a comment, or some other punctuation symbol. The following table shows all the Java keywords.

abstract	continue	for	new	synchronized
assert	default	goto	package	this
boolean	do	if	private	throw
break	double	implements	protected	throws
byte	else	import	public	transient
case	enum	instanceof	return	try
catch	extends	int	short	void
char	final	interface	static	volatile
class	finally	long	super	while
const	float	native	switch	

The keywords `const` and `goto` have no meaning in Java. They are keywords in C++, a language that was a precursor to Java. They are included as keywords to facilitate error reporting when programmers with C++ experience accidentally use them. In addition, the words `null`, `true`, and `false` look like keywords in that they have a predefined meaning, but they are in fact literals, as discussed later.

2.3.4 IDENTIFIERS

Identifiers are the names used to specify different elements of a Java program, such as a class, method, or variable. (We discuss variables in Section 2.4.1, *Variables*, on page 20.) An identifier in our first program was `HelloWorld`, a name we picked for the `class`. Another identifier was the library method name `println`, a name picked by the Java developers. In both cases, the name gives a clue as to the use of that element of the program. An *identifier* is any sequence of Java letters and digits, the first of which must be a *Java letter*, with two exceptions. A keyword can't be an identifier, and the special literal terms `true`, `false`, and `null` can't be used as identifiers. The Java letters and digits include the letter and digit symbols for many modern written languages. The Java letters include the English language uppercase and lowercase letters, the $, and the _ (underscore). The last two are included because of Java's close kinship to the programming language C, which included these symbols as legal characters in identifiers. The Java digits are 0 through 9. In our examples, we use only English letters and digits. The following are some examples of legal identifiers along with comments providing some explanation.

```
data                        //variable name conveying its use
HelloWorld                  //class name
youCanAlmostMakeASentence   //unlikely
nextInt                     //method name from Scanner class
x                           //simple variable usually double
__123                       //obscure name-a poor choice
```

The following are some illegal identifiers, along with comments indicating what the sequence of symbols really is.

```
3              //a digit or integer literal
x+y            //an expression where x and y are identifiers
some***name    //illegal internal characters
no Space       //intended was noSpace
1floor         //cannot start with a digit
class          //keyword - cannot be used as an identifier
```

2.3.5 LITERALS

Java has built-in *types* for numbers, characters, and booleans. Java also has a standard class type for strings. The *type* of a data value tells the computer how to interpret the data. These built-in types are called *primitive types* in Java. *Literals,* also called *constants,* are the literal program representations of values for the primitive numeric types, the primitive type `boolean`, the primitive character type `char`, and the standard class string type `String`. Without going into the details of these various types, the following are some examples of literal values

JAVA TYPE	EXPLANATION	EXAMPLES
`int`	Integers—numbers without fractional parts	`123   -999   0`
`double`	Double precision numbers with fractional parts	`1.23   -0.01`
`String`	Arbitrary strings of characters	`"Oh J" "123"`
`boolean`	Logical values true or false	`true   false`
`char`	Single characters	`'a'    '1'`

Like keywords, the symbols `true` and `false` can't be used as identifiers. They are reserved to represent the two possible boolean values. We explain later, in more detail, what constitutes an acceptable literal for each data type.

2.3.6 OPERATORS AND PUNCTUATION

In addition to keywords, identifiers, and literals, a Java program contains operators and separators or punctuation. The operators are things like "+", and the separators are things like the ";" that terminates some statements and the braces "{ }" used to group things. To fully understand operators, you need to understand type, precedence, and associativity. *Type* determines the kind of value computed, such as `int` or `double`. *Precedence* determines among operators, such as `+` and `/` used in an expression, which is done first. *Associativity* is the order in which operators of the same precedence are evaluated and is usually left-most first—for example,

```
int n = 2;
n = n * 3 + 2;     //an assignment expression
```

The variable `n` is an integer variable initialized to 2. Next `n` is multiplied by the integer literal 3. This result is added to 2, and finally this value is assigned to `n`, replacing the original value of 2 with the new value of 8. Precedence of the multiplication operator `*` is higher than `+`, so the multiplication is done before the addition. Precedence of the assignment operator `=` is lowest, so assignment occurs as the last action in this expression. We discuss precedence and associativity of operators further in Section 2.13, *Precedence and Associativity of Operators*, on page 37.

2.4 DATA TYPES AND VARIABLE DECLARATIONS

In order to do something useful, computer programs must store and manipulate data. Many programming languages, including Java, require that each data item have a declared type. That is, you must specify the kind of information represented by the data, or the data's *type*. The data's type determines how data is represented in the computer's memory and what operations can be performed on the data.

Different programming languages support different data types. A data type can be something as fundamental as a type for representing integers or as complex as a type for representing a digital movie. Some examples of data types found in Java are

- `int`—for representing integers or whole numbers;
- `double`—for representing numbers having a fraction;
- `String`—for representing text;
- `Button`—for representing a push button in a graphical user interface; and
- `Point`—for representing points in a plane.

The types of data that are created, stored, and manipulated by Java programs can be separated into two main groups: *primitive types* and class types, or simply *classes*.

- There are eight primitive types:
- the numeric types `byte`, `short`, `int`, `long`, `float`, and `double` for storing numeric values;
- the character type `char` for storing a single alphabetic character, digit, or symbol; and
- the type `boolean` for storing `true` or `false`.

Primitive data values can be created by using literals such as `100`, `-10.456`, `'a'`, and `true`. Primitive values can also be operated on by using built-in operators such as `+` for addition and `-` for subtraction of two numeric values, producing a new primitive value. For example,

2 + 3

uses + (addition) to operate on the two numeric literal values, 2 and 3, to produce the new primitive value, 5.

Standard Java has more than 1500 classes. The `String`, `Button`, and `Point` types mentioned previously are standard Java classes. You will learn in Chapter 6, *Objects: Data Abstraction*, that you can create your own classes. Also, in Chapter 5, *Arrays And Containers*, we discuss arrays, which are a special case of class types.

The data values that are class types are called *objects*. You can create object data values by using the special operator `new` followed by the class name and possibly some additional values needed to create the new object. For example,

```
new Button("Quit")
```

creates a new object describing a button with the label "Quit."

You can create new objects of type `String`, as a special case, by using the string literal notation that surrounds the text of the string with double quotation marks. For example, `"Hello, world!"` creates a new string object.

In most cases, the operations supported for the particular class are given a name and invoked by placing the name after the object value, separated by a dot. For example,

```
"Hello, world!".length()
```

operates on the literal string `"Hello, world!"` and evaluates to 13—the number of characters in this string, including any blanks. Operations such as `length()` defined for a particular class are called *methods.* We discuss methods in great detail in subsequent chapters.

2.4.1 VARIABLES

In all but the most trivial programs, such as `HelloWorld`, you will declare *variables* that are identifiers used to refer to data values that are stored in the computer's memory. These are called variables because a variable actually refers to a particular place in the computer's memory, and the value stored in the computer's memory can vary as the program runs. A variable declaration always begins with a type and ends with a semicolon. The *type* is used to identify the kind of data that will be stored in the memory location associated with the variable being declared. Some examples of variable declarations are

```
int i, j;
String sentence;
boolean hot, cold, lukeWarm;
Button clickToExit;
```

Note that you can declare several variables of the same type by separating the names with a comma. Good choice of variable names is important to writing clearly understandable code. Stylistically, choose variable names that are meaningful. Also, variable names usually start in lowercase, and if they are multiword, the internal words start with an uppercase character, as in `clickToExit`.

2.4.2 VARIABLE INITIALIZATION

Variables can be given initial values. The preceding set of declarations given initial values for each variable becomes

```
int i = 2, j = 3;
String sentence = "I am a camera.";
boolean hot = true, cold = false, lukeWarm = false;
Button clickToExit = new Button("Exit");
```

Initializing variables with literals of their respective type is normal. In this example, the `int` variable `i` is initially given the value `2`. The `boolean` variable `hot` is initially `true`. The `String` variable `sentence` is initialized with a string literal.

With the exception of `String`, Java has no literals for creating object values. You initialize the `Button` variable by creating a `Button` object, using `new` as discussed briefly earlier. We discuss object creation in Chapter 6, *Objects: Data Abstraction*.

2.5 AN EXAMPLE: STRING CONCATENATION

The following example is a complete program that declares three variables: `word1`, `word2`, and `sentence`. Values are then assigned to the parts of the computer memory referred to by those variables.

```
// HelloWorld2.java - simple variable declarations
class HelloWorld2 {
  public static void main (String[] args) {
    String word1;   // declare a String variable
    String word2, sentence;  // declare two more
    word1 = "Hello, ";
    word2 = "world!";
    sentence = word1.concat(word2);
    System.out.println(sentence);
  }
}
```

Dissection of the *HelloWorld2* program

-
  ```
  String word1;   // declare a String variable
  String word2, sentence;  // declare two more
  ```

Whenever you introduce a new identifier you must declare it. You declare that an identifier is a variable by first writing the name of the kind of value the variable refers to, called the *type*. The type is `String` in this example. Insert the name of the new variable after the type. For variables in Java, the computer must always know the type of value to be stored in the memory location associated with that variable. As shown in the second line, you can declare more than one variable at a time by giving first the type and then a comma separated list of new identifiers.

-
  ```
  word1 = "Hello, ";
  word2 = "world!";
  ```

The symbol `=` is called the *assignment operator* and is used to store values in variables. Read the first statement as “word1 *gets* the value `Hello, ` ” or “word1 is *assigned* the value `Hello, ` ”. Here it is used to assign the `String` literals `"Hello, "` and `"world!"` to the newly declared variables `word1` and `word2`, respectively. The variable name will always be on the left and the new value to be assigned to the variable will always be on the right. Saying “assign the value `"Hello, "` to the variable `word1`” really means to store the `String` value `"Hello, "` in the computer memory location associated with the variable `word1`.

-
  ```
  sentence = word1.concat(word2);
  ```

This statement contains an expression used to create a third `String` value, which is then assigned to the variable `sentence`. This expression uses the method `concat()`, which is defined for values of type `String`. Recall that operations on objects are called methods. This particular operation requires a second `String` value placed between the parentheses. The method name `concat` is short for concatenation. The concatenation of two strings is a new string that contains the symbols from the first string followed by the symbols from the second string. Note that the first string, `word1`, contains a space at the end. Thus, when we concatenate the two strings, we get the new string `"Hello, world!"`, which is assigned to the variable `sentence`.

● `System.out.println(sentence);`

The program then prints out the string in the variable `sentence`, which now is `"Hello, world!"`.

As we showed earlier, when a variable is declared, it can be given an initial value. This approach essentially combines an assignment with the declaration. Using this notation, you can now write the body of `main()` in `HelloWorld2` as

```
String word1 = "Hello, ";
String word2 = "world!";
String sentence = word1.concat(word2);
System.out.println(sentence);
```

You could even combine two initializations in a single statement,

```
String word2 = "world!", sentence = word1.concat(word2);
```

although, as a general rule, multiple complex initializations such as this should be placed on separate lines.

2.5.1 STRINGS VERSUS IDENTIFIERS VERSUS VARIABLES

A *string* is a particular data value that a program can manipulate. A *variable* is a place in the computer's memory with an associated *identifier*. The following example uses the identifier `hello` to refer to a variable of type `String`. The identifier `stringVary` refers to a second variable of type `String`. We first assign `stringVary` the value associated with the `String` variable `hello`. Later we reassign it the string value `"hello"`.

```
//StringVsId.java - contrast strings and identifiers
class StringVsId {
  public static void main(String[] args) {
    String hello = "Hello, world!";
    String stringVary;
    stringVary = hello;
    System.out.println(stringVary);
    stringVary = "hello";
    System.out.println(stringVary);
  }
}
```

The output of this program is

```
Hello, world!
hello
```

The program demonstrates two important points. First it shows the difference between the identifier `hello`, which in fact refers to the string `"Hello, world!"`, and the string `"hello"`, which is referred to at one point by the variable `stringVary`. This example also shows that a variable can *vary*. The variable `stringVary` first refers to the value `"Hello, world!"` but later refers to the value `"hello"`.

2.6 USER INPUT

In the programs presented so far, we have generated output only by using System.out.println(). Most programs input some data as well as generate output. There are lots of ways to input data in Java, but the simplest is to use a class Scanner provided in the package java.util, as shown in the following example.

```
// Area.java -reading from the keyboard
import java.util.*;

class Area {
  public static void main (String[] args)
  {
    double width, height, area;
    Scanner scan = new Scanner(System.in);
    System.out.println("type two doubles for the width and height of a rectangle");
    width = scan.nextDouble();
    height = scan.nextDouble();
    assert (width > 0 & height > 0);
    area = width * height;
    System.out.print("The area is ");
    System.out.println(area);
  }
}
```

Dissection of the *Area* Program

- `import java.util.*;`

This line tells the Java compiler that the program SimpleInput uses some of the classes defined in the package java.util. The * indicates that you might use any of the classes in the package. The class used here is Scanner. This class allows you to do simple terminal input.

- `Scanner scan = new Scanner(System.in);`

This creates the variable scan that is associated with System.in. This makes the variable scan a proxy for using terminal input.

Here is a table of some of the methods used for getting input from the terminal using a Scanner. As you can see the basic data types have a corresponding *nextType()* method for reading and returning an input of that type. If the input is of the wrong type these methods fail and throw an exception. To avoid this, we can use methods of the form *hasNextType()* that return true if the next input can be correctly interpreted. An example would be hasNextInt() that checks that the input can be interpreted as an int.

SCANNER METHOD	EXPLANATION	EXAMPLES
nextInt()	Integers expected	123 -999 0
nextDouble()	Doubles expected	1.23 -0.01
next()	Arbitrary strings	"Java" "123"
nextBoolean()	Logical values	true false
Scanner()	Attaches source of input	new Scanner(System.in)

2.7 CALLING PREDEFINED METHODS

A *method* is a group of instructions having a name. In the programs introduced so far, we've defined a single method called `main()`. In addition, we've used some methods from other classes that were standard Java classes. The methods have names, which makes it possible for you to request the computer to perform the instructions that comprise the method. That's what we're doing by using the expression `System.out.println("Hello, world!")`. We're *calling* a method with the name `println` and asking that the instructions be executed. Just as two people can have the same name, two methods can have the same name. We must therefore, in general, tell Java where to look for the method. Using `System.out` tells Java that we're interested in the `println()` method associated with the object identified by `System.out`. (We must still postpone a full explanation of the meaning of `System.out`.) As we have shown, the same method can be called several times in one program. In `SimpleInput`, we called the method `println()` twice.

For many methods, we need to provide some data values in order for the method to perform its job. These values are provided to the method by placing them between parentheses following the name of the method. These values are called the *parameters* of the method, and we say that we are *passing* the parameters to the method. The `println()` method requires one parameter, which is the value to be printed. As we indicated previously, this parameter can be either a string or a numeric value. In the latter case the numeric value will be converted by the `println()` method to a string and then printed. If more than one parameter is required, we separate the parameter values by commas. For example, the predefined method `Math.min()` is used to determine the minimum of two numbers. The following program fragment will print out `3`.

```
int numberOne = 10, numberTwo = 3, smallest;
smallest = Math.min(numberOne, numberTwo);
System.out.println(smallest);
```

The method `min()` is contained in the standard Java class `Math`, which includes other common mathematical functions, such as `Math.sqrt()` that is used to find the square root of a number.

Here are some of the predefined methods that we've mentioned so far.

```
System.out.print(x)      //print the value of x
System.out.println(x)    //print the value of x followed by a newline
scan.nextInt()           //get an int from the keyboard
Math.min(x,y)            //find the smaller of x and y
Math.sqrt(x)             //find the square root of x
w1.concat(w2)            //concatenate the strings w1 and w2
word.length()            //find the length of the string word
```

Java includes a rich set of many more predefined methods. They include methods for the most common mathematical functions (see Appendix B, *Reference Tables*), for creating graphical user interfaces (see Chapter 8, *Graphical User Interfaces: Part I* and Chapter 9, *Graphical User Interfaces: Part II*), for reading and writing information from and to files (see Chapter 10, *Reading and Writing Files*), for communicating with programs on other computers using a network (see Chapter 13, *Threads: Concurrent Programming*), and many more. An important aspect of Java is the ability to use parts of programs created by others.

