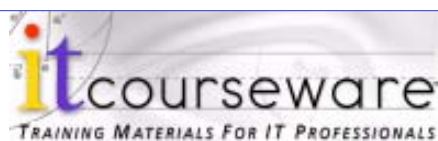

Fundamentals of Computer Programming



7400 E. Orchard Road, Suite 1450 N, Greenwood Village, CO 80111
303-302-5234 | 800-292-3716
SkillDistillery.com

FUNDAMENTALS OF COMPUTER PROGRAMMING

Student Workbook



For Skill Distillery student use only.
Unauthorized reproduction or distribution is prohibited.

FUNDAMENTALS OF COMPUTER PROGRAMMING

William A. Parette

Published by ITCourseware, LLC, 7400 E. Orchard Rd, Suite 1450N, Greenwood Village, CO 80111

Editor: Rob Roselius, Rick Sussenbach.

Editorial Assistant: Ginny Jaranowski

Special thanks to: Several instructors whose ideas and careful review have contributed to the quality of this workbook, including Andrew Boardman, Brandon Caldwell, Roger Jones, John Roach, Jamie Romero and Danielle Waleri, and the many students who have offered comments, suggestions, criticisms, and insights.

Copyright © 2016 by ITCourseware, LLC. All rights reserved. No part of this book may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying, recording, or by an information storage retrieval system, without permission in writing from the publisher. Inquiries should be addressed to ITCourseware, LLC, 7400 E. Orchard Rd, Suite 1450N, Greenwood Village, CO 80111. (303) 302-5280.

All brand names, product names, trademarks, and registered trademarks are the property of their respective owners.

CONTENTS

| | |
|--|----|
| Chapter 1 - Course Introduction | 9 |
| Course Objectives | 10 |
| Course Overview | 12 |
| Using the Workbook | 13 |
| Suggested References | 14 |
| Chapter 2 - Basic Concepts and Definitions | 17 |
| What is a Program? | 18 |
| "Hello, world!" | 20 |
| The Programming Process | 22 |
| Program Files and Program Execution | 24 |
| System Programs vs. Application Programs | 26 |
| Input - Process - Output | 28 |
| Programming Languages | 30 |
| Compiler Errors vs. Runtime Errors | 32 |
| Development Environments | 34 |
| Review Questions | 36 |
| Labs | 38 |
| Chapter 3 - Writing Simple Programs | 41 |
| Reading Input | 42 |
| Performing Numeric Calculations | 44 |
| Formatting Output | 46 |
| Decision Making | 48 |
| Iteration | 50 |
| Commenting Your Source Code | 52 |
| Good Programming Style | 54 |
| Labs | 56 |

FUNDAMENTALS OF COMPUTER PROGRAMMING

| | |
|--|-----|
| Chapter 4 - Data Types, Constants, and Variables | 59 |
| A Program's Purpose is to Process Data | 60 |
| Computer Memory | 62 |
| Data Can Be of Different Types | 64 |
| Named Data: Variables | 66 |
| Literal Data | 68 |
| Assignment | 70 |
| Printing Variables | 72 |
| Review Questions | 74 |
| Labs | 76 |
| Chapter 5 - Screen Output and Keyboard Input | 79 |
| Writing to the Screen | 80 |
| Characters That Have Special Meaning | 82 |
| Some Simple Formatting | 84 |
| Reading from the Keyboard | 86 |
| Prompting and Validating | 88 |
| Example 5 - Formatting Output Data | 90 |
| Review Questions | 92 |
| Labs | 94 |
| Chapter 6 - Expressions | 97 |
| Expressions: Where the Work Gets Done | 98 |
| Expression Evaluation: The Result | 100 |
| Arithmetic Expressions | 102 |
| Relational Expressions | 104 |
| Where are Relational Expressions Used? | 106 |
| And? . . . Or? | 108 |
| Precedence and Associativity | 110 |
| Example 6 - Calculating Miles Per Gallon | 112 |
| Review Questions | 114 |
| Labs | 116 |

| | |
|--|---------|
| Chapter 7 - Decision Making | 119 |
| Sequential Execution | 120 |
| What is Decision Making? | 122 |
| Simple Decisions: if | 124 |
| Two-Way Decisions: else | 126 |
| Code Blocks | 128 |
| Nesting Control Statements | 130 |
| Multi-Way Decisions: switch | 132 |
| Example 7 - Printing Letter Grades Based On Scores | 134 |
| Review Questions | 136 |
| Labs | 138 |
| Chapter 8 - Looping | 141 |
| Kinds of Loops | 142 |
| Iterative Loops | 144 |
| Code Blocks and Loops | 146 |
| Nested Loops | 148 |
| Conditional Loops | 150 |
| Infinite Loops | 152 |
| Example 8 - A Simple Menu Program | 154 |
| Review Questions | 156 |
| Labs | 158 |
| Labs (contd.) | 160 |
| Chapter 9 - Methods | 163 |
| Programming Without Methods | 164 |
| Reusable Code in a Method | 166 |
| The Starting Point | 168 |
| Variable Visibility: Scope | 170 |
| Parameters | 172 |
| Returning a Value | 174 |
| Method Stubs | 176 |
| Libraries | 178 |
| Example 9 – Square and Square Root | 180 |
| Review Questions | 182 |
| Labs | 184 |

| | |
|---|-----|
| Chapter 10 - Data Collections – Arrays | 187 |
| Scalar Data vs. Data Collections | 188 |
| What is an Array? | 190 |
| Accessing Array Elements | 192 |
| Multidimensional Arrays | 194 |
| Array Initialization | 196 |
| Example 10 - Calculating Average Rainfall | 198 |
| Review Questions | 200 |
| Labs | 202 |
| Chapter 11 - Debugging | 205 |
| What is Debugging? | 206 |
| Commenting Out Code | 208 |
| Simple Debugging with Print Statements | 210 |
| Making Debugging Print Statements Conditional | 212 |
| Programs that Help You Debug Programs | 214 |
| Example 11 - Debug Statements | 216 |
| Review Questions | 218 |
| Labs | 220 |
| Appendix A - Additional Exercises | 223 |
| Making Change | 224 |
| Index | 227 |

CHAPTER 1 - COURSE INTRODUCTION

COURSE OBJECTIVES

- * Explain what computer programs are and what computer programming is about.
- * Write and compile simple computer programs.
- * Describe basic computer language data types.
- * Interact with computer programs using your terminal screen and keyboard.
- * Evaluate expressions used in computer programs.
- * Design the sequential execution, and flow of decision making, in a program.
- * Write programs that use loops to perform repetitive tasks.
- * Design and write procedural programs that use methods.
- * Use basic debugging techniques to solve programming problems and increase program quality.
- * Use arrays for managing program data.

This course introduces you to the fundamental concepts, semantic elements, and vocabulary of computer programming.

The material you learn here is illustrated using Java language examples; most of these same examples can easily be converted to the C, C++, or C# languages. Upon completion of this course, you will be prepared for training, or introductory self-study, in these languages and others.

Initially, the programs you write will simply be typed in from the many examples presented in this student workbook. However, you will also be asked to code some programs "from scratch." These programs will start out small and simple as you are carefully guided through the concepts and facilities that will allow you to make your programs more interesting and complex.

COURSE OVERVIEW

- ✧ **Audience:** People with no computer programming background who want to learn the basics. This is a "first course" in computer programming.
- ✧ **Prerequisites:** Although no programming experience is required, it is assumed that you have used a computer before, whether for word processing, spreadsheets, or even playing games. You should know what a file is — how to create one, how to put data in it, where the file is stored, and how to find it at a later time. Basic arithmetic skills and the ability to think logically are necessary for programming. Reasonable typing skills are critical.
- ✧ **Classroom Environment:**
 - One workstation per student. However, you are encouraged to team up, share ideas, and work on some of the later exercises in small groups.

USING THE WORKBOOK

This workbook design is based on a page-pair, consisting of a Topic page and a Support page. When you lay the workbook open flat, the Topic page is on the left and the Support page is on the right. The Topic page contains the points to be discussed in class. The Support page has code examples, diagrams, screen shots and additional information. **Hands On** sections provide opportunities for practical application of key concepts. **Try It** and **Investigate** sections help direct individual discovery.

In addition, there is an index for quick look-up. Printed lab solutions are in the back of the book as well as on-line if you need a little help.

The Topic page provides the main topics for classroom discussion.

The Support page has additional information, examples and suggestions.

JAVA SERVLETS

THE SERVLET LIFE CYCLE

- * The servlet container controls the life cycle of the servlet.
 - When the first request is received, the container loads the servlet class
 - The container uses a separate thread to call
 - The container calls the `destroy()`
- As with Java's `finalize()` method, don't count on this being called.
- * Override one of the `init()` methods for one-time initializations, instead of using a constructor.
 - The simplest form takes no parameters.


```
public void init() {...}
```
 - If you need to know container-specific configuration information, use the other version.


```
public void init(ServletConfig config) {...}
```
 - Whenever you use the `ServletConfig` approach, always call the superclass method, which performs additional initializations.


```
super.init(config);
```

Page 16 Rev 2.0.0 © 2002 ITCourseware, LLC

CHAPTER 2

SERVLET BASICS

Hands On:

Add an `init()` method to your *Today* servlet that initializes along with the current date:

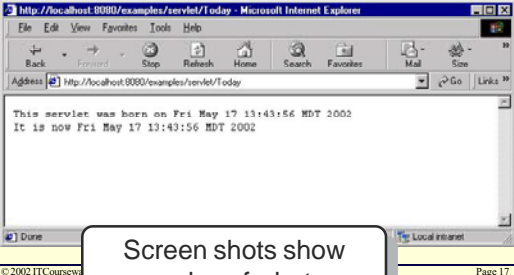
Today.java

```
...
public class Today extends GenericServlet {
    private Date bornOn;
    public void service(ServletRequest request,
        ServletResponse response) throws ServletException, IOException
    {
        ...
        Servlet was born on " + bornOn.toString();
        " + today.toString();
    }
}
```

The `init()` method is called when the servlet is loaded into the container.

Code examples are in a fixed font and shaded. The on-line file name is listed above the shaded area.