2.8 print(), println(), AND printf()

The print() and println() methods can print all the primitive types. The method System.out.println() is a convenient variant of System.out.print() which appends a newline to the end of the output. You can always achieve the same effect by adding \n to the end of what is being printed. For example, the following two lines are equivalent.

```
System.out.print("Hello, world!\n");
System.out.println("Hello, world!");
```

Recall, however, that you can't put a an actual newline in the middle of a string, so the following would not be legal.

```
System.out.println("type two integers for
            the width and height of a box");
```

In the same way that you can combine two strings with the string concatenation operator you can also combine one string and one value of any other type. In this case, the nonstring operand is first converted into a new string and then the two strings are concatenated. This allows rewriting the printing of the result in Area from Section 2.6, *User Input*, on page 23, in the form

```
System.out.println("The area is " + area);
```

This version emphasizes that one message will appear on a single line of the output. The double value area is first converted to a String and then combined with the other string, using string concatenation.

What about the opposite? What if you wanted to have an output message that spanned several output lines but with a single println()? You can do so by putting the symbols \n in a string literal to represent newlines within the string. Such a string will print on more than one output line. Thus for

```
System.out.println("One\nword\nper\nline.");
```

The output is

```
One
word
per
line.
```

The pair of symbols \n, when used in a string literal, mean to put a newline at this point in the string. This *escape sequence* allows you to escape from the normal meaning of the symbols \ and n when used separately. You can find out more about escape sequences in Section 2.9.3, *The char Type*, on page 29.

Care must be taken when you're using string concatenation to combine several numbers. Sometimes, parentheses are necessary to be sure that the + is interpreted as string concatenation, not numeric addition. For

```
int x = 1, y = 2;
System.out.println("x + y = " + x + y);
```

the output is x + y = 12 because the string "x + y =" is first concatenated with the string "1", and then "2" is concatenated onto the end of the string "x + y = 1". To print x + y = 3 you should use parentheses first to force the addition of x and y as integers and then concatenate the string representation of the result with the initial string, as in

```
System.out.println("x + y = " + (x + y));
```

2.8.1 FORMATTING OUTPUT WITH printf()

The method `printf()` was introduced in Java 5.0. It provides formatted output as is done in C. The `printf()` method is passed a list of arguments that can be thought of as

control_string and *other_arguments*

where *control_string* is a string and may contain conversion specifications, or formats. A conversion specification begins with a % character and ends with a conversion character. For example, in the format `%s` the letter `s` is the conversion character.

```
printf("%s", "abc");
```

The format `%s` causes the argument `"abc"` to be printed in the format of a string. Yet another way to do this is with the statement

```
printf("%c%c%c", 'a', 'b', 'c');
```

Single quotes are used to designate character constants. Thus, `'a'` is the character constant corresponding to the lowercase letter *a*. The format `%c` prints the value of an expression as a character. Notice that a constant by itself is considered an expression.

printf() conversion characters

Conversion character	How the corresponding argument is printed
`c`	as a character
`d`	as a decimal integer
`e`	as a floating-point number in scientific notation
`f`	as a floating-point number
`g`	in the `e`-format or `f`-format, whichever is shorter
`s`	as a string

When an argument is printed, the place where it is printed is called its *field* and the number of characters in its field is called its *field width*. The field width and a precision specifier can be specified in a format as two integers separated by a dot occurring between the % and the conversion character. Thus, the statement

```
printf("Airfare is $%6.2f.\nParking is $%6.2f.\n", 300.50, 16.50);
```

specifies that the two numbers should be printed in a field of 6 characters with two digits (the precision) to the right of the decimal point. It will print

```
Airfare is $300.50.
Parking is $ 16.50.
```

We will explain the various detailed conventions for using formats in Chapter 10, on page 304

.

2.9 NUMBER TYPES

There are two basic representations for numbers in most modern programming languages: integer representations and floating point representations. The integer types are used to represent integers or whole numbers. The floating point types are used to represent numbers that contain fractional parts or for numbers whose magnitude exceeds the capacity of the integer types.

2.9.1 THE INTEGER TYPES

A *bit* is the smallest unit of information that can be stored in a computer. It can represent only two values, such as 1/0 or on/off or true/false. A sequence of bits can be used to represent a larger range of values. For the integer types, the bits are interpreted with the binary number system. The binary number system has only two digits: 0 and 1. These two digits can be combined to represent any number in the same way that the 10 digits of the normal, decimal number system can be combined to form any number. (See Appendix A, *Getting Down to the Bits*).

Some mechanism is needed to represent positive and negative numbers. One simple solution would be to set aside one bit to represent the sign of the number. However, this results in two different representations of zero, +0, and −0, which causes problems for computer arithmetic. Therefore an alternative system called two's complement is used.

An 8-bit value called a *byte* can represent 256 different values. Java supports five integral numeric types. The type `char`, although normally used to represent symbolic characters in a 16-bit format called Unicode, can be interpreted as an integer. As discussed shortly, the result of combining a `char` value with another numeric value will never be of type `char`. The `char` value is first converted to one of the other numeric types. These types are summarized in the following table.

TYPE	NUMBER OF BITS	RANGE OF VALUES
`byte`	8	`-128 to 127`
`short`	16	`-32768 to 32767`
`char`	16	`0 to 65536`
`int`	32	`-2147483648 to 2147483647`
`long`	64	`-9223372036854775808 to 9223372036854775807`

The asymmetry in the most negative value and the most positive value of the integer types results from the two's complement system used in Java to represent negative values.

Literal integer values are represented by a sequence of digits, possibly preceded by a minus sign. An integer literal is either an `int` or `long`. An explicit conversion, called a *cast*, of an `int` literal must be used to assign them to the smaller integer types `short` and `byte` (see Section 2.10.2, *Type Conversion*, on page 32). Integer literals can be expressed in base 10 (decimal), base 8 (octal), and base 16 (hexadecimal) number systems. Decimal literals begin with a digit `1-9`, octal literals begin with the digit `0`, and hexadecimal literals begin with the two-character sequence `0x`. To specify a `long` literal instead of an `int` literal, append the letter *el*, uppercase or lowercase, to the sequence of digits.

Here are some examples:

- `217`—the decimal value two hundred seventeen

- `0217`—an octal number equivalent to 143 in the decimal system

  $(2 \times 8^2 + 1 \times 8^1 + 7)$

- `0195` would be illegal because 9 is not a valid octal digit (only 0-7 are octal digits)

- 0x217—a hexadecimal number equivalent to 535 in the decimal system

  $(2 \times 16^2\ 1\ \ 1 \times 16^1 + 7)$;

  the hexadecimal digits include 0–9 plus $a = 10$, $b = 11$, $c = 12$, $d = 13$, $e = 14$, and $f = 15$.

- `14084591234L`—a `long` decimal literal; without the `L` this would be an error because `14084591234` is too large to store in the 32 bits of an `int` type integer.

2.9.2 THE FLOATING POINT TYPES

Java supports two floating point numeric types. A floating point number consists of three parts: a sign, a magnitude, and an exponent. These two types are summarized in the following table.

TYPE	NUMBER OF BITS	APPROXIMATE RANGE OF VALUES	APPROXIMATE PRECISION
`float`	32	$+/-10^{-45}$ to $+/-10^{+38}$	7 decimal digits
`double`	64	$+/-10^{-324}$ to $+/-10^{+308}$	15 decimal digits

To represent floating point literals, you can simply insert a decimal point into a sequence of digits, as in `3.14159`. If the magnitude of the number is too large or too small to represent in this fashion, then a notation analogous to scientific notation is used. The letter `e` from *exponent* is used to indicate the exponent of 10 as shown in the following examples.

JAVA REPRESENTATION	VALUE
`2.17e-27`	2.17×10^{-27}
`2.17e99`	2.17×10^{99}

Unless specified otherwise, a floating point literal is of type `double`. To specify a floating point literal of type `float`, append either `f` or `F` to the literal—for example, `2.17e-27f`.

2.9.3 THE `char` TYPE

Java provides `char` variables to represent and manipulate characters. This type is an integer type and can be mixed with the other integer types. Each `char` is stored in memory in 2 bytes. This size is large enough to store the integer values 0 through 65535 as distinct character codes or nonnegative integers, and these codes are called Unicode. Unicode uses more storage per character than previous common character encodings because it was designed to represent all the world's alphabets, not just one particular alphabet.

For English, a subset of these values represents actual printing characters. These include the lowercase and uppercase letters, digits, punctuation, and special characters such as `%` and `+`. The character set also includes the white space characters blank, tab, and newline. This important subset is represented by the first 128 codes, which are also known as the ASCII codes. Earlier languages, such as C and C++, worked only with this more limited set of codes and stored them in 1 byte.

The following table illustrates the correspondence between some character literals and integer values. Character literals are written by placing a single character between single quotes, as in `'a'`.

Some Character Constants and Their Corresponding Integer Values	
Character constants	`'a'  'b'  'c' ... 'z'`
Corresponding values	`97  98  99 ...  122`
Character constants	`'A'  'B'  'C' ... 'Z'`
Corresponding values	`65  66  67 ...  90`
Character constants	`'0' '1' '2' ... '9'`
Corresponding values	`48  49  50 ... 57`
Character constants	`'&'  '*'  '+'`
Corresponding values	`38  42  43`

There is no particular relationship between the value of the character constant representing a digit and the digit's intrinsic integer value. That is, the value of `'2'` is *not* 2. The property that the values for `'a'`, `'b'`, `'c'`, and so on occur in order is important. It makes the sorting of characters, words, and lines into lexicographical order convenient.

Note that character literals are different from string literals, which use double quotes, as in `"Hello"`. String literals can be only one character long, but they are still `String` values, not `char` values. For example, `"a"` is a string literal.

Some nonprinting and hard-to-print characters require an escape sequence. The horizontal tab character, for example, is written as `\t` in character constants and in strings. Even though it is being described by the two characters `\` and `t`, it represents a single character. The backslash character `\` is called the *escape character* and is used to escape the usual meaning of the character that follows it. Another way to write a character constant is by means of a hexadecimal-digit escape sequence, as in `'\u0007'`. This is the alert character, or the audible bell. These 4 hexadecimal digits are prefixed by the letter `u` to indicate their use as a Unicode literal. The 65,536 Unicode characters can be written in hexadecimal form from '*u0000*' to `'\uFFFF'`.

The following table contains some nonprinting and hard-to-print characters.

Name of Character	Escape	INT	HEX
Backslash	`\\`	92	`\u005C`
Backspace	`\b`	8	`\u0008`
Carriage return	`\r`	13	`\u000D`
Double quote	`\"`	34	`\u0022`
Formfeed	`\f`	12	`\u000C`
Horizontal tab	`\t`	9	`\u0009`
Newline	`\n`	10	`\u000A`
Single quote	`\'`	39	`\u0027`
Null character		0	`\u0000`
Alert		7	`\u0007`

The alert character is special; it causes the bell to ring. To hear the bell, try executing a program that contains the line

```
System.out.print('\u0007');
```

Character values are small integers, and, conversely, small integer values can be characters. Consider the declaration

```
char   c = 'a';
```

The variable c can be printed either as a character or as an integer:

```
System.out.printf("%c",c);         // a is printed
System.out.printf("%d",c);         // 97 is printed
```

2.9.4 NUMBERS VERSUS STRINGS

The sequence of 0s and 1s inside a computer used to store the `String` value `"1234"` is different from the sequence of 0s and 1s used to store the `int` value `1234`, which is different from the sequence of 0s and 1s used to store the `double` value `1234.0`. The string form is more convenient for certain types of manipulation, such as printing on the screen or for combining with other strings. The `int` form is better for some numeric calculations, and the `double` form is better for others. So how does the computer know how to interpret a particular sequence of 0s and 1s? The answer is: Look at the type of the variable used to store the value. This answer is precisely why you must specify a type for each variable. Without the type information the sequence of 0s and 1s could be misinterpreted.

We do something similar with words all the time. Just as computers interpret sequences of 0s and 1s, human beings interpret sequences of alphabetic symbols. How people interpret those symbols depends on the context in which the symbols appear. For example, at times the same word can be a noun, and at other times it can be a verb. Also, certain sequences of letters mean different things in different languages. Take the word *pie* for example. What does it mean? If it is English, it is something good to eat. If it is Spanish, it means foot.

From the context of the surrounding words we can usually figure out what the type of the word is: verb or noun, English or Spanish. Some programming languages do something similar and "figure out" the type of a variable. These programming languages are generally considered to be more error prone than languages such as Java, which require the programmer to specify the type of each variable. Languages such as Java are called *strongly typed* languages.

2.10 ARITHMETIC EXPRESSIONS

The basic arithmetic operators in Java are addition `+`, subtraction `-`, multiplication `*`, division `/`, and modulus `%`. You can use all arithmetic operators with all primitive numeric types: `char`, `byte`, `short`, `int`, `long`, `float`, and `double`. In addition, you can combine any two numeric types by using these operators in what is known as *mixed mode arithmetic*. Although you can use the operators with any numeric type, Java actually does arithmetic only with the types `int`, `long`, `float`, and `double`. Therefore the following rules are used first to convert both operands into one of four types.

1. If either operand is a `double`, then the other is converted to `double`.
2. Otherwise, if either operand is a `float`, then the other is converted to `float`.
3. Otherwise, if either operand is a `long`, then the other is converted to a `long`.
4. Otherwise, both are converted to `int`.

This conversion is called *binary numeric promotion* and is also used with the binary relational operators discussed in Section 3.2.1, *Relational and Equality Operators*, on page 49.

When both operands are integer types, the operations of addition, subtraction, and multiplication are self-evident except when the result is too large to be represented by the type of the operands.

Integer values can't represent fractions. In Java, integer division truncates toward 0. For example, `6 / 4` is `1` and `6 / (-4)` is `-1`. A common mistake is to forget that integer division of nonzero values can result in 0. To obtain fractional results you must force one of the operands to be a floating point type. In expressions involving literals you can do so by adding a decimal point, as in `6.0 / 4`, which results in the floating point value `1.5`. In addition, integer division by zero will result in an error called an `ArithmeticException`. An *exception*, as the name implies, is something unexpected. Java provides a way for you to tell the computer what to do when exceptions occur. If you don't do anything and such an error occurs, the program will print an appropriate error message and terminate. We discuss exceptions in Chapter 11, *Coping with Errors*.

Unlike some programming languages, Java doesn't generate an exception when integer arithmetic results in a value that is too large. Instead, the extra bits of the true result are lost, and in some cases pollute the bit used for the sign. For example, adding two very large positive numbers could generate a negative result. Likewise, subtracting a very large positive number from a negative number could generate a positive result. If values are expected to be near the limit for a particular type, you should either use a larger type or check the result to determine whether such an overflow has occurred.

When one of the operands is a floating point type, but both operands are not of the same type, one of them is converted to the other as described earlier. Unlike some programming languages, floating point arithmetic operations in Java will never generate an exception. Instead three special values can result: positive infinity, negative infinity, and "Not a Number." See Appendix A, *Getting Down to the Bits* for more details.

The modulus operator `%` returns the remainder from integer division. For example, `16 % 3` is `1`, because 16 divided by 3 is 5 with a remainder of 1. The modulus operator is mostly used with integer operands; however, in Java it can be used with floating point operands. For floating point values `x % y` is `n`, where `n` is the largest integer such that `|y| * n` is less than or equal to `|x|`.

2.10.1 AN INTEGER ARITHMETIC EXAMPLE: CHANGE

The computation in Section 1.3, *Implementing Our Algorithm in Java*, on page 5, whereby we made change for a dollar, is a perfect illustration of the use of the two integer division operators, `/` and `%`.

```
// MakeChange.java - change in dimes and pennies
import java.util.*;

class MakeChange {
  public static void main (String[] args) {
    int price, change, dimes, pennies;
    Scanner scan = new Scanner(System.in);
    System.out.println("type price (0 to 100):");
    price = scan.nextInt();
    change = 100 - price;        //how much change
    dimes = change / 10;         //number of dimes
    pennies = change % 10;       //number of pennies
    System.out.print("The change is : ");
    System.out.println(dimes + " dimes " + pennies + " pennies");
  }
}
```

Dissection of the *MakeChange* Program

● `int price, change, dimes, pennies;`

This program declares four integer variables. These types determine the range of values that can be used with these variables. They also dictate that this program use integer arithmetic operations, not floating point operations.

● `price = scan.nextInt();`

The `scan.nextInt()` is used to obtain the input from the keyboard. The `nextInt()` method is a member of the class `Scanner`. At this point we must type in an integer price. For example, we would type 77 and hit Enter.

- `change = 100 - price; //how much change`

This line computes the amount of change. This is the integer subtraction operator.

-
  ```
  dimes = change / 10;        //number of dimes
  pennies = change % 10;      //number of pennies
  ```

To compute the number of dimes, we compute the integer result of dividing change by 10 and throw away any remainder. So if `change` is 23, then the integer result of 23/10 is 2. The remainder 3 is discarded. The number of pennies is the integer remainder of `change` divided by 10. The `%` operator is the integer remainder or modulo operator in Java. So if `change` is 23, then `23 % 10` is 3. In Exercise 17 on page 43, we ask you to use `double` for the variables and to report on your results.

2.10.2 TYPE CONVERSION

You may want or need to convert from one primitive numeric type to another. As mentioned in the preceding section, Java will sometimes automatically convert the operands of a numeric operator. These automatic conversions are also called *widening primitive conversions*. These conversions always convert to a type that requires at least as many bits as the type being converted, hence the term *widening*. In most but not all cases, a widening primitive conversion doesn't result in any loss of information. An example of a widening primitive conversion that can lose some precision is the conversion of the `int` value 123456789 to a `float` value, which results in 123456792. To understand this loss of information, see Appendix A, *Getting Down to the Bits*. The following are the possible widening primitive conversions:.

FROM	TO
`byte`	`short`, `int`, `long`, `float`, or `double`
`short`	`int`, `long`, `float`, or `double`
`char`	`int`, `long`, `float`, or `double`
`int`	`long`, `float`, or `double`
`long`	`float` or `double`
`float`	`double`

In addition to performing widening conversions automatically as part of mixed mode arithmetic, widening primitive conversions are also used to convert automatically the right-hand side of an assignment operator to the type of the variable on the left. For example, the following assignment will automatically convert the integer result to a floating point value.

```
int x = 1, y = 2;
float z;
z = x + y;    // automatic widening from int to float
```

A *narrowing primitive conversion* is a conversion between primitive numeric types that may result in significant information loss. The following are narrowing primitive conversions.

FROM	TO
byte	char
short	byte or char
char	byte or short
int	byte, short, or char
long	byte, short, char, or int
float	byte, short, char, int, or long
double	byte, short, char, int, long, or float

Narrowing primitive conversions generally result only from an explicit type conversion called a *cast*. A cast is written as (*type*)*expression*, where the expression to be converted is preceded by the new type in parentheses. A cast is an operator and, as the table in Section 2.13, *Precedence and Associativity of Operators*, on page 38, indicates, has higher precedence than the five basic arithmetic operators. For example, if you are interested only in the integer portion of the floating point variable someFloat, then you can store it in someInteger, as in

```
int someInteger;
float someFloat = 3.14159;
someInteger = (int)someFloat;
System.out.println(someInteger);
```

The output is

```
3
```

If the cast is between two integral types, the most significant bits are simply discarded in order to fit the resulting format. This discarding can cause the result to have a different sign from the original value. The following example shows how a narrowing conversion can cause a change of sign.

```
int i = 127, j = 128;
byte iAsByte = (byte)i, jAsByte = (byte)j;
System.out.println(iAsByte);
System.out.println(jAsByte);
```

The output is

```
127
-128
```

The largest positive value that can be stored in a byte is 127. Attempting to force a narrowing conversion on a value greater than 127 will result in the loss of significant information. In this case the sign is reversed. To understand exactly what happens in this example, see Appendix A.

COMMON PROGRAMMING ERROR

Remember that integer division truncates toward zero. For example, the value of the expression 3/4 is 0. Both the numerator and the denominator are integer literals, so this is an integer division. If what you want is the rounded result, you must first force this to be a floating point division and then use the routine `Math.round()` to round the floating point result to an integer. To force a floating point division you can either make one of the literals a floating point literal or use a cast. In the following example first recall that floating point literals of type `float` are specified by appending the letter `f`. Then for

```
int x = Math.round(3.0f/4);
```

the variable `x` will get the value 1. Forcing the division to be a floating point divide is not enough. In the following example, `z` will be 0.

```
int z = (int)(3.0f/4);
```

The conversion of 0.75 to an `int` truncates any fractional part.

2.11 ASSIGNMENT OPERATORS

To change the value of a variable, we have already made use of assignment statements such as

```
a = b + c;
```

Assignment is an operator, and its precedence is lower than all the operators we've discussed so far. The associativity for the assignment operator is right to left. In this section we explain in detail its significance.

To understand = as an operator, let's first consider + for the sake of comparison. The binary operator + takes two operands, as in the expression a + b. The value of the expression is the sum of the values of a and b. By comparison, a simple assignment expression is of the form

variable = *rightHandSide*

where *rightHandSide* is itself an expression. A semicolon placed at the end would make this an assignment statement. The assignment operator = has the two operands *variable* and *rightHandSide*. The value of *rightHandSide* is assigned to *variable,* and that value becomes the value of the assignment expression as a whole. To illustrate, let's consider the statements

```
b = 2;
c = 3;
a = b + c;
```

where the variables are all of type `int`. By making use of assignment expressions, we can condense these statements to

```
a = (b = 2) + (c = 3);
```

The assignment expression `b = 2` assigns the value `2` to the variable `b`, and the assignment expression itself takes on this value. Similarly, the assignment expression `c = 3` assigns the value `3` to the variable `c`, and the assignment expression itself takes on this value. Finally, the values of the two assignment expressions are added, and the resulting value is assigned to `a`.

Although this example is artificial, in many situations assignment occurs naturally as part of an expression. A frequently occurring situation is *multiple assignment*. Consider the statement

```
a = b = c = 0;
```

Because the operator = associates from right to left, an equivalent statement is

```
a = (b = (c = 0));
```

First, c is assigned the value 0, and the expression c = 0 has value 0. Then b is assigned the value 0, and the expression b = (c = 0) has value 0. Finally, a is assigned the value 0, and the expression a = (b = (c = 0)) has value 0.

In addition to =, there are other assignment operators, such as += and -=. An expression such as

```
k = k + 2
```

will add 2 to the old value of k and assign the result to k, and the expression as a whole will have that value. The expression

```
k += 2
```

accomplishes the same task. The following list contains all the assignment operators

.Assignment Operators						
=	+=	-=	*=	/		
=	%=	>>=	<<=	&=	^=	|=

All these operators have the same precedence, and all have right-to-left associativity. The meaning is specified by

variable op= expression

which is equivalent to

variable = *variable op* (*expression*)

with the exception that if *variable* is itself an expression, it is evaluated only once. Note carefully that an assignment expression such as

j *= k + 3   is equivalent to   j = j * (k + 3)

rather than

```
j = j * k + 3
```

The following table illustrates how assignment expressions are evaluated.

Declarations and Initializations			
int i = 1, j = 2, k = 3, m = 4;			
Expression	Equivalent Expression	Equivalent Expression	Value
i += j + k	i += (j + k)	i = (i + (j + k))	6
j *= k = m + 5	j *= (k = (m + 5))	j = (j * (k = (m + 5)))	18

2.12 THE INCREMENT AND DECREMENT OPERATORS

Computers are very good at counting. As a result, many programs involve having an integer variable that takes on the values 0, 1, 2, . . . One way to add 1 to a variable is

```
i = i + 1;
```

which changes the value stored in the variable `i` to be 1 more than it was before this statement was executed. This procedure is called *incrementing* a variable. Because it is so common, Java, like its predecessor C, includes a shorthand notation for incrementing a variable. The following statement gives the identical result.

```
i++;
```

The operator `++` is known as the increment operator. Similarly, there is a decrement operator, `--`, so that the following two statements are equivalent:

```
i = i - 1;
i--;
```

Here is a simple program that demonstrates the increment operator.

```
//Increment.java - demonstrate incrementing
class Increment {
  public static void main(String[] args) {
    int i = 0;
    System.out.println("i = " + i);
    i = i + 1;
    System.out.println("i = " + i);
    i++;
    System.out.println("i = " + i);
    i++;
    System.out.println("i = " + i);
  }
}
```

The output of this program is

```
i = 0
i = 1
i = 2
i = 3
```

Note that both increment and decrement operators are placed after the variable to be incremented. When placed after its argument they are called the *postfix* increment and *postfix* decrement operators. These operators also can be used before the variable. They are then called the *prefix* increment and *prefix* decrement operators. Each of the expressions `++i` *(prefix)* and `i++` *(postfix)* has a value; moreover, each causes the stored value of `i` in memory to be incremented by 1. The expression `++i` causes the stored value of `i` to be incremented first, with the expression then taking as its value the new stored value of `i`. In contrast, the expression `i++` has as its value the current value of `i`; then the expression causes the stored value of `i` to be incremented. The following code illustrates the situation.

```
int   a, b, c = 0;
a = ++c;
b = c++;
System.out.println("a = " + a);   //a = 1 is printed
System.out.println("b = " + b);   //b = 1 is printed
System.out.println("c = " + ++c); //c = 3 is printed
```

Similarly, `--i` causes the stored value of `i` in memory to be decremented by 1 first, with the expression then taking this new stored value as its value. With `i--` the value of the expression is the current value of `i`; then the expression causes the stored value of `i` in memory to be decremented by 1. Note that `++` and `--` cause the value of a variable in memory to be changed. Other operators do not do so. For example, an expression such as `a + b` leaves the values of the variables `a` and `b` unchanged. These ideas are expressed by saying that the operators `++` and `--` have a *side effect*; not only do these operators yield a value, but they also change the stored value of a variable in memory. Of course this is also true of the assignment operator.

In some cases we can use `++` in either prefix or postfix position, with both uses producing equivalent results. For example, each of the two statements

`++i;` and `i++;`

is equivalent to

```
i = i + 1;
```

In simple situations you can consider `++` and `--` as operators that provide concise notation for the incrementing and decrementing of a variable. In other situations, you must pay careful attention as to whether prefix or postfix position is used.

2.13 PRECEDENCE AND ASSOCIATIVITY OF OPERATORS

When evaluating expressions with several operators, you need to understand the order of evaluation of each operator and its arguments. *Operator precedence* gives a hierarchy that helps determine the order in which operations are evaluated. For example, precedence determines which arithmetic operations are evaluated first in

```
x = (-b + Math.sqrt(b * b - 4 * a * c))/(2 * a)
```

which you may recognize as the expression to compute one of the roots of a quadratic equation, written like this in your mathematics class:

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

To take a simpler example, what is the value of integer variable `x` after the assignment `x = 7 + 5 * 3`? The answer is `22`, not $(7 + 5) \times 3$ which is 36. The reason is that multiplication has *higher precedence* than addition; therefore `5 * 3` is evaluated before `7` is added to the result.

In addition to some operators having higher precedence than others, Java specifies the *associativity* or the order in which operators of equal precedence are to be evaluated. For example, the value of `100 / 5 * 2` is 40, not 10. This is because `/` and `*` have equal precedence and arithmetic operators of equal precedence are evaluated from left to right. If you wanted to do the multiplication before the division, you would write the expression as `100 / (5 * 2)`. Parentheses can be used to override the normal operator precedence rules.

This left to right ordering is important in some cases that might not appear obvious. Consider the expression `x + y + z`. From simple algebra, the associativity of addition tells us that `(x + y) + z` is the same as `x + (y + z)`. Unfortunately, this is true only when you have numbers with infinite precision. Suppose that `y` is the largest positive integer that Java can represent, `x` is `-100`, and `z` is `50`. Evaluating `(y + z)` first will result in integer overflow, which in Java will be equivalent to some very large negative number, clearly not the expected result. If instead, `(x + y)` is evaluated first, then adding `z` will result in an integer that is still in range and the result will be the correct value.

The precedence and associativity of all Java operators is given in Appendix Appendix B, *Reference Tables*. The following table gives the rules of precedence and associativity for the operators of Java that we have used so far.

OPERATOR PRECEDENCE AND ASSOCIATIVITY	
OPERATOR	**ASSOCIATIVITY**
`( )` `++` (postfix) `--` (postfix)	Left to right
`+` (unary) `-` (unary) `++` (prefix) `--` (prefix)	Right to left
`new` (*type*)*expr*	Right to left
`*` `/` `%`	Left to right
`+` `-`	Left to right
`=` `+=` `-=` `*=` `/=` *etc.*	Right to left

All the operators on the same line, such as `*`, `/`, and `%`, have equal precedence with respect to each other but have higher precedence than all the operators that occur on the lines below them. The associativity rule for all the operators on each line appears in the right-hand column.

In addition to the binary `+`, which represents addition, there is a unary `+`, and both operators are represented by a plus sign. The minus sign also has binary and unary meanings. The following table gives some additional examples of precedence and associativity of operators.

DECLARATIONS AND INITIALIZATIONS		
`int a = 1, b = 2, c = 3, d = 4;`		
EXPRESSION	**EQUIVALENT EXPRESSION**	**VALUE**
`a * b / c`	`(a * b) / c`	0
`a * b % c + 1`	`((a * b) % c) + 1`	3
`++a * b - c--`	`((++a) * b) - (c --)`	1
`7 - - b * ++d`	`7 - ((- b) * (++d))`	17

2.14 PROGRAMMING STYLE

A clear, consistent style is important to writing good code. We use a style that is largely adapted from the Java professional programming community. Having a style that is readily understandable by the rest of the programming community is important.

We’ve already mentioned the importance of comments for documenting a program. Anything that aids in explaining what is otherwise not clear in the program should be placed in a comment. Comments help the programmer keep track of decisions made while writing the code. Without good documentation, you may return to some code you have written, only to discover that you have forgotten why you did some particular thing. The documentation should enable someone other than the original programmer to pick up, use, and modify the code. All but the most trivial methods should have comments at the beginning, clearly stating the purpose of the method. Also, complicated blocks of statements should be preceded by comments summarizing the function of the block. Comments should add to the clarity of the code, not simply restate the program, statement by statement. Here is an example of a useless comment.

```
area = width * height;      // compute the area
```

Good documentation includes proper choice of identifiers. Identifiers should have meaningful names. Certain simple one-character names are used to indicate auxiliary variables, such as `i`, `j`, or `k`, as integer variables.

The code should be easy to read. Visibility is enhanced by the use of white space. In general, we present only one statement to a line and in all expressions separate operators from arguments by a space. As we progress to more complex programs, we shall present, by example or explicit mention, accepted layout rules for program elements.

Some naming conventions used by many Java programmers

- Class names start with uppercase and embedded words, as in `HelloWorld`, are capitalized.
- Methods and variables start with lowercase and embedded words, as in `readInt`, `data`, `toString`, and `loopIndex`, are capitalized.
- Although legal, the dollar sign, `$`, should not be used except in machine-generated Java programs.

Summary

- To create a Java program first define a class. Give the class a method called `main()`. Put whatever instructions you want the computer to execute inside the body of the method `main()`.
- A program stores data in variables. Each variable is given a type, such as `int` for storing integers or `String` for storing strings.
- You can use literals to embed constant values of various types in a program, such as in the constant string `"Hello, world!"` or the integer constant `3`.
- You can combine literals and variables in expressions by using operators such as `+` for addition or string concatenation and `*` for numeric multiplication.
- You can store the result of evaluating an expression in a variable by using the assignment operator `=`. The variable is always on the left, and the expression being assigned to the variable is always on the right.
- You can call a method from another class by writing the name of the class followed by a dot and the name of the method—for example, `Math.sqrt()`.
- The lexical elements of a Java program are keywords, identifiers, literals, operator symbols, punctuation, comments, and white space.
- You can print strings and numbers to the screen by using the method `System.out.print()` or `System.out.println()`. The latter appends a newline to whatever is printed. These methods are part of the standard Java classes. Java 5.0 introduced `printf()` which allows formatted printing as derived from the C language.
- You can input from the keyboard by using the methods of class `Scanner`. These include `next()` for strings, `nextInt()` for integers and `nextDouble()` for doubles.
- Java supports the primitive integer types `char`, `byte`, `short`, `int`, and `long`. It also supports two floating point types, `float` and `double`.