Callout boxes point out important parts of the example code.



Screen shots show examples of what you should see in class.

© 2002 ITCourseware, LLC Page 17

Topics are organized into first (*), second (➤) and third (▪) level points.

Pages are numbered sequentially throughout the book, making lookup easy.

SUGGESTED REFERENCES

Cadenhead, Rogers. 2012. *Sams Teach Yourself Java in 21 Days (6th Edition)*. Sams, Indianapolis, IN. ISBN 978-0672335747.

Horstmann, Cay and Gary Cornell. 2012. *Core Java 2, Volume I: Fundamentals (9th Edition)*. Prentice Hall PTR, Upper Saddle River, NJ. ISBN 978-0137081899.

Savitch, Walter. 2011. *Java: An Introduction to Problem Solving and Programming, 6th Edition*. Prentice Hall, Englewood Cliffs, NJ. ISBN 978-0132162708.

Schildt, Herbert. 2014. *Java, A Beginner's Guide (6th Edition)*. McGraw Hill, New York, NY. ISBN 978-0071809252.

Shackleford, Russell L. 1997. *Introduction to Computing and Algorithms*. Addison-Wesley Publishing Co. Menlo Park, CA. ISBN 978-0201314519.

Sierra, Kathy and Bert Bates. 2005. *Head First Java (2nd Edition)*. O'Reilly & Associates, Sebastopol, CA. ISBN 978-0596009205.

<http://www.javaworld.com/blog/java-101>

<http://www.javaranch.com>

<http://www.oracle.com/technetwork/java>

CHAPTER 2 - BASIC CONCEPTS AND DEFINITIONS

OBJECTIVES

- ✧ Define computer programming.
- ✧ Describe the purpose of a computer program.
- ✧ Describe the steps involved in writing a computer program.
- ✧ List some of the files created during the programming process and where they are stored.
- ✧ Differentiate between system programs and application programs.
- ✧ Diagram the flow of information through a computer program.
- ✧ List several popular programming languages.
- ✧ Differentiate between a programming language and a development environment.

WHAT IS A PROGRAM?

- * *A computer program* is a set of detailed instructions that tell a computer to perform some specific task.
- * How would you tell someone how to do something as simple as picking up a glass that is sitting on a table in front of them?
 1. Extend your arm to bring the forearm parallel to the table.
 2. Open the fingers of the hand.
 3. Move the arm and hand such that the fingers of the hand gently encircle the glass.
 4. Close the fingers firmly, but gently, around the glass.
 5. Raise the arm so that the glass rises off the table.

➤ And so on . . .
- * How do you tell a computer to add two numbers?
 1. Retrieve the first number from its memory location.
 2. Retrieve the second number from its memory location.
 3. Add the two numbers.
 4. Store the result of the computation in some other memory location.

➤ And so on . . .
- * Did you notice the detail? The logic? The careful sequencing?

Computer programming is the art of communicating algorithms to computers. An *algorithm* is a computational procedure whose steps are completely specified and elementary.¹

A program has the instructions that tell the computer what we need done. Each time a computer needs to perform a specific task, it has to follow the same instructions — the computer program — over and over again.

Also, computers must be given their instructions at a very low level of detail. You can't just tell the computer to perform some task. Instead, you have to give very detailed, specific instructions on how to perform every step of the task.

¹Al Kelley & Ira Pohl, *A Book on C*.

"HELLO, WORLD!"

- * The best way to learn about programming is by writing programs.
- * The classic program to write, no matter what language you're using, is a program that prints the words "Hello, world!" on your screen:

```
// A simple Hello, world program
// Note: The file name must be the same as the
// class name, with a .java extension.

package examples;

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

Hands On:

The instructor will lead you through the steps required for the particular programming environment used in the classroom to write, compile, and execute this program. You'll finish writing, compiling, and running this program before continuing on to the next page.

For each language there are certain rules you, the programmer, must follow.

- For Java programs, you must write your program statements (called "source code") in a file whose name ends in *.java*.
 - Name your source code file for this example *Hello.java*.
- Java programs are case sensitive: it makes a difference whether you use uppercase (capital) letters or lowercase letters. Make sure your keyboard's <Caps Lock> key isn't on, and enter the statements exactly as they're shown on the facing page.
- As you create your program, notice that each statement ends with a semicolon, ; , which marks the end of (terminates) the statement.
- Java is an object-oriented language. In such a language, all of your program code will be placed in a class. The Java programs shown in the examples all use a class, but an explanation of classes will not be presented until later.

THE PROGRAMMING PROCESS

- ✧ The programmer performs several steps when writing a computer program.
 1. **Write** the source code for your program.
 2. **Compile** the source code with an appropriate compiler.
 3. **Execute** the compiled, executable program to test it.
 4. **Debug** the program — find the errors and fix them.
 5. Repeat the process as needed.

Source code: the statements of the programming language (the instructions) that you have written to perform your specific application task. You put your source code in a file using any text editor or word processor (make sure you save the file in a text-only mode!).

Executable code: the instructions that have been translated from your source code statements down to the most elementary level — machine instructions — that the computer executes directly.

Source code is much closer to human language than executable code is. Programming languages are designed with this in mind. Some languages are terse and somewhat hard to read and are intended for expert programmers. Others are verbose and more like human languages, with nouns and verbs, and are intended to help nonexpert programmers get started. But when a program is compiled, the compiler translates the program statements into the same numeric machine instructions, no matter which language was used for the source code.

The process of writing, compiling, running, changing, compiling, running, changing, compiling, running, etc. will quickly become very natural as you write more programs.

PROGRAM FILES AND PROGRAM EXECUTION

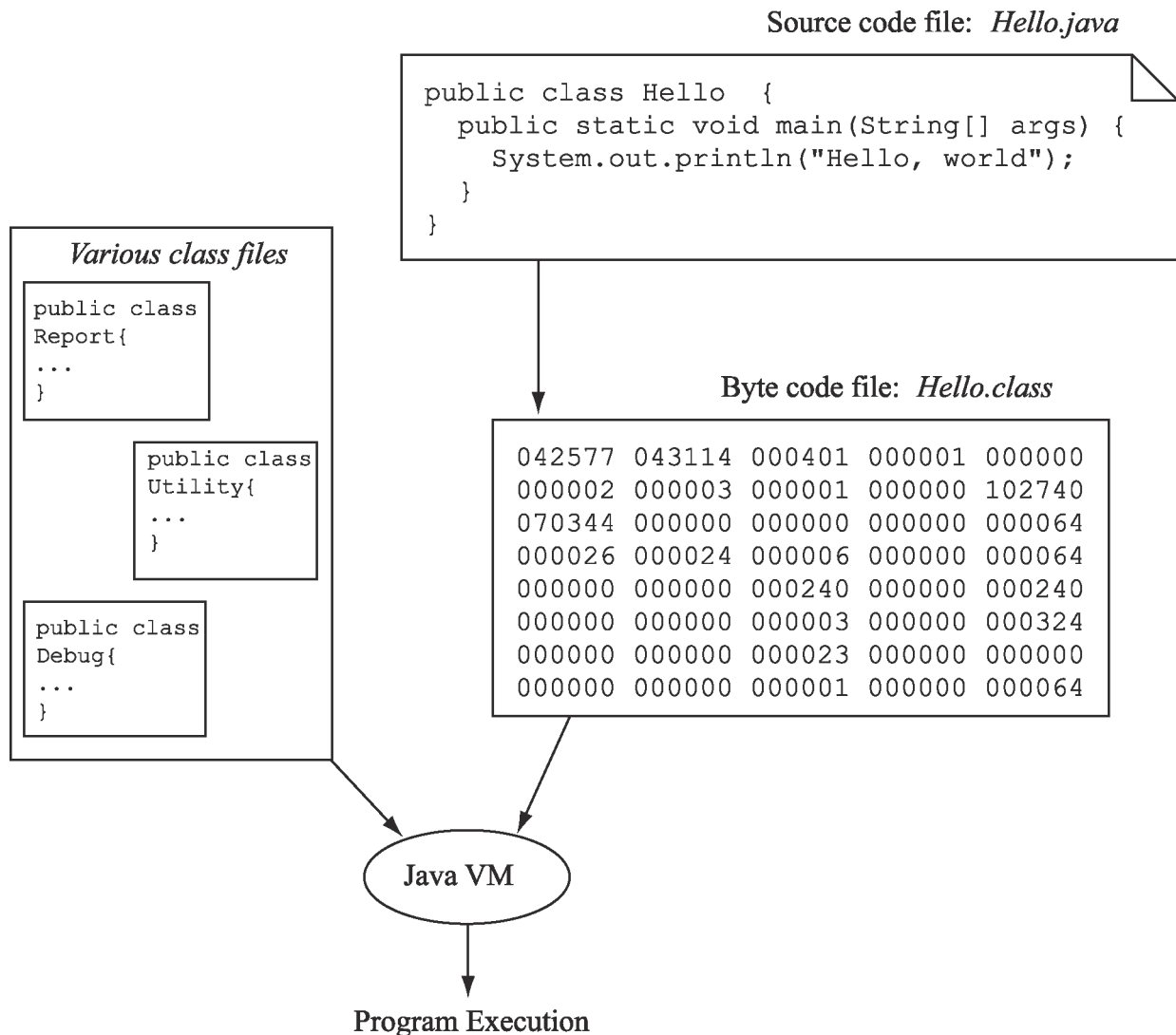
- * Many different files will be created, used, and (sometimes) removed during the compilation process.
- * For languages like Java:
 - You will write your program instructions (called *statements*) in a *source code file*.
 - The compiler will put the translated instructions (called *byte code*) into what is called a *class file*.
 - Your own class file is combined with other class files written by other programmers in your company or with class files that were installed as part of the Java system.
 - To run a Java program, you must use the *Java Virtual Machine (VM)*. The VM reads the byte code produced by the compiler, combines it with byte code from other classes and executes the program that they contain.

You have control over the location of the source code and the executable code files. You tell the compiler where your source code is and you also tell it where you want it to put the executable file.

Typically these files will be kept in your *current working directory* or your *current folder*.

In Java, while the source code file contains the text of the program you have written, the executable code file contains something called *byte code* and the file name will have *.class* on the end. The Java Virtual Machine executes the byte code to run your program.

All other files are located in directories that are known to the compiler. You generally don't have to worry about these files; we just want to introduce you to a few terms that you'll encounter again if you continue to learn about programming languages.



SYSTEM PROGRAMS VS. APPLICATION PROGRAMS

- * *System programs* are used by the computer for its own operations.
 - Operating system programs.
 - Language compiler programs.
 - *Utility programs*: editors, sort utilities, text search tools, file display programs, etc.
- * *Application programs* perform some user-oriented data processing task.
 - Payroll programs.
 - Customer tracking programs.
 - Inventory programs.
 - Accounting programs: accounts receivable programs, accounts payable programs, etc.
 - Game programs.

System programs are written by the computer manufacturer, or some third-party company, to make the computer useful to users. Operating system programs allow us to use the computer without having to speak the binary language that is the computer's "native tongue," or know the mundane details of its hardware configuration. Language compiler programs take the computer programs we write and translate them into the language that the computer understands.

Utility programs can be thought of as a subset of system programs. Over the years, programmers have written programs for many common tasks that are performed so frequently — sorting, searching, editing text, and such — that many computer manufacturers include these system utility programs with their computers.

Application programs are the programs we write on the user side. We want to use the computer to help our company do its business better and faster. So, we write computer programs to track, process, and format the data in ways that are useful to the business goals. Whether it is to process employee data, customer data, corporate data, whatever, these programs typically perform an application — or process — that is useful to our company and are written by the programmers employed by the company.

INPUT - PROCESS - OUTPUT

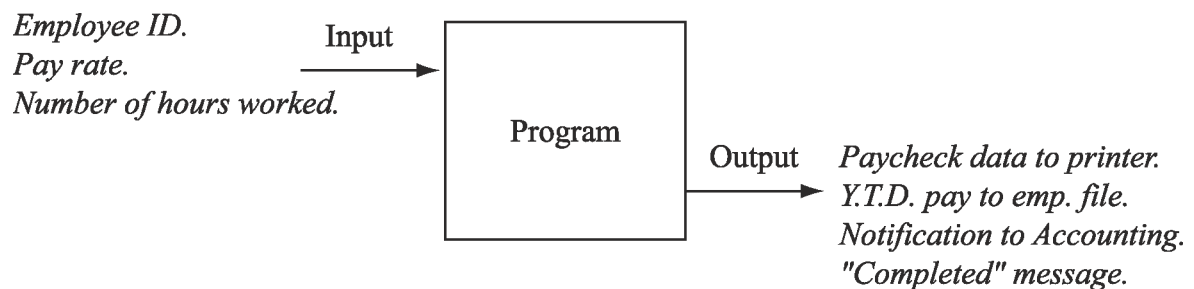
* Regardless of the type of application, a computer program almost always performs three basic functions:

1. **Input** some data.
 - From a file.
 - From the keyboard.
2. **Process** the data.
 - Formatting.
 - Calculating.
 - Searching.
 - Sorting.
3. **Output** the processed data.
 - To a file.
 - To the screen.
 - To a printer.

Input involves gathering data from where it is stored and bringing it into your application program so that it can be manipulated. The payroll program will input the employee data from the employee file on the disk. Other data about the employee may be input from the keyboard where the user is running the program. The payroll program may "pass" data as an input to other parts of the payroll program during the payroll program's normal operation.

Process involves manipulating the data in a variety of different ways. The payroll program will process the hours worked and the pay rate for an employee to determine his or her gross pay. The gross pay will be further processed, along with tax rates, to determine his or her net pay.

Output involves taking the results of the process — and, perhaps, even some of the original input data — and giving it back to the user. The payroll program will output some of its data back to the employee file; other data may be output to the user's screen; while still other data will be output in the form of paychecks on a printer.



PROGRAMMING LANGUAGES

- * Programming languages are to a computer as English, French, Dutch, and German are to a human — they're used to communicate.
 - The communication we send to the computer is simply a series of instructions on how to perform a specific task.
- * Different languages can be used to instruct the computer to do the same task.
 - *Assembly language* (or just *Assembler*): One step away from the numeric *machine language* the computer uses for its most basic operation; suitable for system-level programs.
 - *Fortran* (FORmula TRANslation): Developed by scientists to help write scientific, or "number crunching," programs.
 - *COBOL* (COmmon Business-Oriented Language): Originally developed by the U.S. Navy to write business programs.
 - *PL/1* (Programming Language/1): Developed by IBM as a "best of both worlds" language; supports features of both Fortran and COBOL, as well as others.
 - *C* (based on an old language named *B*): Developed at Bell Laboratories; used originally to write system-level software, but later used for all kinds of programs.
 - *C++* (beyond C): Also developed at Bell Labs as a "better C" with extensions to support object-oriented features.
 - *Java* (as in coffee, which programmers drink a lot of . . .): Developed by Sun Microsystems for distributed programming of all kinds; based on C and others.

In addition to the languages on the facing page, there are many, many more. BASIC, Smalltalk, RPG, Algol, Ada, Pascal, Modula, LISP, Python, Tcl, SQL, Awk, KornShell, Perl, C#, Visual Basic, Ruby, Groovy, Scala, Objective-C, PHP, JavaScript and many others were developed with particular types of system and application programming in mind.

In this immense programming language landscape, only a handful have persisted in widespread use over the years. Among these, C and most of its derivatives have proven themselves to be robust, well-rounded, and feature-full languages that are suited for many different types of programming tasks — both system programs and application programs. The C programming language has influenced many other languages, partly because it's available on a wider variety of computer systems than perhaps any other language, and partly because many of the engineers who design new languages first learned programming in C. C++, Java, Perl, C#, and many other languages copied their basic syntax features from C.

COMPILER ERRORS VS. RUNTIME ERRORS

- * Debugging is simply the process of finding and fixing errors that you have unknowingly placed in your program; there are two types:
- * *Compiler errors* are generated during the compilation of your program.
 - These are mistakes that you've made in using the syntax and grammar of the language; the compiler can't understand your instructions.
- * *Runtime errors* are generated when you execute (*run*) your program.
 - These are mistakes that you've made in the design and logic of your program; the compiler and the computer understand your instructions, but you've told it to do something unintended.

```
if ( year > )
```

- If `year` is greater than what?

- You can't compile your program if there's a compiler error; if your program isn't compiled, you can't run it.

Y2k.java

```
int year = 14;

if (year < 2000)
    System.out.print("You'll be ");
else
    System.out.print("You were ");

System.out.print(2000 - year + age);
System.out.println(" years old in the year 2000.");
```

- `year` should have been set to 2014, not just 14!

Compiler errors generally indicate that you have a wrong instruction, or that an instruction was typed incorrectly, in your program. These are generally easier to fix because you simply have to type in the correct syntax for the instruction in your program and try compiling your program again. Most compilers will even tell you which line contains the error and what sort of error it is.

Runtime errors are generally harder to fix, because they're harder to find. They indicate a logic error in your program. The program compiles correctly, but it does the wrong thing or ends incorrectly without producing the desired results. The program might even work correctly with some data, but incorrectly with other data.

Try It: Compile and run the Y2k program in your chapter directory. Does the output look correct?

DEVELOPMENT ENVIRONMENTS

- * On some systems you use separate utilities to perform each of the separate steps in the programming process:
 - An editing program (such as *notepad* or the UNIX *vi* editor).
 - A compiler program (such as the `javac` command).
 - A virtual machine (such as the `java` command).
 - A debugging program (such as *jdb*).
- * Some software vendors provide for all of the steps of the programming process in a single, combined utility program, called an *Integrated Development Environment* (IDE).
 - Eclipse
 - NetBeans
 - IntelliJ IDEA
- * All compiler vendors adhere to published language standards defining common syntax and semantics.
 - The products listed above all adhere to standards defined by Oracle.
 - You only have to learn the language once and then you can use any development environment.
- * Don't confuse learning the programming language with learning the development environment.

It is very important to notice the difference between a particular computer programming language and a particular development environment. Eclipse and NetBeans provide a user-friendly environment for writing computer programs. The *language* that you use to write your programs is the same for each.

REVIEW QUESTIONS

1. What is computer programming?
2. What is an algorithm?
3. What is the purpose of a computer program?
4. Explain the difference between a system program and an application program.
5. List the three steps that most computer programs perform to accomplish their task.
6. What are some popular programming languages?
7. List the steps involved in writing a computer program.
8. Name some of the files created during the programming process and where they are stored.
9. What's the difference between a compiler error and a runtime error?
10. Explain the difference between a programming language and a development environment.

LABS

- ❶ Change the "Hello, world!" program to print "Hello, " followed by your first name. Compile and run your modified program.
(Solution: *HelloName.java*)
- ❷ Change the "Hello, world!" program again to use separate `System.out.println(...)` calls to print "Hello," followed by your first name on two different lines. Compile and run it again (in fact, in each lab exercise from now on, compile and run your program after making the given change).
(Solution: *TwoPrints.java*)
- ❸ Change the "Hello, world!" program again by inserting one or more `\t` in the beginning of one of the character strings being printed and look for any differences in the output.
(Solution: *HelloTabs.java*)
- ❹ Change the "Hello, world!" program yet again. Have it print a row of asterisks above the text and another line of asterisks below.

Add an asterisk at the beginning and the end of each line of the message itself. Experiment using spaces and tabs, and see if you can get the message to appear in the center of a box made up of asterisks.

(Solution: *HelloStars.java*)

- ❺ Write a new program that prints the following on the screen:

```
Testing ...
... 1 ...
... 2 ...
... 3 ...
```

Can this program be written with one `System.out.println(...)` call?... with more than one?

(Solution: *Testing123.java*)

You will find that the labs in this workbook contain more exercises than can be completed during a normal lab session in class. There are several reasons for this:

- To provide a selection of exercises from which you can choose.
- To leave many exercises you can work on in your own time, at work or at home.
- To provide additional challenges for students who already have some programming experience.

So, don't worry if you don't finish all, or even most, of the labs in the time available during class. Just start at the beginning and keep working until the lab session is over.

CHAPTER 3 - WRITING SIMPLE PROGRAMS

OBJECTIVES

- * Write small programs to perform simple tasks.
- * Modify programs.
- * Use the *edit–compile–run–debug* cycle of program development.

READING INPUT

✧ To read input into a program:

1. Create a variable to hold the input.
2. Prompt the user, telling him what he needs to enter.
3. Read the input into the variable you created.

✧ Here's how to read input into a variable:

```
...  
java.util.Scanner scanner =  
    new java.util.Scanner(System.in);  
  
int num = 0;  
System.out.print("Please enter a number: ");  
num = scanner.nextInt();  
  
scanner.close();  
...
```

- num is the name of the variable.
- nextInt() is how Java takes whatever the user types and stores it in a numeric variable.

This is the first version of a program that you will modify several times. This version of the program does some simple input of data from the keyboard and puts the data in one of its variables. No processing is done here, nor is any output produced.

Hands On:

Enter this program into a new file.