- Integer division truncates toward zero—it doesn't round to the nearest whole number. You can use `Math.round()` if rounding is what you want.

Review Questions

1. What line appears in every complete Java program indicating where to begin executing the program?
2. What one-line instruction would you use to have a Java program print `Goodbye`?
3. What affect do strings such as `/* what is this */` have on the execution of a Java program?
4. What is a variable?
5. What is a method?
6. Complete the following table.

Text	Legal ID	Why or Why Not
`3xyz`	No	Digit is first character.
`xy3z`		
`a = b`		
`main`		
`Count`		
`class`		

7. What does the symbol `=` do in Java?
8. Programmers say ___________ a method when they mean go and execute the instructions for the method.
9. True or false? A multiline comment can be placed anywhere white space could be placed.
10. True or false? Keywords can also be used as variables, but then the special meaning of the keyword is overridden.
11. What convention for identifiers given in this chapter is used in `whatAmI`, `howAboutThis`, `someName`?
12. What primitive types are used to store whole numbers?
13. What is the difference between `x = 'a'` and `x = a`?
14. What is the difference between `s = "hello"` and `s = hello`?
15. Which version of the Java program `HelloWorld` is the one you can view and edit with a text editor, *HelloWorld.java* or *HelloWorld.class*? What does the other one contain? What program created the one you do not edit?
16. What is Unicode?
17. List the primitive types in Java.
18. Before it can be used, every variable must be declared and given a ___________.

19. What is the value of the Java expression `"10"+"20"`? Don't ignore the quotation marks; they are crucial.

20. Write a Java statement that could be used to read an integer value from the keyboard and store it in the variable `someNumber`.

21. What is wrong with the following Java statement?

```
System.out.println("This statement is supposed
     to print a message. What is wrong?");
```

22. Every group of input statements should be preceded by what?

23. How do you write `x` times `y` in Java?

24. What is the difference between `System.out.print("message")` and `System.out.println("message")`?

25. Write a *single* Java statement that will produce the following output.

```
X
XX
XXX
```

26. Approximately, what is the largest value that can be stored in the primitive type `int`? One thousand? One million? One billion? One trillion? Even larger?

27. What primitive Java type can store the largest numbers?

28. What is the value of the following Java expressions?

```
20 / 40
6 / 4
6.4 / 2
```

EXERCISES

1. Write a Java program that prints "Hello *your name*." You can do this by a simple modification to the `HelloWorld` program. Compile and run this program on your computer.

2. Write a Java program that prints a favorite poem of at least eight lines. Be sure to print it out neatly aligned. At the end of the poem, print two blank lines and then the author's name.

3. Design your own signature logo, such as a sailboat icon if you like sailing, and print it followed by "yours truly—*your name*." A sailboat signature logo might look like

```
         /\
        /  \
       /    \
      /      \
        |
=================
\  Yours truly    /
   Bruce McPohl
```

4. Write a Java program to read in two numbers and print the sum. Be sure to include a message to prompt the user for input and a message identifying the output. See what happens if you type in something that is not a number when the program is run. See how large a number you can type in and still have the program work correctly.

5. The following code contains three syntax errors and produces two syntax error messages from *javac*. Fix the problems.

```
// Ch2e1.java - fixing syntax errors
Class Ch2e1 {
  public static void main(String[] args) {
    System.out.println(hello, world);
  }
```

The *javac* compiler's message reads:

```
Ch2e1.java:2: Class or interface declaration
expected.
 Class ch2e1 {
 ^
Ch2e1.java:7: Unbalanced parentheses.
^
2 errors
```

6. The following code produces one syntax error message from *javac*. Fix the problem.

```
// Ch2e2.java - more syntax errors
class Ch2e2 {
  public static void main(String[] args) {
    int count = 1, i = 3,
    System.out.println("count + i = ", count + i);
  }
}
```

The *javac* compiler's message reads

```
Ch2e2.java:6: Invalid declaration.
System.out.println("count + i = ", count + i);

                  ^
1 error
```

Here, unlike the previous exercise, the compiler doesn't as clearly point to the errors. Frequently, errors in punctuation lead to syntax error messages that are hard to decipher. After you fix the first syntax error in the code, a second error will be identified.

7. Continue with the code class `Ch2e2`. If you fixed just the syntax errors, you may get a running program that still has a run-time bug. Namely, the output may not be the sum of `count + i`. Fixing run-time or semantic bugs is harder than fixing syntax bugs because something is wrong with your understanding of how to program the solution. Without introducing any other variables fix the run-time bug.

8. Write a program that draws a box like the one shown, using a *single* `println()` statement.

```
**********
*        *
*        *
**********
```

9. Use `scan.nextDouble()` to read in one double precision floating point number and then print the results of calling `Math.sin()`, `Math.cos()`, `Math.asin()`, `Math.exp()`, `Math.log()`, `Math.floor()`, and `Math.round()` with the input value as a parameter. Be sure to prompt the user for input and label the output.

10. Write a program to read two double precision floating point numbers, using `nextDouble()`. Print the sum, difference, product, and quotient of the two numbers. Try two very small numbers, two very large numbers, and one very small number with one very large number. You can use the same notation used for literals to enter the numbers. For example, `0.123e-310` is a very small number.

11. Write a program to compute the area of a circle given its radius. Let `radius` be a variable of type `double` and use `nextDouble()` to read in its value. Be sure that the output is understandable. The Java class `Math` contains definitions for the constants `E` and `PI`, so you can use `Math.PI` in your program.

12. Extend the previous program to write out the circumference of a circle and the volume of a sphere given the radius as input. Recall that the volume of a sphere is

$$V = \frac{4 \times \pi r^3}{3}$$

13. Write a program that asks for a `double` and then prints it out. Then ask for a second `double`, this time printing out the sum and average of the two `double`s. Then ask for a third double and again print out the accumulated sum and the average of the three doubles. Use variables `data1`, `data2`, `data3`, and `sum`. Later, when we discuss loops, you will see how this is easily done for an arbitrary number of input values.

14. Write a program that reads in an integer and prints it as a character. Remember that character codes can be nonprinting. ‘

15. Write a program that asks for the number of quarters, dimes, nickels, and pennies you have. Then compute the total value of your change and print the number of dollars and the remaining cents.

16. Write a program capable of converting one currency to another. For example, given U.S. dollars it should print out the equivalent number of Euros. Look up the exchange rate and use it as input.

17. Change the `MakeChange` program to use variables that are `double`s. Run the program and see what goes wrong.

```
class MakeChange {
  public static void main (String[] args) {
    double price, change, dimes, pennies;
    ...
```

18. The following is a C program for printing “Hello, world!”.

```
/* Hello World In C
 * Purpose:
 *   The classic "Hello, world!" program.
 *   It simply prints a message to the screen.
 * Author:
 *    Jane Programmer
 *       as derived from Kernighan and Richie
 */
```

```
#include <stdio.h>     /* needed for IO */
int main(void) {
  printf("Hello, world!\n");
  return 0;           /* unneeded in Java */
}
```

Note how similar this program is to the Java version. A key difference is the lack of class encapsulation of `main()`. As in Java, `main()` starts the program's execution. In C, methods are known as functions. The `printf()` function is found in the standard input-output library imported by the C compiler for use in this program. The `return 0` ends program execution and is not used in Java. Convert the following C program to Java.

```
#include <stdio.h>
int main(void) {
   printf("Hello, world!\n");
   printf("My name is George.\n");
   printf("Yada Yada Yada ...\n");
   return 0;
}
```

Applet Exercise

The following program is an example of a Java applet. This program uses several features of Java that we explain later. Note that there is no method `main()`; instead there is the method `paint()`. For now just concentrate on the body of the method `paint()`, treating the surrounding code as a template to be copied verbatim. By invoking the appropriate drawing operations on the `Graphics` object `g`, you can draw on the applet.

```
/* To place this applet in a web page, add the
   following two lines to the html document for the
   page.
  <applet code="FirstApplet.class"
  width=500 height=200></applet>
*/

// AWT and Swing together comprise the collection of
// classes used for building graphical Java programs
import java.awt.*; //required for programs that draw
import javax.swing.*; //required for Swing applets
public class FirstApplet extends JApplet {
  public void paint(Graphics g) {
    // draw a line from the upper left corner to
    // 100 pixels below the top center of the Applet
    g.drawLine(0,0,250,100);
    // draw a line from the end of the previous line
    // up to the top center of the Applet
    g.drawLine(250,100,250,0);
    // draw an oval inscribed in an invisible
    // rectangle with its upper left corner at the
    // intersection of the two lines drawn above
    g.drawOval(250,100,200,100);
  }
}
```

The class `Graphics` is used for simple drawing, and many drawing operations are defined for it. In this example we use the method `drawLine()` to draw a line. The first two numbers in parentheses for the `drawline()`

operation are the *xy* coordinates of one end of the line, and the last two numbers are the *xy* coordinates of the other end. As you can see from the output of the program, the location (0, 0) is in the upper left corner, with increasing *x* moving to the right and increasing *y* moving down. To draw an oval, give the coordinates of the upper left corner and the width and height on an invisible rectangle. The oval will be inscribed inside the rectangle.

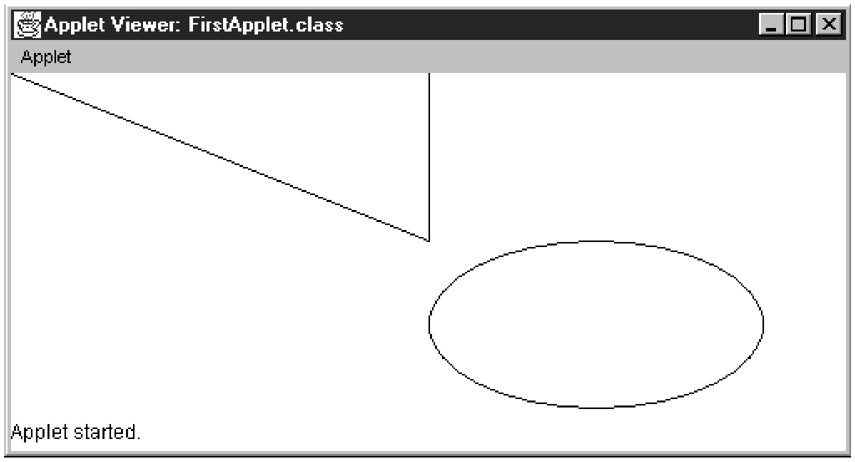
To execute an applet, first compile it like you do other Java programs. Then you can either run the program *appletviewer* or use a web browser to view the applet. To view the applet `FirstApplet` using the *applet-viewer* on Unix and Windows machines, type the following at a command line prompt.

```
appletviewer FirstApplet.java
```

Notice that we are passing `FirstApplet.java`—not `FirstApplet` or `FirstApplet.class`—to the applet-viewer. This procedure is different from running regular Java programs. In fact `appletviewer` just looks in the text file passed to it for an applet element. An applet element begins with `<applet>` and ends with `</applet>`. Any text file containing the applet element shown in the opening comment for `FirstApplet.java` would work just as well.

To view the applet in a Web browser, create a file—for example, `FirstApplet.html`. Put the applet tag in the html file. Put the html file in the same directory as your applet and then open the html file with a Web browser.

The applet looks like the following when run with an `appletviewer`.



Modify this applet to draw a simple picture. Look up the documentation for `Graphics` on the Web at

```
http://java.sun.com/j2se/1.5.0/docs/api/java/awt/Graphics.html
```

and use at least one method/operation of `Graphics` not used in `FirstApplet`.

CHAPTER 3

STATEMENTS AND CONTROL FLOW

The program examples presented until now have executed from top to bottom without making any decisions. In this chapter, we have programs select among two or more alternatives. We also demonstrate how to write programs that repeatedly execute the same sequence of instructions. Both instructions to computers and instructions in everyday life are filled with conditional and iterative statements. A conditional instruction for your microwave oven might say "*If* you wish to defrost press the defrost button; otherwise, press the full power button." An iterative instruction for baking a loaf of bread might say "Let the dough rise in a warm place *until* it has doubled in size." Conditional and iterative statements are controlled by boolean expressions. A boolean expression is either true or false. "If it is raining, wear your raincoat" is an instruction given by many parents and is followed if it is true that it is raining. In Java, expressions that evaluate as true or false are of type `boolean`. To direct the flow of control properly you need to learn how to write `boolean` expressions.

3.1 EXPRESSION, BLOCK, AND EMPTY STATEMENTS

Java has many kinds of *statements*. Most of the statements that we have shown have specified the evaluation of an expression. We'll soon look at statements that select between two alternatives and statements that repeat many times. Before doing that, we need to look more closely at the statements that we have been using. The normal flow of instructions in Java is to execute the statements of the program in sequential order from top to bottom.

All the statements used so far have been either *variable declaration statements* or *expression statements*. Variable declaration statements begin with a type, such as `int` or `String`, and end with a semicolon, as in

```
int width, height, area;
String hello = "Hello, world!";
double size = 1.5, x;
```

The first declares three variables of type `int`. The second declares one variable of type `String` and initializes it. The third declares and initializes the variable `size`, but not the variable `x.` Declaration statements start with a type and are followed by one or more variables, separated by commas. The variables may be initialized by using the equals sign followed typically by a literal. In Java all variables need to be declared.

Expression statements are formed by adding a semicolon to the end of an expression. Expressions are basic to performing computations. Not all expressions are valid in expression statements. The two types of expressions used so far that are valid in expression statements are assignment expressions and method call expressions. An *assignment expression* is any expression involving the assignment operator. A *method call expression* does not involve an assignment operator. The following are examples of expression statements.

```
area = width * height;   //simple assignment statement
System.out.println(...); //method call expression
```

A statement used for grouping a number of statements is a block. A *block* is a sequence of one or more statements enclosed by braces. A block is itself a statement. A simple example is

```
{
  x = 1;
  y = 2 * x + 1;
  System.out.println(y);
  System.out.println(x);
}
```

Statements inside a block can also be blocks. The inside block is called an *inner block*, which is *nested* in the *outer block*. An example is

```
{ //outer block
  x = 1;
  { //inner block
    y = 2;
    System.out.println(y);
  } //end of inner block
  System.out.println(x);
}
```

This example merely demonstrates the syntax of a block; we wouldn't normally put a block inside another block for no reason. Most nested blocks involve declaration statements that create local variables. A simple example of a block with declarations is

```
{
  int i = 5 + j;
  //i is created in this block, j is from elsewhere
  ...
} //end of block i disappears
```

In this example the `int` variable `i` is created when this block is executed. When this block is started, `i` is placed in memory with its initial value calculated as 5 plus the value of `j`. When the block is exited, the variable disappears.

Blocks are not terminated by semicolons. Rather they are terminated by a closing brace, also called the right brace. Recall that the semicolon, when used, is part of the statement, not something added to the statement. For example, the semicolon turns an expression into a statement. Understanding this will make it much easier for you to create syntactically correct programs with the new statement types that we introduce in this chapter.

3.1.1 EMPTY STATEMENT

The simplest statement is the *empty statement*, or *null statement*. It is just a semicolon all by itself and results in no action. A semicolon placed after a block is an empty statement and is irrelevant to the program's actions. The following code fragment produces exactly the same result as the nested block example in the preceding section. The string of semicolons simply create seven empty statements following the inner block.

```
{
  x = 1;
  {
    y = 2;
    System.out.println(y);
  };;;;;;;
  System.out.println(x);
}
```

3.2 BOOLEAN EXPRESSIONS

A *boolean expression* is any expression that evaluates to either true or false. Java includes a primitive type `boolean`. The two simplest boolean expressions are the boolean literals `true` and `false`. In addition to these two literals, boolean values result from expressions involving either relational operators for comparing numbers or logical operators that act on boolean values.

3.2.1 RELATIONAL AND EQUALITY OPERATORS

All conditional statements require some boolean expression to decide which execution path to follow. Java uses four *relational operators*: less than, `<` ; greater than, `>` ; less than or equal, `<=` ; and greater than or equal, `>=` . Java also contains two equality operators: equal, `==` ; and not equal, `!=` . They can be used between any two numeric values. The equality operators may also be used when comparing nonnumeric types. They are listed in the following table.