```
package examples;

public class Squares {
    public static void main(String[] args) {
        // Use a Scanner to read from standard in
        java.util.Scanner scanner = new java.util.Scanner(System.in);

        int num = 0;

        System.out.print("Please enter a number: ");
        num = scanner.nextInt();

        scanner.close();
    }
}
```

Save this source code as a file named *Squares.java* in a directory called *examples*. Compile your program, and then run it. The instructor will help you complete this before moving on to the next page.

When compiled and then executed, this program displays:

```
Please enter a number:
```

It then waits for you to input something. When you do, though, the program doesn't do anything; it just returns you to your working environment.

PERFORMING NUMERIC CALCULATIONS

- * A program can perform a numeric calculation and store the result in a variable.
- * Numeric calculation statements in a program look very much like arithmetic equations.

```
sum = 5 + 3;  
diff = 32 - 25;  
div = 125 / 5;
```

- In most computer languages, the symbol for multiplication is an asterisk: *.

```
product = 7 * 52;
```

- * Calculations can include both simple numbers and variable names.

```
area = radius * radius * 3.14159;
```

Hands On:

Modify your program, adding the lines shown in bold below. Here, the program will take the data that was input and perform a simple mathematical operation on it. Again, no output is yet performed.

```
package examples;

public class Squares {
    public static void main(String[] args) {
        // Use a Scanner to read from standard in
        java.util.Scanner scanner = new java.util.Scanner(System.in);

        int num = 0;
        int result = 0;

        System.out.print("Please enter a number: ");
        num = scanner.nextInt();

        result = num * num;

        scanner.close();
    }
}
```

When recompiled and then executed, this program still just displays its prompt. You can go ahead and enter something, but you won't see any results:

```
Please enter a number: 8
```

FORMATTING OUTPUT

✧ By combining your own text with values calculated at runtime, you can produce clearly formatted output.

✧ You can use several separate output statements:

```
System.out.print("Distance: ");  
System.out.print(data);
```

➤ You usually need to specify where you want the output line to end:

```
System.out.println(" miles.");
```

✧ You can combine all of the data into a single output statement:

```
System.out.println("Distance: " + data + " miles.");
```

Hands On:

Modify your program as shown, adding the two output statements. Finally, we see a version of the program that has all three components: input, process, and output.

```
package examples;

public class Squares {
    public static void main(String[] args) {
        // Use a Scanner to read from standard in
        java.util.Scanner scanner = new java.util.Scanner(System.in);

        int num = 0;
        int result = 0;

        System.out.print("Please enter a number: ");
        num = scanner.nextInt();

        result = num * num;
        System.out.print("The result of squaring " + num);
        System.out.println(" is: " + result);

        scanner.close();
    }
}
```

When recompiled and then executed, this program now displays output too. Type in a number at your prompt:

```
Please enter a number: 8
The result of squaring 8 is: 64
```

Try running your program a few times with some larger numbers:

```
Please enter a number: 46340
The result of squaring 46340 is: 2147395600
```

```
Please enter a number: 46341
The result of squaring 46341 is: -2147479015
```

... What's this?

DECISION MAKING

- * Programs from time to time need to check various conditions and decide what to do next; for example,
 - Making sure the data that was input makes sense for the program's processing.
 - Making sure the computer is ready for an operation that the program needs to perform, such as writing data into a file.
- * An `if` statement (or conditional) evaluates an expression and takes a specified action only if the expression is true.

```
if ( age < 0 ) {  
    System.err.println ("Age can't be less than zero!");  
}
```

Hands On:

Modify your program, adding the lines shown in bold. Here, our program will do some simple decision making. We will discuss decision making in great detail later. We'll use it here to keep our program from trying to calculate a result that's too large for the computer to represent.

```
package examples;

public class Squares {
    public static void main(String[] args) {
        // Use a Scanner to read from standard in
        java.util.Scanner scanner = new java.util.Scanner(System.in);

        int num = 0;
        int result = 0;

        System.out.print("Please enter a number: ");
        num = scanner.nextInt();

        if (num < 46341) {
            result = num * num;
            System.out.print("The result of squaring " + num);
            System.out.println(" is: " + result);
        }
        else {
            System.err.println("Input # " + num + " is too large.");
        }
        scanner.close();
    }
}
```

When you execute this program, try entering a number that's too large:

```
Please enter a number: 46341
Input # 46341 is too large.
```

ITERATION

- ✧ Computers are well suited to repetitive tasks (much better suited than people are).
- ✧ There are several ways of setting up a program to iterate (re-execute the same statements many times).
 - Define a fixed number of iterations.
 - Have the number of iterations determined by the program at runtime.
 - Iterate indefinitely, stopping only when some condition changes at runtime.

Hands On:

We'll cover looping in more detail later; we use it here to have our program calculate the squares of five different input numbers. Modify your program again, adding the code shown in bold below. Don't miss the closing curly brace of the loop, four lines from the end.

```
package examples;

public class Squares {
    public static void main(String[] args) {
        // Use a Scanner to read from standard in
        java.util.Scanner scanner = new java.util.Scanner(System.in);

        int num = 0;
        int result = 0;
        int count = 0;

        for (count = 0; count < 5; count = count + 1) {
            System.out.print("Please enter a number: ");
            num = scanner.nextInt();

            if (num < 46341) {
                result = num * num;
                System.out.print("The result of squaring " + num);
                System.out.println(" is: " + result);
            }
            else {
                System.err.println("Input # " + num + " is too large.");
            }
        }
        scanner.close();
    }
}
```

Compile and run your modified program, entering five different numbers.

COMMENTING YOUR SOURCE CODE

- * Comments are very useful to programmers but are ignored by the compiler.
 - Comments help those who *are not* familiar with the syntax of your chosen language to be able to read your program and understand what it does.
 - Comments help those who *are* familiar with the syntax of the language to understand more clearly your programming intent.
- * A `//` indicates the start of a Java comment that will continue to the end of the line:

```
int count1 = 0;  //Define an integer variable
```

- * Another style of Java comment starts with `/*` and ends with `*/`:

```
int count2 = 0;  /* Define an integer variable */
```

```
/******  
* Define 3 integer variables *  
*****/  
/* Define an integer variable */  
int count3 = 0;  
  
/*  
*Define an integer variable  
*/  
int count4 = 0;  
  
int /*Define*/ count5 /*an*/ = /*integer*/ 0 /*variable*/;
```

All good programming languages let the programmer include comments in the source code, like footnotes in a book.

Java uses the exact same comment markers that C++ uses.

C uses just the `/* ... */` comment markers, which the creators of Java borrowed:

```
/* Define 3 integer variables */  
int count, input, result;
```

Perl, and UNIX scripting languages like Shell and Awk, use the pound sign:

```
# Create 3 variables in Perl  
$count, $input, $result;
```

Fortran uses a C or an asterisk in the first position on the line; modern versions of Fortran can use an exclamation instead:

```
C Define 3 variables in Fortran  
INTEGER :: count, input, result ! All three are integers
```

COBOL uses an asterisk. A COBOL comment can't be on the same line as a statement:

```
* DEFINE 3 INTEGER VARIABLES IN COBOL  
01  COUNT  PIC 99 COMPUTATIONAL  
01  INPUT  PIC 99 COMPUTATIONAL  
01  RESULT PIC 99 COMPUTATIONAL
```

MS-DOS batch files use the word `rem` (which is short for `remark`):

```
rem Create 3 variables in MS-DOS  
set count=0  
set input=0  
set result=0
```

GOOD PROGRAMMING STYLE

- * Like comments, the format of your source code (indentation, space between symbols and lines) helps others read your programs.
- * Some programming languages have formatting rules that must be followed.
- * Others are called *free-format* languages because you can format your program's source code in any way that looks good to you:

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

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

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

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

Formatting of your program's source code is an important part of the program development process. It's worth a little bit of extra time to format your program's source code so that other people can read it. The format of your program should show the reader the structure of the program, as well as give a good indication of the program's flow of execution. It should help a reader visually distinguish different functional elements of your program, just as paragraphs help a reader visually distinguish different elements of a story.

There are as many formatting requirements as there are languages.

Assembler languages are typically formatted so that there is space for an optional label. The label must be followed by a space and the instruction. The instruction must be followed by another space and other operands.

FORTRAN programs leave the first 6 columns of a line available for an optional number-label. Column 7 is usually blank, but, if not, indicates that the line is a continuation of the previous line in the code.

COBOL programs have an *A* and a *B* paragraph. Paragraph headers start in the *A* paragraph, while the paragraph body starts in the *B* paragraph.

However, Java, C and C++ are what we call free-format languages. This means that you can format your source code in any way that feels comfortable to you. As shown on the facing page, anything is possible, though some of it is unreadable.

The last two examples on the facing page are popular styles used by many programmers.

Whichever formatting style you decide to use, *be consistent* in its use.

LABS

- 1 Take the last version of the square program that you worked on in this chapter and modify it. Try adding several comments to the program that describe its operation based on what you learned about it in this chapter.

(Solution: *CommentSquare.java*)

- 2 Modify the program from the previous exercise so that it asks the user for a number in a different way. The program currently prints:

```
Please enter a number:
```

Change the format of the question so that it looks like:

```
Please enter a number  
(less than 46341):
```

(Solution: *FormatInSquare.java*)

- 3 Modify the program from the previous exercise so that it formats the output differently. The program currently prints:

```
Please enter a number  
(less than 46341): 5  
The result of squaring 5 is: 25  
Please enter a number  
(less than 46341): 3333  
The result of squaring 3333 is: 11108889  
Please enter a number
```

Have it format the output as follows:

```
Please enter a number  
(less than 46341): 5  
The result of squaring 5 is:           25  
Please enter a number  
(less than 46341): 3333  
The result of squaring 3333 is:       11108889  
Please enter a number
```

(Solution: *FormatOutSquare.java*)

CHAPTER 4 - DATA TYPES, CONSTANTS, AND VARIABLES

OBJECTIVES

- * Describe the primary purpose of a computer program.
- * Explain where data comes from.
- * Write programs using different types of data.
- * Explain what constant, or literal, data is.
- * Use variables to hold data in a program.
- * Use assignment to place data in a variable.

A PROGRAM'S PURPOSE IS TO PROCESS DATA

- * Programs are really all about information: data.
 - A human resources program would process information about employees, like their names, dates of employment, rates of pay, and so on . . .
 - It might allow a user to enter data identifying a certain employee, retrieve data about the employee from a file or database, and display the information in a window so the user can view and edit it.
 - An employee might use a computer-aided design program to draw a design for a product, inputting positions and sizes of components.
 - The data could be sent to a plotter to create blueprints, or saved in a file for later use.
 - A factory control program might read design data from a file.
 - It could translate the data into instructions it sends to a robotic arm, causing it to weld parts of a car together.
 - A game program maintains data on the number of players, their current scores and positions in the game, and actions they can take.
 - It translates these into screen animation commands, instructions to a sound device, etc.
- * Computer programs can do this processing much faster and more accurately than people can.

Information Technology (IT), Information Systems (IS), Management Information Systems (MIS), and Information Theory are all general terms for professions involved with processing information using computers.

COMPUTER MEMORY

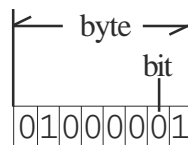
- * To process data, a computer must bring the data into its *memory*.
 - Computer memory is a working area where programs can place information they are using.
 - When it starts, each program gets its own area of memory, which other programs can't use.
 - A program then allocates specific parts of its own memory to store specific pieces of data.
 - When a program finishes running, it releases all of its memory so that other programs can use it when they are run.
- * The smallest piece of information in a computer is a *bit*.
 - A bit is like a light switch: it's either on or off (1 or 0).
- * The smallest unit of memory a program works with is a group of eight bits — a *byte*.
 - A typical program needs millions of bytes of memory to hold all of the information it uses while it's running.

Computers and Memory

One of the primary components of a computer system is its *main memory*. This is where the computer puts your program to execute, and where your program puts the data it is to process. You may also hear main memory called *RAM* or *core*.

Memory size is measured in bytes. A byte is composed of 8 *binary digits*, called bits. Each of these bits can represent either a 0 or a 1. No matter what the computer does, it reduces everything — programs, data, everything — into patterns of ones and zeros. For example, in a single byte, there are 256 different patterns of 0's and 1's possible (00000000, 00000001, 00000010, 00000011, and so on, up to 11111111); the computer might use these different patterns to represent different letters or punctuation marks.

Or, with four 8-bit bytes joined together (for a total of 32 bits), there are 4,294,967,296 different patterns of 0's and 1's possible (00000000000000000000000000000000, 00000000000000000000000000000001, and so on, up to the pattern with all bits set: 11111111111111111111111111111111). The computer might use these different bit patterns to represent all of the different positive integers from 0 to 4294967295. Or it might use the first bit to indicate whether the number is positive or negative, and thus represent all the integers (including 0) from -2147483646 to 2147483647.

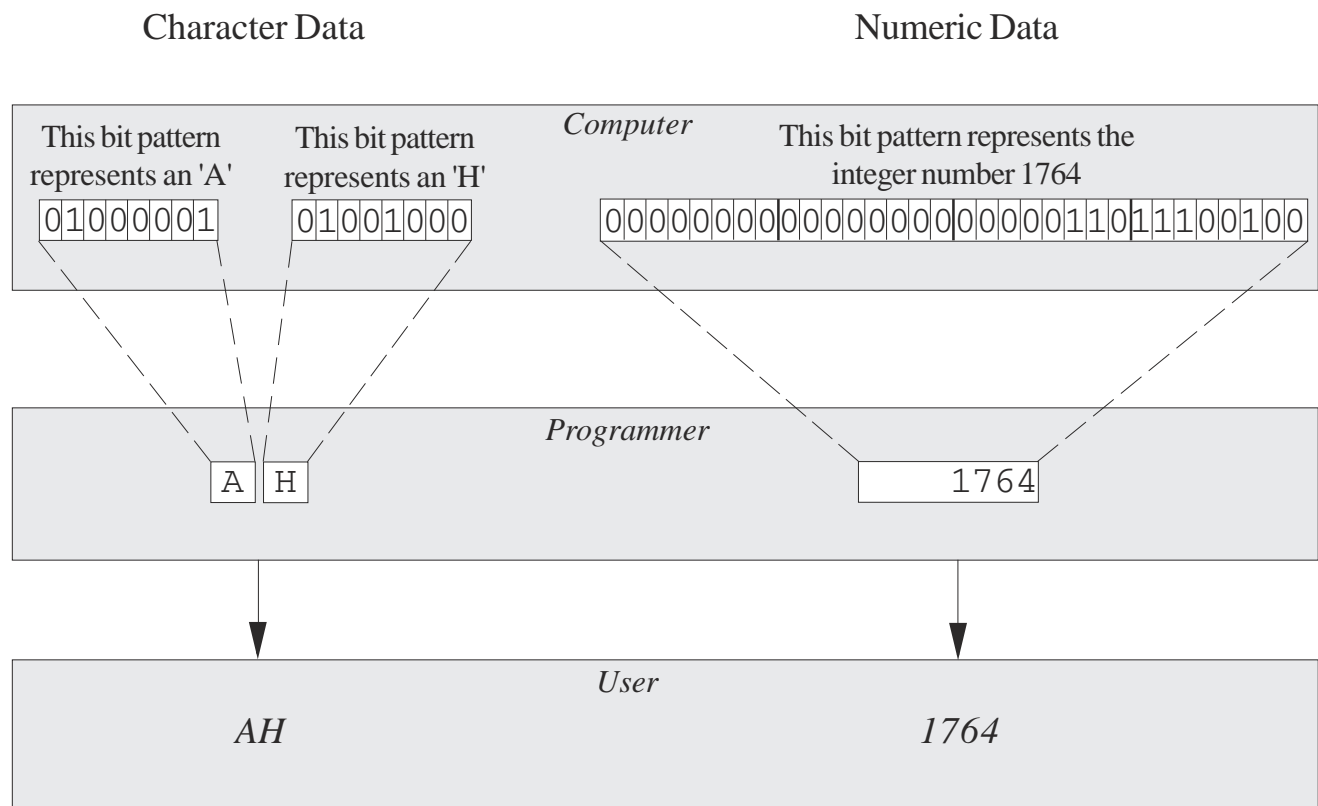


| Computer | | |
|------------------------------------|------------------------------------|---|
| This bit pattern represents an 'A' | This bit pattern represents an 'H' | This bit pattern represents the integer number 1764 |
| 01000001 | 01001000 | 000000000000000000000000000011011100100 |

DATA CAN BE OF DIFFERENT TYPES

- * In a program, each piece of data is of one specific data type.
- * Programmers usually work with data of three basic types:
 - *Characters* represent a single letter, numeral, or punctuation mark: *H, k, 9, (, **.
 - *Strings* are a collection of characters with a length attribute.
 - *Integer* data are numbers that do not have a decimal point; whole numbers: *42, 0, -123, 1928374*.
 - *Floating point* data are numbers that do have a decimal point; real numbers: *0.0, 3.14, -15.9997*.
 - *Double-precision floating points* are floating points that are more precise, taking up more space in memory.
- * Each of these data types takes up a fixed amount of memory on a particular computer system.
 - Characters typically require 1 or 2 bytes.
 - Strings can require any number of bytes. The length is determined when a string variable is created and initialized with data.
 - Integers typically require 4 bytes .
 - Floating points typically require 4 bytes.
 - Double-precision floating points typically require 8 bytes.
- * Each data type uses memory in a different way to represent its value.

No matter what the computer does, it reduces everything — programs, data, everything — into ones and zeros. However, we don't work at that level. In our program, we refer to types of data that are meaningful and useful to us; the computer worries about what pattern of bits to use to represent them. For example, for character data, there are 256 different patterns of 0's and 1's that can be represented in 8 bits; each of these patterns represents a different letter or punctuation mark. For numeric data, several 8-bit bytes can be joined together to form larger patterns of bits, representing a wide range of numbers.



NAMED DATA: VARIABLES

- ✧ A *variable* is data that has a name and represents a changing value — the data in the variable may vary as the program runs.
- ✧ Variables are used in mathematics.
 - An equation stating that $A^2 + B^2 = C^2$ suggests that there are many different values for A, B, and C for which the equation is true.
- ✧ In most programming languages, you must include statements in your program to define variables; for example, in Java:
 - The keyword `char` defines character variables:

```
char middleInitial;
```

 - The `String` class name defines string variables.
 - The keyword `int` defines integer variables:

```
int k, factor, day;
```
 - The keyword `float` defines floating point variables:

```
float averageAge;
```

 - The keyword `double` defines double-precision floating point variables.

```
double diameter, circumference, radius;
```
- ✧ Always use variable names that will help you remember what you use the variable for.

In Java (and many other programming languages) you must declare a variable before you use it. You declare a variable with a statement that specifies the type of data the variable will contain. Declaring a variable reserves enough memory for one value of the variable's type.