OPERATOR	NAME	EXAMPLE
`<`	Less than	`10 < 20` is true.
`>`	Greater than	`10 > 20` is false.
`==`	Equal	`10 == 20` is false.
`<=`	Less than or equal	`10 <= 10` is true.
`>=`	Greater than or equal	`11 >= 10` is true.
`!=`	Not equal	`10 != 20` is true.

The relational operators can be used in assignment to boolean variables, as in

```
int i = 3, j = 4;
boolean flag;
flag = 5 < 6;           //flag is now true
flag = (i == j);        //flag is now false
flag = (j + 2) <= 6;    //flag is now true
```

3.2.2 LOGICAL OPERATORS

Once you have a boolean value, either stored in a variable representing a primitive boolean value (for example, `boolean done = false;`) or as the result of an expression involving a relational operator (for example, (`x < y`)), you can combine these boolean values by using the logical operators. Java provides three logical operators, "and," "or," and "not." The meaning of these operators is given in the following table.

OPERATOR	NAME	DESCRIPTION	EXAMPLE—ASSUME x IS 10 AND y IS 20
&&	and	The expression x && y is true if both x AND y are true and false otherwise.	(x < 20) && (y < 30) is true.
||	or	The expression x || y is true if either x OR y (or both) is true and false otherwise.	(x < 20) || (y > 30) is true.
!	not	The expression !x is true if x is false and false otherwise.	!(x < 20) is false.

For example, if you wanted to determine whether a person in a database was an adult but not a senior citizen, you could check if their age was greater than or equal to 18 *and* their age was less than 65. The following Java code fragment will print out “full fare adult is true” if this condition is met; otherwise, it prints “full fare adult is false”.

```
boolean b = (ageOfPerson >= 18 && ageOfPerson < 65);
System.out.println("full fare adult is " + b);
```

For an example of the use of “or,” consider the opposite situation as above where you wanted to find out if a reduced fair was appropriate. You might write

```
b = (ageOfPerson < 18 || ageOfPerson >= 65);
System.out.println("reduced fare is " + b);
```

The logical operators && and || use *short-circuit evaluation*. In the preceding example of a logical “and” expression, if the ageOfPerson were 10, then the test for ageOfPerson < 65 would be omitted. Used partly for efficiency reasons, this approach is helpful when the second part of such an expression could lead to an undesirable result, such as program termination.

As with other operators, the relational, equality, and logical operators have rules of precedence and associativity that determine precisely how expressions involving these operators are evaluated, as shown in the following table.

OPERATOR PRECEDENCE AND ASSOCIATIVITY	
OPERATORS	ASSOCIATIVITY
() ++ (postfix) -- (postfix)	Left to right
+ (unary) - (unary) ++ (prefix) -- (prefix) !	Right to left
* / %	Left to right
+ -	Left to right
< <= > >=	Left to right
== !=	Left to right
&&	Left to right
||	Left to right
= += -= *= /= *etc.*	Right to left

Note that with the exception of the boolean unary operator negation, the relational, boolean, and equality operators have lower precedence than the arithmetic operators. Only the assignment operators have lower precedence.

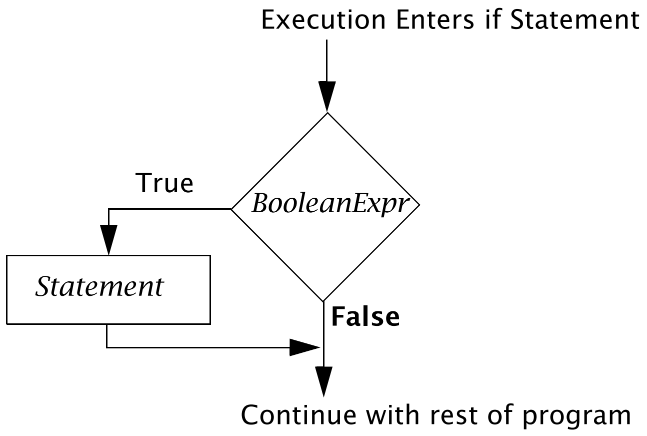
3.3 THE `if` STATEMENT

Computers make decisions by evaluating expressions and executing different statements based on the value of the expression. The simplest type of decision is one that can have only two possible outcomes, such as go left versus go right or continue versus stop. In Java, we use *boolean expressions* to control decisions that have two possible outcomes.

The `if` *statement* is a conditional statement. An `if` statement has the general form

```
if ( BooleanExpr )
  Statement
```

If the expression *BooleanExpr* is true, then the statement, *Statement,* is executed; otherwise, *Statement* is skipped. *Statement* is called the *then statement.* In some programming languages, but not Java, `then` is used to signal the then statement. After the `if` statement has been executed, control passes to the next statement. The flow of execution can skip around *Statement* as shown in the following diagram.



Note the absence of semicolons in the general form of the `if` statement. Recall that the semicolon, when required, is part of the *Statement* and is not used to separate statements, as in

```
if (temperature < 32)
  System.out.println("Warning: Below freezing!");
System.out.println("It's " + temperature + " degrees");
```

The message `Warning: Below freezing!` is printed only when the `temperature` is less than 32. The second print statement is always executed. This example has a semicolon at the end of the `if` statement because the statement inside the `if` statement is an expression statement that ends with a semicolon.

When the *Statement* inside an `if` statement is a block, you get `if` statements that look like

```
if (temperature < 32)
{
  System.out.println("Warning Warning Warning!");
  System.out.println("Warning: Below freezing!");
  System.out.println("Warning Warning Warning!");
}
```

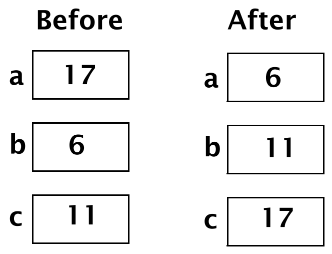
Here you can see the importance of the block as a means of grouping statements. In this example, what otherwise would be three separate statements are grouped, and all are executed when the boolean expression is true. The formatting shown of the `if` statement with a block as the *Statement* aligns vertically with the braces of the block statement. An alternative formatting—and the one that we use—places the opening brace on the same line as the keyword `if` and then aligns the closing brace with the keyword as shown here.

```
if (temperature < 32) {
  System.out.println("Warning Warning Warning!");
  System.out.println("Warning: Below freezing!");
  System.out.println("Warning Warning Warning!");
}
```

At the end of this chapter, we discuss further which style to choose.

3.3.1 PROBLEM SOLVING WITH THE `if` STATEMENT

A different number is initially placed in each of three boxes, labeled *a*, *b*, and *c*, respectively. The problem is to rearrange or sort the numbers so that the final number in box *a* is less than that in box *b* and that the number in box *b* is less than that in box *c*. Initial and final states for a particular set of numbers are as follows.



Pseudocode for performing this sorting task involves the following steps.

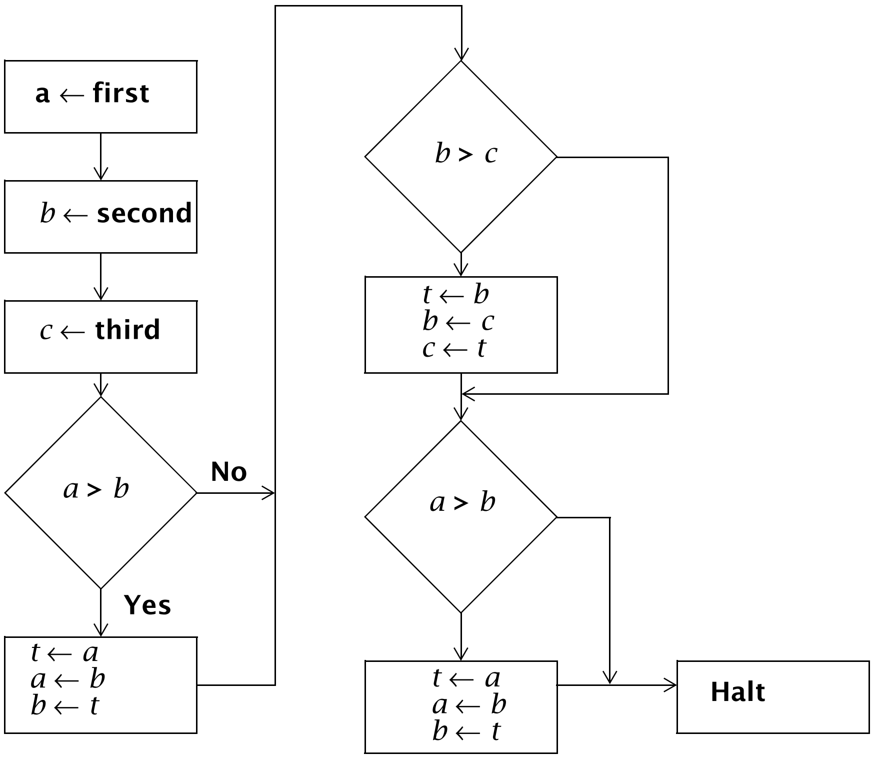
PSEUDOCODE FOR THREE-NUMBER SORT

```
1.  Place the first number in box a.
2.  Place the second number in box b.
3.  Place the third number in box c.
4.  If the number in a is not larger than the number in b, go to step 6.
5.  Interchange the number in a with that in b.
6.  If the number in b is larger than the number in c, then go to step 7;
    otherwise, halt.
7.  Interchange the numbers in b and c.
8.  If the number in a is larger than that in b, then go to step 9; otherwise,
    halt.
9.  Interchange the numbers in a and b.
10. Halt.
```

Let's execute this pseudocode with the three specific numbers previously given: 17, 6, and 11, in that order. We always start with the first instruction. The contents of the three boxes at various stages of execution are shown in the following table.

Box	Step 1	Step 2	Step 3	Step 5	Step 7
a	17	17	17	6	6
b		6	6	17	11
c			11	11	17

To execute step 1, we place the first number, 17, in box *a*; similarly, at the end of instruction 3, the 6 has been inserted into box *b*, and box *c* contains the 11. As 17 is larger than 6, the condition tested in step 4 is false, and we proceed to instruction 5; this step switches the values into boxes *a* and *b* so that box *a* now contains the 6 and box *b* has the 17. Step 6 has now been reached, and we compare the number in box *b* (17) to that in box *c* (11); 17 is greater than 11, so a transfer is made to step 7. The numbers in boxes *b* and *c* are then interchanged so that box *b* has the 11 and box *c* has the 17. The test in step 8 fails (6 is not larger than 11) and the computation then halts. The three numbers have been sorted in ascending sequence (i.e., 6 < 11 < 17). You should convince yourself by bench testing this algorithm with other values of *a, b*, and *c* that the computation described by the pseudocode will work correctly for any three numbers. A flowchart of the sorting algorithm is shown in the following diagram.



Note that we decomposed the operation of interchanging two numbers into three more primitive instructions. Box *t* is used as temporary storage to hold intermediate results. In order to interchange or switch the two numbers *a* and *b*, we first temporarily store one of the numbers, say, *a*, in *t* ($t \leftarrow a$); next the other number is stored in *a* ($a \leftarrow b$), and, last, the first number is placed in *b* ($b \leftarrow t$). Note that the instruction sequence "$a \leftarrow b$; $b \leftarrow a$" will not interchange *a* and *b* because the first instruction effectively destroys the old value in *a*. In computer terms, the labeled boxes are analogous to memory or storage areas that can contain values.

Next we code in Java the pseudocode version of our sorting program.

```
// SortInput.java - sort three numbers
import java.util.*;

class SortInput {
  public static void main (String[] args) {
    int a, b, c, t;
    Scanner scan = new Scanner(System.in);
    System.out.println("type three integers:");
    a = scan.nextInt();
    b = scan.nextInt();
    c = scan.nextInt();
    if (a > b) {
      t = a;
      a = b;
      b = t;
    }
    if (b > c) {
      t = b;
      b = c;
      c = t;
    }
    if (a > b) {
      t = a;
      a = b;
      b = t;
    }
    System.out.print("The sorted order is : ");
    System.out.println(a + ", " + b + ", " + c);
  }
}
```

Dissection of the *SortInput* Program

● `int a, b, c, t;`

This program declares four integer variables. The variables `a`, `b`, and `c` are inputs to be sorted, and `t` is to be used for temporary purposes, as described in the pseudocode.

● `System.out.println("type three integers:");`

This line is used to *prompt* the user to type the three numbers to be sorted. Whenever a program is expecting the user to do something, it should print out a prompt telling the user what to do.

●
```
a = scan.nextInt();
b = scan.nextInt();
c = scan.nextInt();
```

The method call expression `scan.nextInt()` is used to obtain the input from the keyboard. Three separate integers need to be typed. The values read will be stored in the three variables.

●
```
if (a > b) {
  t = a;
  a = b;
  b = t;
}
if (b > c) {
  t = b;
  b = c;
  c = t;
}
if (a > b) {
  t = a;
  a = b;
  b = t;
}
```

The `if` statements and resulting assignments are Java notation for the same actions described in the sort flow chart. To comprehend these actions you need to understand why the interchange or swapping of values between two variables, such as `a` and `b`, requires the use of the temporary `t`. Also note how the three assignments are grouped as a block, allowing each `if` expression to control a group of actions.

```
● System.out.print("The sorted order is : ");
  System.out.println(a + ", " + b + ", " + c);
}
```

If the input values were 10, 5, and 15, the output would be

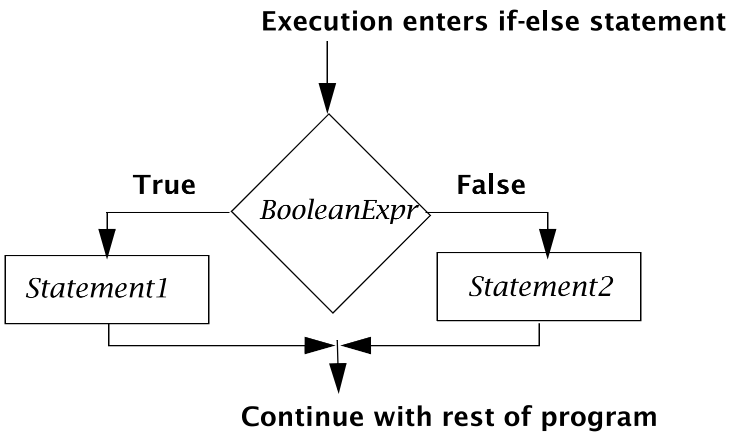
```
The sorted order is : 5, 10, 15
```

3.4 THE if-else STATEMENT

Closely related to the `if` statement is the `if-else` *statement*. An `if-else` statement has the following general form:

`if (` *BooleanExpr* `)` *Statement1* `else` *Statement2*

If the expression *BooleanExpr* is true, then *Statement1* is executed and *Statement2* is skipped; if *BooleanExpr* is false, then *Statement1* is skipped and *Statement2* is executed. *Statement2* is called the `else` statement. After the `if-else` statement has been executed, control passes to the next statement. The flow of execution branches and then rejoins, as shown in the following diagram.


Execution enters if-else statement
True
BooleanExpr
False
Statement1
Statement2
Continue with rest of program

Consider the following code:

```
if (x < y)
  min = x;
else
  min = y;
System.out.println("min = " + min);
```

If `x < y` is true, then `min` will be assigned the value of `x`; if it is false, then `min` will be assigned the value of `y`. After the `if-else` statement is executed, `min` is printed.

As with the `if` statement, either branch of an `if-else` statement can contain a block, as shown in the following example.

```
if (temperature < 32) {
  System.out.println("Warning Warning Warning!");
  System.out.println(32 - temperature + " degrees below freezing!");
  System.out.println("Warning Warning Warning!");
}
else {
  System.out.println("It’s " + temperature + "degrees fahrenheit.");
}
```

Common Programming Error

A semicolon is used to change an expression to an expression statement. All statements do not end in semicolons, and extraneous semicolons can cause subtle errors. Look carefully at the following code fragment. What is printed when it executes if *x* is 3 and *y* is 4?

```
if (x < y);
  System.out.println("The smaller is " + x);
if (y < x);
  System.out.println("The smaller is " + y);
```

The answer is

```
    The smaller is 3
    The smaller is 4
```

Note the extra semicolon after the close parenthesis in each `if` statement. The indentation is misleading: All four lines should be indented the same to reflect what is actually going to happen. The true branch of each `if` statement is the empty statement “`;`”. The example code is syntactically correct but semantically incorrect.