```
int i;
```

```
char c;
```



Each variable must have a distinct name; the name cannot be one of the language's reserved **keywords**. Keywords are part of the language's own syntax, and the compiler would become very confused if you used one of them as the name of a variable.

For example, here are some Java keywords we have seen so far (there are many others):

```
char    double    else    float    for    if    int    void    while
```

Java variable names must start with a letter and can be made up of any sequence of letters, digits, and the underscore, `_`. Java is a **case-sensitive** language; so, in a single program, the following variable names would all refer to different variables:

```
number    Number    NUMBER    NuMbEr
```

You should create variable names that are descriptive of the data. So, for example, if a variable is to hold the total value of something you will calculate, you might name the variable something like `total_value` or `TotalValue`. If you name the variable `flarp`, there is absolutely no way to tell, without reading every line of your program, what the variable is used for.

LITERAL DATA

- * *Literals* are used in mathematics and elsewhere.
 - If someone tells you to add 2 to a number, 2 is a literal value — it has no other name and it represents a fixed amount.
 - If someone tells you to put the letter *s* at the end of a word, *s* is a literal value — it has no name and it represents one particular letter.
 - An equation stating that $3^2 + 4^2 = 5^2$ says only that this is true for these three particular numbers.
- * You can put literal values in your program representing data of any one of the previously mentioned data types.
 - A character literal is always enclosed in a pair of single quotes: 'H', 'k', '9', '(', '*'.
 - A String literal is always enclosed in a pair of double quotes: "Hello, world."
 - Integer literals are numbers that do not contain a decimal point: 42, 0, -123, 1928374.
 - Double-precision floating point literals are numbers that *do* contain a decimal point: 42.0, 0.0, 3.14, -15.9997.
 - Floating point literals look just like double-precision floating point literals except they have an F on the end (upper or lower case): 42.0F, 0.0f, 3.14f, -15.9997F.
 - This uses less memory, when less accuracy is OK.

A literal is just that: a literal piece of data whose value cannot change. You will use many literals that have a variety of different data types in your programs. You may compare a data item to a literal to see if it is in range. Or you may use a literal to tell the program how many times you want a particular task accomplished. Or you may assign a literal to another data item to initialize it with a known value.

There is one type of literal data in Java that has an object type instead of a data type. In your programs, as we have already seen, you can express a string object literal, as a list of characters in double quotes.

Please Note: A *character* literal appears between a pair of single quotes ('), and a *string* literal appears between a pair of double quotes (") — there is a difference!

ASSIGNMENT

- * *Assignment* is the action of putting a value in a variable.

```
i = i + 10;  
c1 = initial;  
pi = 3.14f;
```

- * An assignment has a *left-hand side* (or lhs) and a *right-hand side* (rhs).

```
i = i + 10;
```

- The expression on the right-hand side is evaluated, and then the result is placed in the variable on the left-hand side.

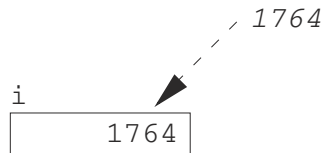
- The left-hand side must be a variable.

- * Most languages allow you to give a variable an initial value in the same statement that declares the variable.

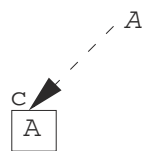
```
int i = 42;  
char middleInitial = 'A';  
float pi = 3.1415926F;
```

Assignment lets you take any literal, or the result of an expression (more on expressions later), or data from any number of other sources, and put it into a variable. Assignments can appear at many different locations in a program and are used frequently.

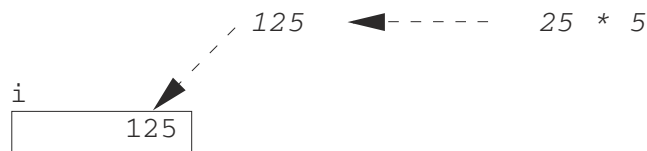
i = 1764;



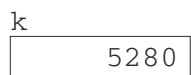
c = 'A';



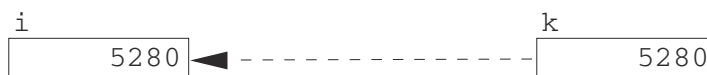
i = 25 * 5;



int k = 5280;



i = k;



PRINTING VARIABLES

- * When you use a variable in an output statement, the value stored in the variable is printed.

PrintVars.java

```
package examples;

public class PrintVars {
    public static void main(String[] args) {
        int i_five, i_seventeen;
        char c_five, c_one, c_seven;
        float f_five, f_seventeen;
        double d_five, d_seventeen;

        i_five = 5;
        c_five = '5';
        f_five = 5.001F;
        d_five = 5.00001;

        System.out.println("An int: " + i_five);
        System.out.println("A char: " + c_five);
        System.out.println("A float: " + f_five);
        System.out.println("A double: " + d_five + "\n");

        i_seventeen = 17;
        c_one = '1';
        c_seven = '7';
        f_seventeen = 17.001F;
        d_seventeen = 17.00001;

        System.out.println("An int: " + i_seventeen);
        System.out.println("Two chars: " + c_one + " and " +
            c_seven);
        System.out.println("A float: " + f_seventeen);
        System.out.println("A double: " + d_seventeen);
    }
}
```

Try It: Compile and run *PrintVars.java*, shown on the facing page. If you're feeling adventurous, try changing some of the values or data types.

REVIEW QUESTIONS

1. What is the primary purpose of a computer program?
2. Where does a program keep the information it is processing?
3. How big is a byte?
4. How does a computer represent all information internally?
5. What are the three basic data types in computer programming?
6. Explain what constant, or literal, data is.
7. What is a variable used for?
8. Explain and describe the purpose of assignment.

LABS

- ① Write a program, similar to the *PrintVars.java* program at the end of this chapter, that defines two variables: one named `age` (for your age) and one named `mi` (for your middle initial). Initialize the variables appropriately. Have the program print your first name as a string literal, your middle initial from the variable, your last name as a string literal, the string `" : Age: "`, and your age from the variable.
(Solution: *AgeInitial.java*)
- ② Write a program that defines four variables named `score1`, `score2`, `score3`, and `score4` — one for each of a student's four test scores. Assign appropriate values to the variables. Define another variable named `average` in the program. Have the program calculate the average of the test scores and assign the result to the variable:

```
average = (score1 + score2 + score3 + score4) / 4;
```

Print out the four test scores and the average (be creative).

(Solution: *StudentScores.java*)

CHAPTER 5 - SCREEN OUTPUT AND KEYBOARD INPUT

OBJECTIVES

- * Write output from your program to your terminal screen.
- * Explain the purpose of escape characters.
- * Do some simple formatting in your program's output.
- * Read input from the terminal keyboard into your program.
- * Explain why and how you prompt your program's user for input.

WRITING TO THE SCREEN

- * The simplest way a program can communicate with its user is to print data on the screen.

- * The computer screen is called the *standard output* device ("standard out").

```
System.out.println("You were " + age + " in 1990.");
```

- * Error messages are usually sent to the *standard error* device.

```
System.err.println("Your age cannot be negative!");
```

- The standard error device is also your computer screen.
- Users running your program can put standard error messages in a separate log file if they choose.

- * Standard out is where your normal program output should be written, and standard error is where any error or warning messages should be printed.

- * Output statements are very programming language dependent; each language has its own syntax and mechanisms to perform output operations.

```
/* In C: */  
printf("You were %d in 1990.\n", age);
```

```
// In C++:  
cout << "You were " << age << " in 1990." << endl;
```

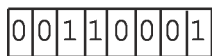
```
# In Perl:  
print "You were $age in 1990.\n";
```

```
int i = 17;
```

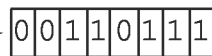
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |

```
System.out.println(i);
```

Bit pattern for the numeral '1'



Bit pattern for the numeral '7'



17

CHARACTERS THAT HAVE SPECIAL MEANING

✴ There are several *special characters* that your program can output for some "special effects."

✴ These special characters, represented by escape sequences, can be used in character or string variables or as literals.

`\n` A "newline" character causes a line-feed (i.e., a new line) to be generated.

`\t` A "tab" character causes a tab to be generated. This tab will cause the cursor to move to the next "tabstop," which usually occurs every eight character positions on the screen.

`\b` A "backspace" character causes the cursor to back up.

`\ "` A literal double quote character — used in string literals.

`\\` A literal backslash character — used in string literals.

✴ Each of these two-character escape sequences really represents only one character of output.

➤ Escape sequences are used for characters that can't be represented directly in your program.

Computer and software designers follow many agreed-upon conventions so that their products will all operate together. The **ASCII character set** is a standard defining, for character data, which bit-patterns represent which letters, numerals, and punctuation marks. The bit patterns are all numbered, from 0 to 127:

| | | | |
|------|--|---------------------------------------|------------|
| 0: | | <i>nul</i> control character | ' \0 ' |
| 1: | | <i>soh</i> control character (unused) | |
| ... | | | |
| 7: | | <i>bel</i> (bell) | ' \u0007 ' |
| 8: | | <i>bs</i> (back space) | ' \b ' |
| 9: | | <i>ht</i> (horizontal tab) | ' \t ' |
| 10: | | <i>lf</i> (line feed) | ' \n ' |
| ... | | | |
| 56: | | Digit Eight | ' 8 ' |
| 57: | | Digit Nine | ' 9 ' |
| 58: | | Colon | ' : ' |
| ... | | | |
| 65: | | Capital Letter A | ' A ' |
| 66: | | Capital Letter B | ' B ' |
| ... | | | |
| 97: | | Small Letter A | ' a ' |
| 98: | | Small Letter B | ' b ' |
| ... | | | |
| 126: | | Tilde | ' ~ ' |
| 127: | | <i>del</i> (delete) control character | |

(Don't worry, you don't need to memorize these. But you will encounter character codes in any programming language you study, and they'll become more and more familiar.) Sometimes you need to use an ASCII character in your program, but there is no "printable" character that you can type. The first 32 characters in the ASCII character set are, in fact, exactly like this. These special characters are used to control peripheral devices such as teletypewriters (the ASCII standard is that old), disks, tapes, printers, and the terminal you use to type in your program's source code.

Investigate:

Find your "Hello, world!" program and modify the output statement. Try replacing the " , " in the text being printed with " , \t ". See if you can tell if anything different happens when you compile and run it. Try placing a '\u0007' in the text being printed, and see if you can tell if anything different happens when you compile and run it.

SOME SIMPLE FORMATTING

- * If you want data printed to the screen to line up in any special way — that is, if you want to *format* your data — you must put the blanks, tabs, and newlines out to the screen yourself.

- If you don't put a space between two data items, they will be printed together with no space between them.

```
System.out.print("Output: " + i + j);
```

- If you don't put a newline out at the end of an output statement, everything will be printed on a single line.

```
System.out.print(a);  
System.out.print(b);
```

- * You are responsible for formatting the data that you display on the screen in a way that is useful, looks good, and makes sense:

```
System.out.println(i + " " + j);  
System.out.println(a + "\n" + b);
```

- * Some programming languages provide special features and commands that will help you control the format of your output.

Formatting.java

```
package examples;

public class Formatting {
    public static void main(String[] args) {
        java.util.Scanner scanner = new java.util.Scanner(System.in);

        int var1 = 0, var2 = 0;

        System.out.print("Please enter a number: ");
        var1 = scanner.nextInt();
        System.out.print("Please enter another number: ");
        var2 = scanner.nextInt();

        // No formatting at all:
        System.out.println("No format: " + var1 + var2 +
            (var1 + var2) + "\n");

        // Some simple formatting:
        System.out.print("The sum of " + var1 + " and " + var2);
        System.out.println(" is " + (var1 + var2) + "\n");

        // A different way:
        System.out.println("\t" + var1 + "\n+\t" + var2);
        System.out.println("-----\n\t" + (var1 + var2) + "\n");

        // Maybe a little too much formatting?
        System.out.println("The Amazing SUM Program !!!!!!!!!!!!!\n");
        System.out.println("*****");
        System.out.println("***\tThe sum of the number\t" + var1);
        System.out.println("***\t added to the number\t" + var2);
        System.out.println("***\t results in...");
        System.out.println("*****");
        System.out.println("***\t\t Ta DA!\t" + (var1 + var2));
        System.out.println("*****");

        scanner.close();
    }
}
```

Try It: Compile and run *Formatting.java*. Enter two integers to be added.

READING FROM THE KEYBOARD

- * Input from the keyboard is called *standard input*, or "standard in."
- * Use data read from standard in immediately (or place it in a variable).
- * Input statements are programming language dependent; each language has its own statements and mechanisms to perform input operations.

```
// In Java:  
java.util.Scanner scanner =  
    new java.util.Scanner(System.in);  
age = scanner.nextInt();  
scanner.close();
```

```
/* In C: */  
scanf("%d", &age);
```

```
# In Perl:  
$age = <STDIN>;
```

- * For keyboard input, some programming languages will convert the user's keystrokes into the binary data format needed by the computer.
 - In Java, you must use different method calls.

```
String s = scanner.next();  
short s = scanner.nextShort();  
int i = scanner.nextInt();  
long l = scanner.nextLong();  
boolean b = scanner.nextBoolean();  
float f = scanner.nextFloat();  
double d = scanner.nextDouble();
```

Input.java

```
package examples;

public class Input {
    public static void main(String[] args) {
        java.util.Scanner scanner = new java.util.Scanner(System.in);

        int age = 0;
        int year = 0;

        System.out.print("Please enter your age in years: ");
        age = scanner.nextInt();
        System.out.print("Please enter the current year: ");
        year = scanner.nextInt();

        scanner.close();

        if (year < 2000) {
            System.out.print("You'll be ");
        }
        else {
            System.out.print("You were ");
        }

        System.out.println((2000 - year + age) + " years old in the "
            + "year 2000.");
    }
}
```

Try It: This example reads two integer numbers from the keyboard and converts them to ints.

PROMPTING AND VALIDATING

- * You will find it necessary, from time to time, to ask the person using your program for some input.

- This is referred to as *prompting* the user.

- * There are three steps required:

1. Define a variable to store the data in:

```
int theGrade;
```

2. Print a message to standard out that tells the user what you want:

```
System.out.print("What is the next student's grade? ");
```

3. Read the data the user types on the keyboard into your program's variables:

```
theGrade = scanner.nextInt();
```

- * Prompting will help the user to enter the correct data, but your program will usually need to *validate* that what was entered was correct.

- Your program may need a number, but the user might type a letter.
- Validating the data that is input is an important part of any program.

InputValid.java

```
package examples;

public class InputValid {
    public static void main(String[] args) {
        java.util.Scanner scanner = new java.util.Scanner(System.in);

        int age = 0;
        int year = 0;

        System.out.print("Please enter your age in years: ");
        age = scanner.nextInt();
        System.out.print("Please enter the current year: ");
        year = scanner.nextInt();

        scanner.close();

        if (age <= 0) {
            System.out.println("Age can't be less than zero!");
            return;
        }

        if (year < 100) {
            System.out.println("Assuming you meant " + (year + 1900));
            year = year + 1900;
        }
        else if (year < 1990) {
            System.out.println("Have we gone back in time?");
            return;
        }

        if (year < 2000)
            System.out.print("You'll be ");
        else
            System.out.print("You were ");

        System.out.println((2000 - year + age) + " years old in the "
            + "year 2000.");
    }
}
```

Try It: Try running this program with a negative age or a year less than 100.

EXAMPLE 5 - FORMATTING OUTPUT DATA

PromptFormat.java

```
package examples;

public class PromptFormat {
    public static void main(String[] args) {
        java.util.Scanner scanner = new java.util.Scanner(System.in);

        // Define some variables to contain student data
        int stu1Num = 0, stu2Num = 0, stu3Num = 0;
        float stu1Avg = 0.0F, stu2Avg = 0.0F, stu3Avg = 0.0F;
        String stu1Class = "", stu2Class = "", stu3Class = "";

        // Prompt for, and read, data for three students
        System.out.print("Please enter Student #1's student number"
            + " (9999): ");
        stu1Num = scanner.nextInt();
        System.out.print("Please enter Student #1's grade average"
            + " (99.99): ");
        stu1Avg = scanner.nextFloat();
        System.out.print("        Please enter Student #1's class"
            + " (F/P/J/S): ");
        stu1Class = scanner.next();

        System.out.print("Please enter Student #2's student number"
            + " (9999): ");
        stu2Num = scanner.nextInt();
        System.out.print("Please enter Student #2's grade average"
            + " (99.99): ");
        stu2Avg = scanner.nextFloat();
        System.out.print("        Please enter Student #2's class"
            + " (F/P/J/S): ");
        stu2Class = scanner.next();

        System.out.print("Please enter Student #3's student number"
            + " (9999): ");
        stu3Num = scanner.nextInt();
        System.out.print("Please enter Student #3's grade average"
            + " (99.99): ");
        stu3Avg = scanner.nextFloat();
```

Try It: Compile and run *PromptFormat.java*. This example may also be helpful as a starting point for some of the labs at the end of the chapter.

```
System.out.print("      Please enter Student #3's class"
    + " (F/P/J/S): ");
stu3Class = scanner.next();

// Format the data and print the student report
System.out.println("Student Student Student");
System.out.println(" Number Average Class");
System.out.println("-----");

System.out.println("      " + stu1Num + "\t" + stu1Avg + "\t"
    + stu1Class);
System.out.println("      " + stu2Num + "      " + stu2Avg + "      "
    + stu2Class);
System.out.println("      " + stu3Num + "      " + stu3Avg + "      "
    + stu3Class);

System.out.print("-----");

scanner.close();
}
}
```

REVIEW QUESTIONS

1. Where should your program's output, other than error messages, be sent?
2. Where should your program's error messages be sent?
3. What is the standard output device?
4. What is the purpose of an escape sequence?
5. What is meant by the phrase "formatting your program's output"?
6. From where does your program read input the user types?
7. List the steps needed to prompt your program's user for input.

LABS

- ① Write a program that prompts the user for a list of five integers. Place each of the numbers in a separate variable:

- a. Print each of the numbers separated by a space.
- b. Print each of the numbers separated by a tab.
- c. Print each of the numbers separated by a newline.

(Solution: *FiveIntegers.java*)

- ② Write a program that prompts the user for a String department code, a floating-point salary, and an integer employee id. Put each of the items in a separate variable:

- a. Print each of the items and a string literal label separated by a space.
- b. Print each of the items and a string literal label separated by a tab.
- c. Print each of the items and a string literal label separated by a newline.

(Solution: *ThreeVariables.java*)

- ③ Write a program that prints out your initials as a large banner, with each large letter made up of the same letter; here's what we mean:

```

W      W      A      P P P P P
W W W      A A      P      P
W W W      A  A      P      P
W W W      A      A  P P P P P
W W W      A A A A A A P
W W W      A      A  P
  WW WW      A      A  P
  
```

(Solution: *BigLetters.java*)

- ④ Write a program that uses your initials to draw a picture of a house:

```

          AAAA
        P P P P P P P P A A
      P P P P P P P P A A
    P P P P P P P P A A
  P P P P P P P P P P P P
PW  WWWWWWWWWWWWWWWWWWWWWWW
W   W W AAAAA W W W W
W   WWWWW A A WWWWW W
W       A AA W
W       A A W
WWWWWWWWWWWWWWWWWWWWWWWWWWWW

```

(Solution: *LetterHouse.java*)

CHAPTER 6 - EXPRESSIONS

OBJECTIVES

- * Explain how expressions get the work done in a program.
- * Write expressions that get evaluated into a result.
- * Use arithmetic expressions to calculate values.
- * Write programs using several arithmetic operators.
- * Use relational expressions to test values in your programs.
- * Write programs that use *and* and *or* logic.
- * Explain precedence and associativity of operators.

EXPRESSIONS: WHERE THE WORK GETS DONE

- * An *expression* is a combination of literals, variables, and *operators*.
- * An *operator* is a symbol in your source code that specifies an operation to be performed on some data (its *operands*).
 - The expressions you write tell the computer how to process your data — the actual work your program is designed to perform.
- * There are several types of expressions:

- *Assignment expressions* — assign data to a variable.

```
age = 35
```

- *Arithmetic expressions* — perform arithmetic calculations.

```
5 + 3
```

- *Relational expressions* — come up with a true/false answer to a question.

```
i < 10
```

- *Logical expressions* — combine multiple true/false answers into one.