Comon Programming Error

Another common error is to be misled by indentation and to try to include two statements in one branch without using a block. Consider this example.

```
if (temperature < 32)
    System.out.print("It is now");
    System.out.print(32 - temperature);
    System.out.println(" below freezing.");
System.out.println("It's " + temperature + "degrees");
```

A user might mistakenly think that, if `temperature` is 32 or above, the code will execute only the last print statement displaying `temperature`. That is almost certainly what was intended. Unfortunately, if `temperature` is 32 or above, only the first `print()` would be skipped.

If `temperature` is 45 the following confusing message would be printed:

```
    -13 below freezing.
    It's 45 degrees
```

To get the behavior suggested by the indentation shown, a block must be used, as in

```
if (temperature < 32) {
    System.out.print("It is now");
    System.out.print(32 - temperature);
    System.out.println(" below freezing.");
}
System.out.println("It's " + temperature + "degrees");
```

3.4.1 NESTED `if-else` STATEMENTS

The `if` statement is a full-fledged statement and can be used anywhere a statement is expected, so you could put another `if` statement in either branch of an `if` statement. For example, you could rewrite the earlier air fare checking statement as

```
if (ageOfPerson >= 18)
  if (ageOfPerson < 65)
    System.out.println("full fare adult");
```

If the expression `(ageOfPerson >= 18)` is true, then the statement

```
  if (ageOfPerson < 65)
    System.out.println("full fare adult");
```

is evaluated. The true branch for the `ageOfPerson >= 18` `if` statement is itself an `if` statement. In general, if the true branch of an `if` statement contains only another `if` statement, writing it as a single `if` statement is more efficient, combining the expressions with the operator `&&`, as we did in Section 3.2.2, *Logical operators*, on page 49. Using `&&` in most cases is also clearer.

These nested `if` statements can't always be collapsed. Consider the following variation on our earlier example involving temperature.

```
if (temperature < 32) {
  System.out.println("Warning Warning Warning!");
  if (temperature < 0)
    System.out.println((-temperature) + " degrees below Zero!"
  else
    System.out.println(32 - temperature + "(F) below Freezing!");
  System.out.println("Warning Warning Warning!");
}
else {
  System.out.println("It is " + temperature + " degrees fahrenheit.");
}
```

The `then` statement for the outer `if-else` statement is a block containing another `if-else` statement.

Common Programming Error

In algebra class you came across expressions such as $18 \leq age < 65$. This expression tempts many new programmers to write

```
if (18 <= age < 65) ...
```

In evaluating the expression, Java first evaluates `(18 <= age)`—let's call it `part1`—which is either true or false. It then tries to evaluate `(part1 < 65)`, which is an error because relational operators are used to compare two numbers, not a boolean and a number. The only good thing about this type of mistake is that the compiler will catch it. Our example earlier for printing `full fare adult` showed the proper way to handle this situation.

3.4.2 `if-else-if-else-if ...`

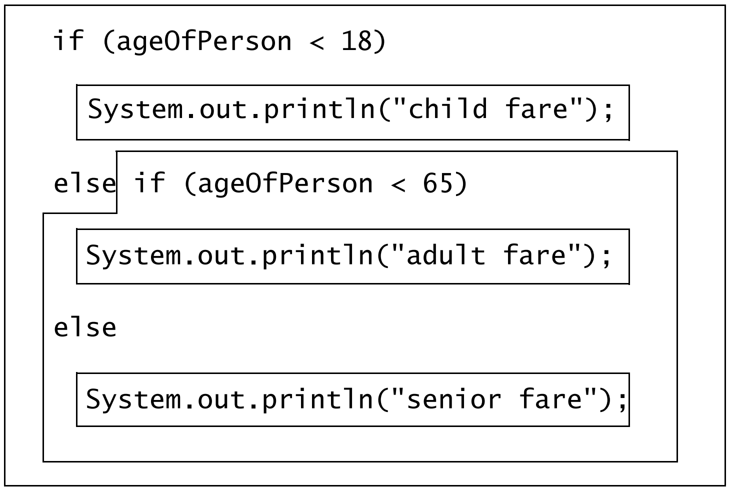
In the preceding example, the nesting was all done in the `then` statement part of the `if-else` statement. Now we nest in the `else` statement part of the `if-else` statement:

```
if (ageOfPerson < 18)
  System.out.println("child fare");
else {
  if (ageOfPerson < 65)
    System.out.println("adult fare");
  else
    System.out.println("senior fare");
}
```

The braces are not needed; we added them only for clarity. This form is so common that experienced programmers usually drop the braces. In addition, the `else` and the following `if` are usually placed on the same line, as in

```
if (ageOfPerson < 18)
  System.out.println("child fare");
else if (ageOfPerson < 65)
  System.out.println("adult fare");
else
  System.out.println("senior fare");
```

Note that the second `if` statement is a *single* statement that constitutes the else branch of the first statement. The two forms presented are equivalent. You should be sure that you understand why they are. If you need to, look back at the general form of the `if` statement and recall that an entire `if-else` statement is a statement itself. This fact is illustrated in the following figure, wherein each statement is surrounded by a box. As you can see, there are five different statements.



```
if (ageOfPerson < 18)
  System.out.println("child fare");
else if (ageOfPerson < 65)
  System.out.println("adult fare");
else
  System.out.println("senior fare");
```

Sometimes this chain of `if-else-if-else-if-else`... can get rather long, tedious, and inefficient. For this reason, a special construct can be used to deal with this situation when the condition being tested is of the right form. If you want to do different things based on distinct values of a *single* integer expression, then you can use the `switch` statement, which we discuss in Section 3.9, *The switch statement*, on page 69.

3.4.3 THE DANGLING `else` PROBLEM

When you use sequences of nested `if-else` statements, a potential problem can arise as to what `if` an `else` goes with, as for example in

```
if (Expression1)
  if (Expression2)
    Statement1
else
  Statement2
```

The indenting suggests that *Statement2* is executed whenever *Expression1* is false; however, that isn't the case. Rather, *Statement2* is executed only when *Expression1* is true **and** *Expression2* is false. The proper indenting is

```
if (Expression1)
  if (Expression2)
    Statement1
  else
    Statement2
```

The rule used in Java is that an `else` is always matched with the nearest preceding `if` that doesn't have an `else`. To cause *Statement2* to be executed whenever *Expression1* is false, as suggested by the first indenting example, you can use braces to group the statements as shown.

```
if (Expression1) {
  if (Expression2)
    Statement1
}
else
  Statement2
```

The braces are like parentheses in arithmetic expressions that are used to override the normal precedence rules. The nested, or inner, `if` statement is now inside a block that prevents it from being matched with the `else`.

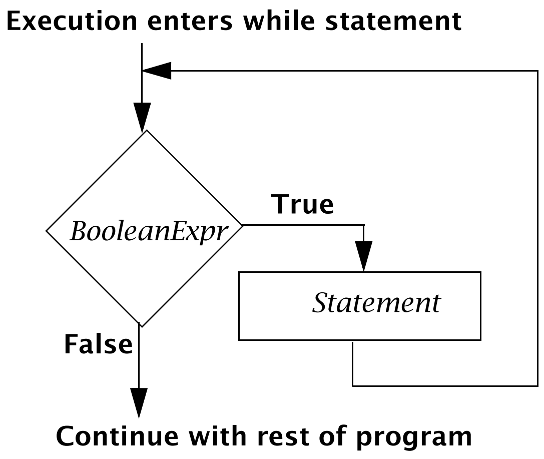
3.5 THE while STATEMENT

We have added the ability to choose among alternatives, but our programs still execute each instruction at most once and progress from the top to the bottom. The *while statement* allows us to write programs that run repeatedly.

The general form of a `while` statement in Java is

```
while ( BooleanExpr )
  Statement
```

Statement is executed repeatedly as long as the expression *BooleanExpr* is true, as shown in the following flowchart.


Execution enters while statement
BooleanExpr
True
Statement
False
Continue with rest of program

This flowchart is the same as the one for the `if` statement, with the addition of an arrow returning to the test box.

Note that *Statement* may not be executed. That will occur when *BooleanExpr* evaluates to false the first time. Note also, that like the description of the `if-else` statement, there are no semicolons in the general form of the `while` statement. The semicolon, if any, would be part of the statement that follows the parenthesized boolean expression.

A simple example for use on Valentine's Day is

```
// Valentine.java - a simple while loop
class Valentine {
  public static void main(String[] args) {
    int howMuch = 0;
    while (howMuch++ < 5)
      System.out.println("I love you.");
  }
}
```

The output is

```
I love you.
I love you.
I love you.
I love you.
I love you.
```

When the body of the loop is a block you get

```
while ( BooleanExpr ) {
  Statement1
  Statement2
  ...
}
```

Modifying our earlier example gives

```
int howMuch = 0;
while (howMuch++ < 5) {
  System.out.print("I love you.");
  System.out.println {" @----->- my rose.");
}
```

which results in the output

```
I love you. @----->- my rose.
I love you. @----->- my rose.
I love you. @----->- my rose.
I love you. @----->- my rose.
I love you. @----->- my rose.
```

Most `while` statements are preceded by some initialization statements. Our example initialized the loop counting variable `howMuch`.

3.5.1 PROBLEM SOLVING WITH LOOPS

Suppose that you wanted to have a program that could read in an arbitrary number of nonzero values and compute the average. If you were to use pseudocode, the program might look like the following.

PSEUDOCODE FOR AVERAGE, USING `goto`

```
1. Get a number.
2. If the number is 0 go to step 6.
3. Add the number to the running total.
4. Increment the count of numbers read in.
5. Go back to step 1.
6. Divide the running total by the count of numbers in order to get the average.
7. Print the average.
```

In this pseudocode, step 5 goes back to the beginning of the instruction sequence, forming a loop. In the 1950s and 1960s many programming languages used a *goto statement* to implement step 5 for coding such a loop. These `goto` statements resulted in programs that were hard to follow because they could jump all over the place. Their use was sometimes called *spaghetti code*. In Java there is no `goto` statement. Java includes `goto` as a keyword that has no use so that the compiler can issue an error message if it is used inadvertently by a programmer familiar with C or C++. Today *structured control constructs* can do all the good things but none of the bad things that you could do with `goto` statements.

A *loop* is a sequence of statements that are to be repeated, possibly many times. The preceding pseudocode has a *loop* that begins at step 1 and ends at step 5. Modern programs require that you use a construct to indicate explicitly the beginning and the end of the loop. The structured equivalent, still in pseudocode, might look like the following.

Pseudocode with Loop

```
Pseudocode for average, without using goto
get a number
while the number is not 0 do the following:
    add the number to the running total
    increment the count of numbers read in
    get a number
(when the loop exits)
divide the running total by the count of numbers to get the average
print the average
```

The loop initialization in this case is reading in the first number. Somewhere within a loop, typically at the end, is something that prepares the next iteration of the loop—getting a new number in our example. So, although not part of the required syntax, `while` statements generally look like the following

```
Statement_init        // initialization for the loop
while ( BooleanExpr ) {
  Statement1
  Statement2
  ...
  Statement_next // prepare for next iteration
}
```

The `while` statement is most often used when the number of iterations isn't known in advance. This situation might occur if the program was supposed to read in values, processing them in some way until a special value called a *sentinel* was read in. Such is the case for our "compute the average" program: It reads in numbers until the sentinel value 0 is read in.

We can now directly translate into Java the pseudocode for computing an average.

```
// Average.java - compute average of input values
import java.util.*;
public class Average {
  public static void main(String[] args) {
    double number;
    int count = 0;
    double runningTotal = 0;
    Scanner scan = new Scanner(System.in);
    // initialization before first loop iteration
    System.out.println("Type some numbers, the last one being 0");
    number = scan.nextDouble();
    while (number != 0) {
      runningTotal = runningTotal + number;
      count = count + 1;
      // prepare for next iteration
      number = scan.nextDouble();
    }
    System.out.print("The average of the ");
    System.out.print(count);
    System.out.print(" numbers is ");
    System.out.println(runningTotal / count);
  }
}
```

Dissection of the *Average* Program

●
```
double number;
int count = 0;
double runningTotal = 0;
```

The variable `number` is used to hold the floating point value typed in by the user. No initial value is given in the declaration because it gets its initial value from user input. The variable `count` is used to count the number of nonzero values read in. Counting is done efficiently and accurately with whole numbers, so the variable `count` is an `int` initialized to zero. The variable `runningTotal` is used to accumulate the sum of the numbers read in so far and is initialized to zero. Similar to assignment statements, these are declaration statements with initial values. Later we show when declaration statements can be used but assignment statements can't.

●
```
System.out.println("Type some numbers, the last one being 0");
number = scan.nextDouble();
```

A message should always be printed to prompt the user when the user is expected to enter data. We could have initialized `number` when it was declared, but it is central to the loop so we initialized it just before entering the loop. These two statements correspond to the first line of our pseudocode. All the preceding code is supporting syntax required by Java.

●
```
while (number != 0) {
```

The loop will continue *while* the value stored in the variable `number` is not 0. The value 0 is used to detect the end of the loop. It acts as a *sentinel* to keep the loop from running forever. A *sentinel* is an important technique for terminating loops correctly.

●
```
  runningTotal = runningTotal + number;
  count = count + 1;
  number = scan.nextDouble();
}
```

The first statement in the loop body would look strange in a mathematics class unless `number` was 0. Recall that the symbol `=` is not an equals sign; it is the assignment operator. It means: Evaluate the expression on the right and then assign that value to the variable on the left. The result in this case is to add the value of `number` to the value of `runningTotal` and store the result in `runningTotal`. This type of expression is common in computer programs: It computes a new value for a variable, using an expression that includes the old value of the variable. Similarly, we increment the value of `count`. The last statement in the body of the loop gets ready for the next iteration by reading in a new value from the user and storing it in `number.` This action overwrites the old value, which is no longer needed. At this point the computer will go back and evaluate the boolean expression to determine whether the loop should execute again (i.e., `number` isn't 0) or continue with the rest of the program.

●
```
System.out.println(runningTotal / count);
```

When the user finally types a 0, the program prints the answer and exits. We have already used the `print()` and `println()` methods with a string and with a number. Here we show that the number can be an expression. Recall that the symbol `/` is the symbol for division. If the first number typed is 0, this program will terminate by printing

```
● The average of the 0 numbers is NaN
```

The symbol NaN stands for *not a number* and results from dividing zero by zero. A better solution might be to test, using an `if-else` statement, for this special case and print a more appropriate message. We leave that for you to do as an exercise.

Common Programming Error

Every programmer has probably forgotten the "prepare for next iteration" line at least once, causing a program to loop forever. When coding a `while` statement, be careful to verify that some variable changes each time around the loop. Also check to be sure that the loop condition will eventually be reached. How many times will the following loop execute?

```
int count = 13;
System.out.println("The multiples of 13 between 1 and 100 are:");
while (count != 100){
  System.out.println(count);
  count = count + 13;
}
```

The loop will run forever because `count` will never equal 100. The problem is obvious in this small example, and such problems do occur—although they're not always so apparent. A test for less than or greater than is always preferable to a test of strict equality or inequality.

3.6 THE DO STATEMENT

A variant of the `while` statement is the `do` statement. The general form of a `do` statement in Java is

```
do
  Statement
while ( BooleanExpr )
```

Here the test for exiting is syntactically and semantically made at the end of executing each iteration of *Statement*. This means that *Statement* is executed at least once. Execution stops when the expression *BooleanExpr* is false, An example using this construct is a loop for computing the greatest common divisor for two integers using Euclid's algorithm.

```
import java.util.*;

class GreatestCommonDivisor{
  public static void main(String[] args)
  {
     Scanner scan = new Scanner(System.in);
     System.out.println("Enter 2 integers.");
     int num1 = scan.nextInt();
     int num2 = scan.nextInt();
     int  m = num1, n = num2, remainder;
     do {
        remainder = m % n;
        m = n;
        n = remainder;
     } while( n != 0);
     System.out.println("GCD (" + num1 + ", " + num2 +
                        ") = " + m );
  }
}
```

You should carry out this computation by hand to make sure you understand the do-statement and its use in this oldest of algorithms.

3.7 THE FOR STATEMENT

You can use the `while` construct to create any loop you will ever need. However, Java, like most modern programming languages, provides two alternative ways to write loops that are sometimes more convenient. The `do` statement discussed in the previous section is one alternative. The other alternative is the `for` *statement*. The `for` *statement* is a looping statement that captures the initialization, termination test, and iteration preparation all in one place at the top of the loop. The general form of the `for` statement is

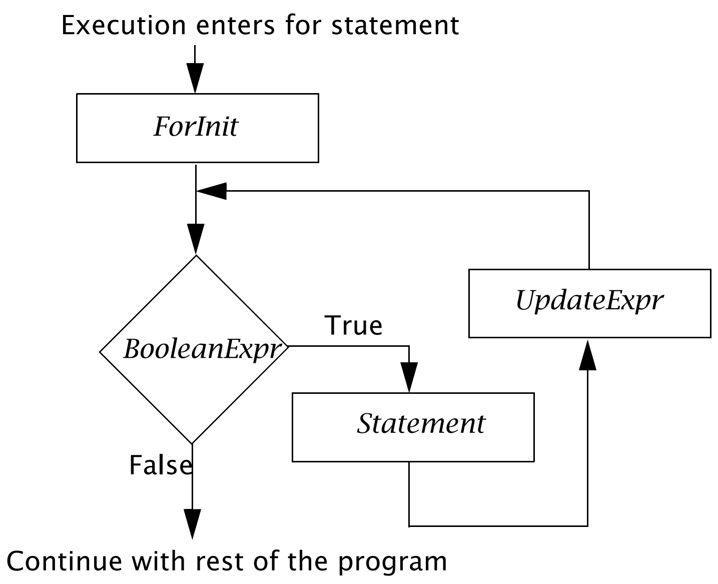
```
for ( ForInit; BooleanExpr ; UpdateExpr )
    Statement
```

As part of the syntax of the `for` statement, the semicolons are used to separate the three expressions. Note the absence of a semicolon between *UpdateExpr* and the closing parenthesis.

The following steps occur in the execution of a `for` statement.

1. *ForInit* is evaluated *once*—before the body of the loop.
2. *BooleanExpr* is tested *before each iteration* and, if true, the loop continues (as in the `while` statement).
3. The loop body *Statement* is executed.
4. *UpdateExpr* is evaluated at the *end of each iteration.*
5. Go to step 2.

The following diagram also shows the flow of execution.


Execution enters for statement
ForInit
BooleanExpr
True
False
UpdateExpr
Statement
Continue with rest of the program

Compare this general form and the flow with that of the `while` statement. In the `for` statement, the initialization expression represented by *ForInit* and the iteration expression represented by *UpdateExpr* frequently involve the use of an assignment operator, but one is not required.

The `for` statement applies most naturally to situations involving going around a loop a specific number of times. For example, if you wanted to print out the square roots of the numbers 1 through 10 you could do so as follows.

```
// SquareRoots.java - print square roots of 1 - 10
import static java.lang.Math.*;
public class SquareRoots {
  public static void main(String[] args) {
    int i;
    double squareRoot;
    for (i = 1; i <= 10; i++) {
      squareRoot = sqrt(i);
      System.out.println("the square root of " + i + " is " + squareRoot);
    }
    System.out.println("That's All!");
  }
}
```

Dissection of the *SquareRoots* Program

- `import static java.lang.Math.*;`

A `import static` can be used to import the static fields and methods from a class for use without qualification in the local context. Here it is used to allow us to call `java.lang.Math.sqrt()` by simply writing `sqrt()`.

- `for (i = 1; i <= 10; i++) {`

The expression `i = 1` is evaluated only once—before the loop body is entered. The expression `i <= 10` is evaluated before *each* execution of the loop body. The expression `i++` is evaluated at the *end of each* loop body evaluation. Recall that `i++` is shorthand for `i = i + 1`.

```
●   squareRoot = sqrt(i);
    System.out.println("the square root of " + i + " is " + squareRoot);
  }
```

The body of the loop is one statement—in this case a block. The braces are part of the block syntax, not part of the syntax of a `for` statement.

```
● System.out.println("That's All!");
```

When the loop exits, program execution continues with this statement.

We can redo the square root printing loop by using a `while` statement as follows.

```
// SquareRoots2.java - replace for with while
public class SquareRoots2 {
  public static void main(String[] args) {
    int i;
    double squareRoot;
    i = 1;               // initialization-expr
    while (i <= 10) {
      squareRoot = Math.sqrt(i);
      System.out.println("the square root of " + i + " is " + squareRoot);
      i++;               // iteration-expr
    }
    System.out.println("That's All!");
  }
}
```

In both versions of this program, the same sequence of steps occurs.

1. The variable `i` is initialized to 1.
2. The boolean expression `i <= 10` is tested, and if true, the loop body is executed. If the expression is false, the loop statement is completed and execution continues with the next statement in the program, the final print statement.
3. After executing the print statement within the loop, the statement incrementing the variable `i` is executed, in preparation for the next loop iteration.
4. Execution continues with step 2.

3.7.1 LOCAL VARIABLES IN THE `for` STATEMENT

The *ForInit* part of the `for` statement can be a local declaration. For example, let's rewrite the above for-loop.

```
    for (int i = 1; i <= 10; i++) {
      squareRoot = sqrt(i);
      System.out.println("the square root of " + i + " is " + squareRoot);
    }
```

In this case, the `int` variable `i` declared in the `for` statement is limited to the `for` statement, including the body of the loop. The variable's existence is tied to the `for` statement code; outside the `for` statement it disappears. If you want the variable `i` to exist independently of the `for` statement you must declare it before the `for` statement. It then is available throughout the block containing the `for` statement. The variable is said to have *local scope*. We discuss scope further in Section 4.4, *Scope of Variables*, on page 86.

3.8 THE break AND continue STATEMENTS

Two special statements,

```
break;   and   continue;
```

interrupt the normal flow of control. The `break` statement causes an exit from the innermost enclosing loop. The break statement also causes a `switch` statement to terminate. (We discuss the `switch` statement in Section 3.9, *The switch statement*, on page 69.) In the following example, a test for a negative argument is made. If the test is true, then a `break` statement is used to pass control to the statement immediately following the loop.

```
while (true) {      //seemingly an infinite loop
  System("Enter a positive integer:")
  n = scan.nextInt();
  if (n < 0)
    break;          // exit loop if n is negative
  System.out.print("squareroot of " + n);
  System.out.println(" = " + Math.sqrt(n));
}
// break jumps here
```

This use of the `break` statement is typical. What would otherwise be an infinite loop is made to terminate upon a given condition tested by the `if` expression.

The `continue` statement causes the current iteration of a loop to stop and causes the next iteration of the loop to begin immediately. The following code adds `continue` to the preceding program fragment in order to skip processing of negative values.

```
// BreakContinue.java - example of break and continue
import java.util.*;
class BreakContinue {
  public static void main(String[] args) {
    int n;
    Scanner scan = new Scanner(System.in);
    while (true) {      //seemingly an infinite loop
      System.out.print("Enter a positive integer ");
      System.out.print("or 0 to exit:");
      n = scan.nextInt();
      if (n == 0)
        break;          // exit loop if n is 0
      if (n < 0)
        continue;       //wrong value
      System.out.print("squareroot of " + n);
      System.out.println(" = " + Math.sqrt(n));
      //continue lands here at end of current iteration
    }
    //break lands here
    System.out.println("a zero was entered");
  }
}
```

The output of this program, assuming that the user enters the values 4, 21, 9, and 0, is

```
os-prompt>java BreakContinue
Enter a positive integer or 0 to exit:4
squareroot of 4 = 2.0
Enter a positive integer or 0 to exit:-1
Enter a positive integer or 0 to exit:9
squareroot of 9 = 3.0
Enter a positive integer or 0 to exit:0
a zero was entered
os-prompt>
```

The `continue` statement may only occur inside `for`, `while`, and `do` loops. As the example shows, `continue` transfers control to the end of the current iteration, whereas `break` terminates the loop.

The `break` and `continue` statements can be viewed as a restricted form of a `goto` statement. The `break` is used to go to the statement following the loop and the `continue` is used to go to the end of the current iteration. For this reason, many programmers think that `break` and `continue` should be avoided. In fact, many uses of `break` and `continue` can be eliminated by using the other structured control constructs. For example, we can redo the `BreakContinue` example, but without `break` and `continue`, as follows.

```
// NoBreakContinue.java - avoiding break and continue
import java.util.*;
class NoBreakContinue {
  public static void main(String[] args) {
    int n;
    Scanner scan = new Scanner(System.in);
    System.out.print("Enter a positive integer ");
    System.out.print("or 0 to exit:");
    n = scan.nextInt();
    while (n != 0) {
      if (n > 0) {
        System.out.print("squareroot of " + n);
        System.out.println(" = " + Math.sqrt(n));
      }
      System.out.print("Enter a positive integer ");
      System.out.print("or 0 to exit:");
      n = scan.nextInt();
    }

    System.out.println("a zero was entered");
  }
}
```

The loop termination condition is now explicitly stated in the `while` statement and not buried somewhere inside the loop. However, we did have to repeat the prompt and the input statement—once before the loop and once inside the loop. We eliminated `continue` by changing the `if` statement to test for when the square root could be computed instead of testing for when it could not be computed.

3.9 THE `switch` STATEMENT

The `switch` statement can be used in place of a long chain of `if-else-if-else-if-else` statements when the condition being tested evaluates to an integer numeric type. Suppose that we have an integer variable `dayOfWeek` that is supposed to have a value of 1 through 7 to represent the current day of the week. We could then print out the day as

```
if (dayOfWeek == 1)
  System.out.println("Sunday");
else if (dayOfWeek == 2)
  System.out.println("Monday");
else if (dayOfWeek == 3)
  System.out.println("Tuesday");
else if (dayOfWeek == 4)
  System.out.println("Wednesday");
else if (dayOfWeek == 5)
  System.out.println("Thursday");
else if (dayOfWeek == 6)
  System.out.println("Friday");
else if (dayOfWeek == 7)
  System.out.println("Saturday");
else
  System.out.println("Not a day number " + dayOfWeek);
```

An alternative is to use a `switch` statement as follows.

```
switch (dayOfWeek) {
  case 1:
    System.out.println("Sunday");
    break;
  case 2:
    System.out.println("Monday");
    break;
  case 3:
    System.out.println("Tuesday");
    break;
  case 4:
    System.out.println("Wednesday");
    break;
  case 5:
    System.out.println("Thursday");
    break;
  case 6:
    System.out.println("Friday");
    break;
  case 7:
    System.out.println("Saturday");
    break;
  default:
    System.out.println("Not a day number " + dayOfWeek);
}
```

Unlike the `if-else` and looping statements described earlier in this chapter, the braces are part of the syntax of the `switch` statement. The controlling expression in parentheses following the keyword `switch` must be of integral type. Here it is the `int` variable `dayOfWeek`. After the expression has been evaluated, control jumps to the appropriate `case` label. All the constant integral expressions following the `case` labels must be unique. Typically, the last statement before the next `case` or `default` label is a `break` statement. If there is no `break` statement, then execution “falls through” to the next statement in the succeeding case. Missing `break` statements are a frequent cause of error in `switch` statements. For example, if the `break;` was left out of just `case 1` and if `dayOfWeek` was 1, the output would be

```
Sunday
Monday
```

instead of just

```
Sunday
```

There may be at most one default label in a switch statement. Typically, it occurs last, although it can occur anywhere. The keywords case and default can't occur outside a switch.

Taking advantage of the behavior when break; is not used, you can combine several cases as shown in the following example.

```
switch (dayOfWeek) {
  case 1:
  case 7:
    System.out.println("Stay home today!");
    break;
  case 2:
  case 3:
  case 4:
  case 5:
  case 6:
    System.out.println("Go to work.");
    break;
  default:
    println("Not a day number " + dayOfWeek);
    break;
}
```

Note that the break statement—not anything related to the case labels—causes execution to continue after the switch statement. The switch statement is a structured goto statement. It allows you to go to one of several labeled statements. It is then combined with the break statement, which does another goto, in this case "go to the statement after the switch."

THE EFFECT OF A switch STATEMENT

1. Evaluate the switch expression.
2. Go to the case label having a constant value that matches the value of the expression found in step 1; or, if a match is not found, go to the default label; or, if there is no default label, terminate switch.
3. Continue executing statements in order until the end of switch is reached or a break is encountered.
4. Terminate switch when a break statement is encountered, or terminate switch by "falling off the end."

3.10 USING THE LAWS OF BOOLEAN ALGEBRA

Boolean expressions can frequently be simplified or more efficiently evaluated by converting them to logically equivalent expressions. Several rules of Boolean algebra are useful for rewriting boolean expressions.

1. Commutative law: *a or b == b or a* and a *and b == b and a.*
2. Distributive *and* law: *a and (b or c) == (a and b) or (a and c).*
3. Distributive *or* law: *a or (b and c) == (a or b) and (a or c).*
4. Double negation: *!!a == a.*
5. DeMorgan's laws: *!(a or b) == (!a and !b)* and *!(a and b) == !a or !b.*

In the expression `x || y`, `y` will not be evaluated if `x` is true. The value of `y` in that case doesn't matter. For this reason, it may be more efficient to have the first argument of a boolean *or* expression be the one that most often evaluates to true. This is the basis for short-circuit evaluation of the logical *or* operator.

```
while (a < b || a == 200) {
  ...
}
while (a == 200 || a < b) {
  ...
}
```

For the two controlling expressions, which order is more likely to be efficient? Note that by the commutative law both `while` conditions are equivalent.

3.11 PROGRAMMING STYLE

We follow Java professional style throughout the programming examples. Statements should readily display the flow of control of the program and be easy to read and follow. Only one statement should appear on a line. Indentation should be used consistently to set off the flow of control. After a left brace, the next line should be indented, showing that it is part of that compound statement. There are two conventional brace styles in Java. The style we follow in this book is derived from the C and C++ professional programming community. In this style, an opening or left brace stays on the same line as the beginning of a selection or iteration statement. The closing or right brace lines up under the keyword that starts the overall statement. For example,

```
if (x > y) {
  System.out.println("x is larger " + x);
  max = x;
}
while (i < 10) {
  sum = sum + i;
  i++;
}
```

An alternative acceptable style derives from the Algol60 and Pascal communities. In this style each brace is kept on a separate line. In those languages the keywords `begin` and `end` were used to set off compound statements and they were placed on their own lines. For example,

```
if ( x > y)
{
  System.out.println("x is larger " + x);
  max = x;
}
```

However, be consistent and use the same style as others in your class, group, or company. A style should be universal within a given community. A single style simplifies exchanging, maintaining, and using each others' code.

SUMMARY

- An expression followed by a semicolon is an expression statement. Expression statements are the most common form of statement in a program. The simplest statement is the empty statement, which syntactically is just a semicolon.

- A group of statements enclosed by braces is a block. A block can be used anywhere a statement can be used. Blocks are important when you need to control several actions with the same condition. This is often the case for selection or iteration statements such as the `if-else` statement or the `for` statement, respectively.

- All statements don't end with a semicolon. The only statements covered in this chapter that end with a semicolon are expression statements and declaration statements.

- The general form of an `if` statement is

```
if ( BooleanExpr )
   Statement
```

- The general form of an `if-else` statement is

```
if ( BooleanExpr )
  Statement1
else
  Statement2
```

- When nesting an `if` statement inside an `if-else` statement or vice-versa, the `else` will always be matched with the closest unmatched `if`. This precedence can be overridden with a block statement to enclose a nested `if` statement or `if-else` statement.

- The general form of the `while` statement is

```
while ( BooleanExpr )
  Statement
```

- The general form of the `for` statement is

```
for ( ForInit ; BooleanExpr ; UpdateExpr )
  Statement
```

- Java includes the usual logical operators *and*, *or*, and *not*.

- The `break` statement can be used to terminate abruptly the execution of either a `switch` statement or a looping statement. When a `break` is executed inside any of those statements, the enclosing `switch` or loop statement is immediately terminated and execution continues with the statement following the `switch` or loop.

- The `continue` statement is used only inside a looping statement. When a `continue` is executed, the remaining portion of the surrounding loop body is skipped and the next iteration of the loop is begun. In the case of the `while` statement, the loop termination expression is tested as the next operation to be performed after the continue. In the case of the `for` statement, the update expression is evaluated as the next operation, followed by the loop termination test.

- The `switch` statement is an alternative to a sequence of `if-else-if-else...` statements. The `switch` statement can be used only when the selection is based on an integer-valued expression.

Review Questions

1. True or false? An expression is a statement.

2. How do you turn an expression into a statement?

3. True or false? All statements end with a semicolon. If false, give an example to show why.

4. True or false? An `if` statement is always terminated with a semicolon. If false, give an example to show why.

5. What are the values of the following Java expressions?

```
true && false
true || false
```

6. Write a Java expression that is true whenever the variable `x` is evenly divisible by both 3 and 5. Recall that `(x % y)` is zero if `y` evenly divides `x`.

7. What is printed by the following program fragment?

```
x = 10;
y = 20;
if ( x < y)
  System.out.println("then statement executed");
else
  System.out.println("else statement executed");
  System.out.println("when is this executed?");
```

8. What is printed by the following program, if 3 and 12 are entered as `x` and `y`? Now if you change the order and enter 12 and 3, what is printed?

```
//PrintMin.java - print the smaller of two numbers
import java.util.*;
class PrintMin {
  public static void main(String[] args) {
    Scanner scan = new Scanner(System.in);
    System.out.println("Type two integers.");
    int x = scan.nextInt();
    int y = scan.nextInt();

    if (x < y)
      System.out.println("The smaller is " + x);
    if (y < x)
      System.out.println("The smaller is " + y);
    if (x == y)
      System.out.println("They are equal.");
  }
}
```

9. For the declarations shown, fill in the value of the expression or enter `*illegal*`.

```
int  a = 2, b = 5, c = 0, d = 3;
```

EXPRESSION	VALUE
`b % a`	
`a < d`	
`(c != b) && (a > 3)`	
`a / b > c`	
`a * b > 2`	

10. What is printed by the following program fragment? How should it be indented to reflect what is really going on?

```
x = 10;
y = 20;
z = 5;
if ( x < y)
if ( x < z)
System.out.println("statement1");
else
System.out.println("statement2");
System.out.println("statement3");
```

11. How many times will the following loop print `testing`?

```
int i = 10;
while (i > 0) {
  System.out.println("testing");
  i = i - 1;
}
```

12. How many times will the following loop print `testing`?

```
int i = 1;
while (i != 10) {
  System.out.println("testing");
  i = i + 2;
}
```

13. What is printed by the following loop? See Question 6, on page 74.

```
int i = 1;
while (i <= 100) {
  if (i % 13 == 0)
    System.out.println(i);
  i = i + 1;
}
```

14. Rewrite the loop in the previous question using a `for` statement.

15. Rewrite the following loop, using a `while` statement.

```
for (i = 0; i < 100; i++) {
    sum = sum + i;
}
```

16. True or false? Anything that you can do with a `while` statement can also be done with a `for` statement and vice-versa.

17. How many times will the following loop go around? This is a trick question—look at the code fragment carefully. It is an example of a common programming error.

```
int i = 0;
while (i < 100) {
  System.out.println(i*i);
}
```

18. What is printed by the following program?

```
class Problem18 {
  public static void main(String[] args) {
    int i, j = 0;
    for (i = 1; i < 6; i++)
      if (i > 4)
        break;
      else {
        j = j + i;
        System.out.println("j= " + j + " i= " + i);
      }
    System.out.println("Final j= " + j + "i= " + i);
  }
}
```

EXERCISES

1. Write a program that asks for the number of quarters, dimes, nickels, and pennies you have. Then compute the total value of your change and print the amount in the form `$X.YY`. See Exercise 15 on page 43 in Chapter 2.

2. Write a program that reads in two integers and then prints out the larger one. Use Review Question 8, on page 40, as a starting point and make the necessary changes to `class PrintMin` to produce `class PrintMax`.

3. Modify the class `Average` from this chapter to print a special message if the first number entered is 0.

4. Write a program that reads in four integers and then prints out `yes` if the numbers were entered in increasing order and prints out `no` otherwise.

5. Write a program that prompts for the length of three line segments as integers. If the three lines could form a triangle, the program prints "Is a triangle." Otherwise, it prints "Is not a triangle." Recall that the sum of the lengths of *any* two sides of a triangle must be greater than the length of the third side. For example, 20, 5, and 10 can't be the lengths of the sides of a triangle because 5 + 10 is not greater than 20.

6. Write a program that tests whether the formula $a^2 + b^2 = c^2$ is true for three integers entered as input. Such a triple is a *Pythagorean triple* and forms a right-angle triangle with these numbers as the lengths of its sides.

7. An operator that is mildly exotic is the conditional operator `?:`. This operator takes three arguments and has precedence just above the assignment operators, as for example in

   ```
   s = (a < b)? a : b;
   // (a < b) true then s assigned a else s assigned b
   ```

   Rewrite the code for class `PrintMin`, in Review Question 8, on page 74, using this operator to eliminate the first two `if` statements. We chose not to use this operator because it is confusing to beginning programmers. It is unnecessary and usually can readily and transparently be replaced by an `if` statement.

8. Write a program that reads in integers entered at the terminal until a value of 0 is entered. A *sentinel* value is used in programming to detect a special condition. In this case the sentinel value is used to detect that no more data values are to be entered. After the sentinel is entered, the program should print out the number of numbers that were greater than 0 and the number of numbers that were less than 0.

9. Write a program that reads in integers entered at the terminal until a sentinel value of 0 is entered. After the sentinel is entered, the program should print out the smallest number, other than 0, that was entered.

10. Rewrite Exercise 8 on page 76 to print the largest number entered before the sentinel value 0 is entered. If you already did that exercise, only a few changes are needed to complete this program. Now code a further program that ends up printing both the smallest or minimum value found and the largest or maximum value found.

11. Rewrite Exercise 5 on page 76 to continue to test triples until the sentinel value 0 is entered.

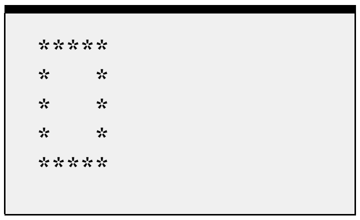
12. Write a program to input values as in Excercises 8, 9, or 10 above. This time run it with the input taken as a file through *redirection.* If the compiled Java program is *NumberTester.class,* then the command

    ```
    java NumberTester < myFile
    ```

    will use `myFile` for input. Redirection is possible with Unix or the Windows console window. This can save a lot of typing of input values for testing.

13. Write a program to print every even number between 0 and 100. Modify the program to allow the user to enter a number *n* from 1 through 10 and have the program print every *n*th number from 0 through 100. For example, if the user enters 5, then 0 5 10 15 20…95 100 are printed.

14. Write a program that will print out a box drawn with asterisks, as shown.



```
*****
*   *
*   *
*   *
*****
```

    Use a loop so that you can easily draw a larger box. Modify the program to read in a number from the user specifying how many asterisks high and wide the box should be.

15. Write a program that reads in numbers until the same number is typed twice in a row. Modify it to go until three in a row are typed. Modify it so that it first asks for "how many in a row should I wait for?" and then it goes until some number is typed that many times. For example, for two in a row, if the user typed "1 2 5 3 4 5 7" the program would still be looking for two in a row. The number 5 had been typed twice, but not in a row. If the user then typed 7, that would terminate the program because two 7s were typed, one directly after the other.

16. Write a program that prints all the prime numbers in 2 through 100. A prime number is an integer that is greater than 1 and is divisible only by 1 and itself. For example, 2 is the only even prime. Why?

    PSEUDOCODE FOR FINDING PRIMES

    ```
    for n = 2 until 100
      for i = 2 until the square root of n
        if n % i == 0 the number is divisible by i
          otherwise n is prime
    ```

    Can you explain or prove why the inner-loop test only needs to go up to the square root of *n*?

17. Write a program that prints the first 100 prime numbers. Use the same algorithm as in Exercise 16 on page 77, but terminate the program upon finding the 100th prime. Assume that the search needs to go no farther than $n = 10{,}000$.

18. Write a program that generates all Pythagorean triples (see Exercise 6 on page 76) whose small sides are no larger than *n*. Try it with $n < 200$. (*Hint*: Use two `for` loops to enumerate possible values for the small sides and then test to determine whether the result is an integral square.

19. Write a program that gives you a different message each day of the week. Use a `switch` statement. Take as input an integer in the range 1 through 7. For example, if 6 means Friday, the message might say, `Today is Friday, tGif.` If the user inputs a number other than 1 through 7, have the default issue an appropriate message.

20. Write a program that gives you a fortune based on an astrological sign. Use a `switch` statement to structure the code.

21. Write a program that generates an approximation of the real number *e*. Use the formula

$$e \approx 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots + \frac{1}{k!} + \ldots$$

where *k*! means *k* factorial = 1 * 2 * . . . *k. Keep track of term 1/*k*! by using a `double`. Each iteration should use the previous value of this term to compute the next term, as in

$$T_{k+1} = T_k \times \frac{1}{k+1}$$

Run the computation for 20 terms, printing the answer after each new term is computed.

22. Write your own pseudorandom number generator. A pseudorandom sequence of numbers appears to be chosen at random. Say that all the numbers you are interested in are placed in a large fishbowl and you reach in and pick out one at a time without looking where you are picking. After reading the number, you replace it and pick another. Now you want to simulate this behavior in a computation. You can do so by using the formula $X_{n+1} = (aX_n + c) \textbf{ mod } m$. Let *a* be 3,141,592,621, *c* be 1, and *m* be 10,000,000,000 (see Knuth, *Seminumerical Algorithms*, Addison-Wesley 1969, p. 86). Generate and print the first 100 such numbers as `long` integers. Let $X_1 = 1$.

Applet Exercise

Redo any one of the first ten exercises in this chapter but use an applet for input and output. You are to do so by modifying the following applet. Recall that the Applet exercise in Chapter 2, on page 44 introduced a special kind of Java program called an *applet*. Among other things, an applet may be used to create graphical output such as plots or diagrams. In order to be useful, most programs, including applets, need some way to receive input. For the regular applications that we've created so far, we've used `scan.nextInt()`, etc., to read values from the console. This exercise introduces you to the use of a `JTextField` object to get input from an applet. For this exercise we need to introduce two more methods: `init()` and `actionPerformed()`. As with the earlier applet exercise, for now you need only to concentrate on the body of the methods `init()` and `actionPerformed()`, treating the surrounding code as a template to be copied verbatim. The following applet reads two numbers and then displays their sum.

```
/* <applet code="AppletSum.class"
    width=420 height=100></applet> */
//AppletSum.java - Text input with an applet
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class AppletSum extends JApplet implements ActionListener
{
  JTextField inputOne = new JTextField(20);
  JTextField inputTwo = new JTextField(20);
  JTextField output = new JTextField(20);
  public void init() {
    Container pane = getContentPane();
    pane.setLayout(new FlowLayout());
    pane.add(new JLabel("Enter one number."));
    pane.add(inputOne);
    pane.add(new JLabel("Enter a number and hit return."));
    pane.add(inputTwo);
    pane.add(new JLabel("Their sum is:"));
    pane.add(output);
    inputTwo.addActionListener(this);
  }

  public void actionPerformed(ActionEvent e) {
    double first, second, sum;
    first = Double.parseDouble(inputOne.getText());
    second = Double.parseDouble(inputTwo.getText());
    sum = first + second;
    output.setText(String.valueOf(sum));
  }
}
```

Dissection of the *AppletSum* Program

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class AppletSum extends JApplet implements ActionListener
```

For now this code is all part of the template for an applet. The only part that you should change is to replace AppletSum with the name of your applet. The import statements tell the Java compiler where to find the various classes used in this program, such as JTextField and JApplet. The extends JApplet phrase is how we indicate that this class is defining an applet instead of a regular application. The implements ActionListener phrase is needed if we want to have our program do something when the user has finished entering data.

```
JTextField inputOne = new JTextField(20);
JTextField inputTwo = new JTextField(20);
JTextField output = new JTextField(20);
```

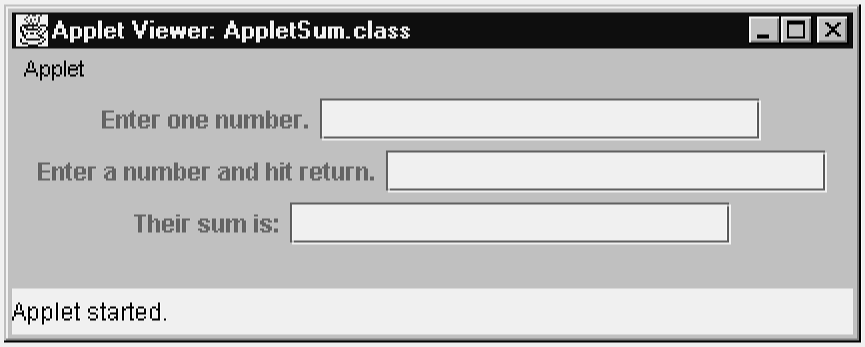
JTextField is a standard Java class. A JTextField is a graphical user interface component in which the user can enter text and the program can then read the text entered by the user. This program will create three text fields—two for the input values and one to display the sum of the first two. Note that these variables aren't declared inside any method. These variables need to be referenced by both the methods in this applet. You should think of them as part of the applet class AppletSum and not part of any one method. (We discuss such declarations in Section 6.9, *Static Fields and Methods*, on page 169.)

```
public void init() {
  Container pane = getContentPane();
  pane.setLayout(new FlowLayout());
```

The method init() is rather like main() for an applet. This method is called to initialize the applet. The first thing we do in the method is get the content pane. The *content pane* is the part of the applet used to display the various graphical components that we add to the applet. The setLayout() method is used to adjust the content pane so that when components are added, they are arranged in lines, like words of text in a word processor. The components are allowed to "flow" from one line to the next. For now, just always include the two statements in your init() method.

●
```
pane.add(new JLabel("Enter one number."));
pane.add(inputOne);
pane.add(new JLabel("Enter a number and hit return."));
pane.add(inputTwo);
pane.add(new JLabel("Their sum is:"));
pane.add(output);
```

The method `add()` is used to add some labels and text fields to the content pane of the applet. `JLabel` is a standard Java class that is used to place labels in graphical user interfaces. Unlike with a `JTextField`, the user can't enter text in a `JLabel` when the applet is running. The order in which we add all the components to the applet is important. The components are placed in the applet, like words in a word processor. That is, the components will be displayed from left to right across the applet until the next component won't fit, in which case a new line of components is started. Later, you'll learn more about controlling the placement of components, but for now just add them in the desired order and then adjust the width of the applet to get the desired appearance. Here is what the applet will look like.


Applet Viewer: AppletSum.class
Applet
Enter one number.
Enter a number and hit return.
Their sum is:
Applet started.

●
```
inputTwo.addActionListener(this);
```

The last statement in the `init()` method says that the `actionPerformed()` method in the applet referred to by `this` should be called when Return is hit in the text field, `inputTwo`.

●
```
public void actionPerformed(ActionEvent e) {
  double first, second, sum;
  first = Double.parseDouble(inputOne.getText());
  second = Double.parseDouble(inputTwo.getText());
  sum = first + second;
  output.setText(String.valueOf(sum));
}
```

This method is called each time the user hits Return with the cursor placed in the text field `inputTwo`. The method gets the text that was entered in the first two text fields by invoking their `getText()` methods. The resulting strings are then converted to floating point numbers, using the standard Java method `Double.parseDouble()`. Next the two numbers are added and stored in the variable `sum`. The floating point value stored in `sum` is converted to a `String`, using `String.valueOf()`, which can be used to convert any primitive value to a `String`. Finally the method `setText()` from the class `JTextField` is used to set the value of the text displayed in the `JTextField` object `output`.