```
i < 10 && j > 5
```

CylinderVolume.java

```
package examples;

public class CylinderVolume {
    public static void main(String[] args) {
        java.util.Scanner scanner = new java.util.Scanner(System.in);

        float volume = 0.0F, radius = 0.0F, height = 0.0F;

        System.out.print("Enter the radius of the cylinder: ");
        radius = scanner.nextFloat();

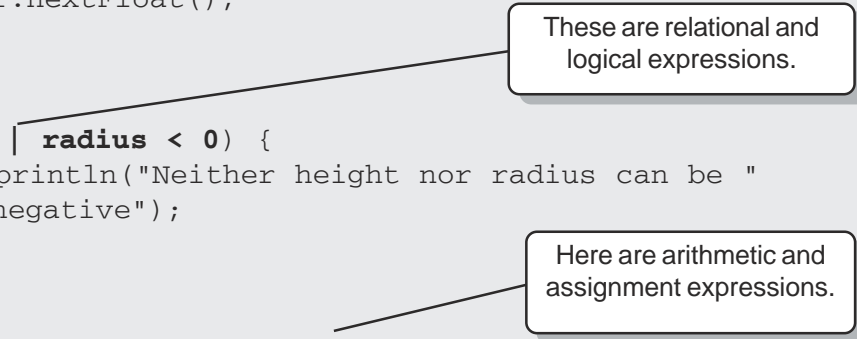
        System.out.print("Enter the height of the cylinder: ");
        height = scanner.nextFloat();

        scanner.close();

        if (height < 0 || radius < 0) {
            System.err.println("Neither height nor radius can be "
                               + "negative");
            return;
        }

        volume = 3.14159265F * radius * radius * height;

        System.out.print("A cylinder of height " + height);
        System.out.print(" and radius " + radius);
        System.out.println(" has a volume of " + volume);
    }
}
```



Try It: Run *CylinderVolume.java* to use an expression to calculate the volume of a cylinder.

EXPRESSION EVALUATION: THE RESULT

- ✱ Evaluation of an expression applies the specified operations to the operands, reducing the entire expression down to a single value.
 - Arithmetic operators perform calculations which result in a numeric value.
 - Relational operators perform comparison operations which result in a true or false (boolean) value.
 - Logical operators perform and/or logic which also result in a true or false (boolean) value.
- ✱ No matter what type of expression is being evaluated, we are looking for a single value to come out of the processing — a result:
 - Evaluating $3 + 5$ results in 8.
 - Evaluating $3 < 5$ results in true.
 - Evaluating $3 < 5$ and $10 < 7$ results in false.
- ✱ The value of an expression isn't determined until the expression is evaluated at runtime.

Average4.java

```
package examples;

public class Average4 {
    public static void main(String[] args) {
        java.util.Scanner scanner = new java.util.Scanner(System.in);

        float n1 = 0.0F, n2 = 0.0F, n3 = 0.0F, n4 = 0.0F;
        float average = 0.0F;

        System.out.println("Enter any four numbers: ");
        System.out.print("          #1: ");
        n1 = scanner.nextFloat();
        System.out.print("          #2: ");
        n2 = scanner.nextFloat();
        System.out.print("          #3: ");
        n3 = scanner.nextFloat();
        System.out.print("          #4: ");
        n4 = scanner.nextFloat();

        // Calculate the result and store it in the variable average:
        average = (n1 + n2 + n3 + n4) / 4;

        System.out.print("The average of (" + n1 + ", " + n2 + ", ");
        System.out.print(n3 + ", " + n4 + ") is: ");
        System.out.println(average);

        scanner.close();
    }
}
```

Try It: This example shows that when you add floats, then divide by an int, you put the result in a float.

ARITHMETIC EXPRESSIONS

* Arithmetic expressions translate mathematical formulas into computer instructions.

* Most arithmetic operators take two operands (*binary* operators).

➤ Addition and subtraction of two numbers:

```
length + 6.13  
5 - radius
```

➤ Multiplication and division of two numbers:

```
width * height  
total / 2
```

➤ Modulus (remainder) of two integers:

```
15 % 4  
year % 100
```

* A few operators work with just one operand (*unary* operators):

➤ To make a data item negative, precede it with the – operator:

```
-score
```

* Many subexpressions can be combined to form a final result:

```
volume = 4.0 / 3.0 * 3.14159 * radius * radius * radius;
```

SphereVolume.java

```

package examples;

public class SphereVolume {
    public static void main(String[] args) {
        java.util.Scanner scanner = new java.util.Scanner(System.in);

        float volume = 0.0F, radius = 0.0F;

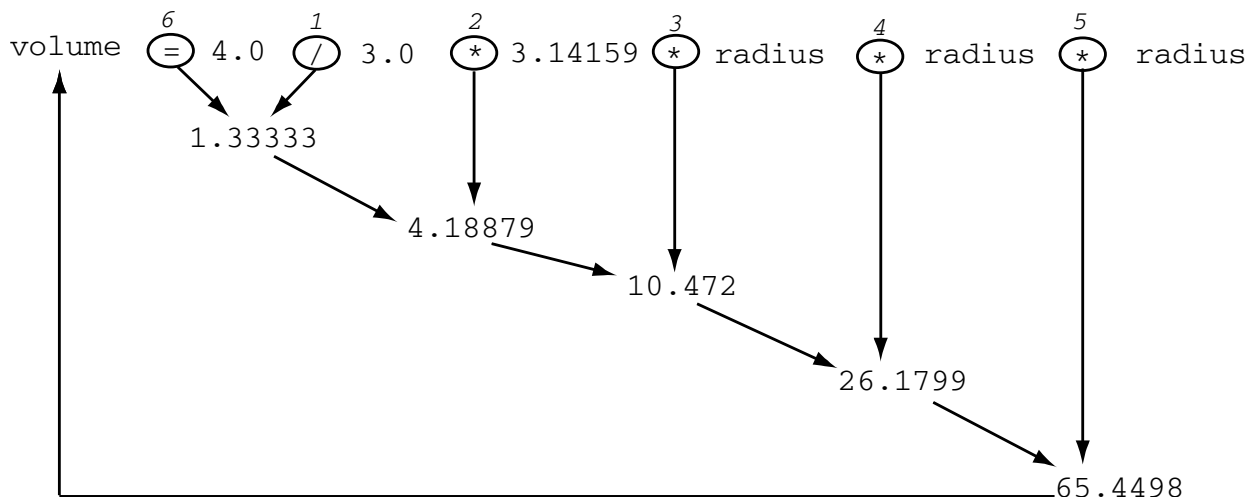
        System.out.print("Enter the radius of the sphere: ");
        radius = scanner.nextFloat();

        volume = 4.0F / 3.0F * 3.14159265F * radius * radius * radius;
        System.out.print("A sphere of radius " + radius);
        System.out.println(" has a volume of " + volume);

        scanner.close();
    }
}

```

Expressions often have many operators. Each operator is evaluated in turn, with the result used as an operand for the next operator. For example, if radius is 2.5:



RELATIONAL EXPRESSIONS

- * Relational expressions are used to test a relationship between two data items to ask: "Does one data item compare to another in a particular way?"

- These expressions result in a *boolean* value: a true or false.

- * Relational expressions are constructed by combining a relational operator with two operands:

- The "less than" and "less than or equal to" operators:

```
hoursWorked < 40  
age <= 12
```

- The "greater than" and "greater than or equal" operators:

```
income > 55000  
numPlayers >= 3
```

- The "equal to" and "not equal to" operators:

```
menuItem == choice  
choice != 'q'
```

- * Relational expressions are used to test values.

Note:

In Java and related languages (C, C#, C++, Perl, etc.), a doubled equals sign, `==`, is the relational "equal to" operator, whereas a single equals sign, `=`, is the assignment operator. There is a difference, and mixing them up can cause both compiler errors and runtime errors (depending on where you made the mistake).

The big question, of course, is where do you use relational expressions? They don't appear to be very useful in assignments or output statements.

WHERE ARE RELATIONAL EXPRESSIONS USED?

- * Relational expressions are used where we need to make some kind of true/false decision.
- * These types of decisions need to be made in conditional (`if`) statements and in loops:

- `if` statements let you decide whether one group of statements or another will be executed, based on the result of a relational expression.

```
if (age > 18)
// They can register to vote.
```

- A conditional loop executes a group of statements repeatedly, as long as the result of evaluation of an expression is true.

```
while (movesRemaining > 0)
// Allow another move in the game.
```

- * Conditionals and loop statements will be covered in more detail soon.

AND? ... OR? ...

- * Logical expressions are used to combine multiple relational expressions in `if` statements and `while` loops.
- * A logical expression compares two true/false values and results in a single true/false value.

- If both operands of a *logical and* (`&&`) are true, the result is true; otherwise the result is false.

```
if ( a > 0 && a < 10 )  
    System.out.println("a is between 0 and 10");
```

```
if ( sex == 'M' && age > 65 )  
    System.err.println("See special medication info");
```

- If either operand of a *logical or* (`||`) is true, the result is true; otherwise the result is false.

```
if ( answer == 'y' || answer == 'Y' )  
    // They meant "Yes"...
```

If you were to think of the logical operators as a sort of strange arithmetic operator, the following "calculations" can be performed with the indicated result:

| | | | |
|---|--|--|--|
| $\frac{\begin{array}{c} \textit{false} \\ \&\& \textit{false} \end{array}}{\textit{false}}$ | $\frac{\begin{array}{c} \textit{false} \\ \&\& \textit{true} \end{array}}{\textit{false}}$ | $\frac{\begin{array}{c} \textit{true} \\ \&\& \textit{false} \end{array}}{\textit{false}}$ | $\frac{\begin{array}{c} \textit{true} \\ \&\& \textit{true} \end{array}}{\textit{true}}$ |
| $\frac{\begin{array}{c} \textit{false} \\ \textit{false} \end{array}}{\textit{false}}$ | $\frac{\begin{array}{c} \textit{false} \\ \textit{true} \end{array}}{\textit{true}}$ | $\frac{\begin{array}{c} \textit{true} \\ \textit{false} \end{array}}{\textit{true}}$ | $\frac{\begin{array}{c} \textit{true} \\ \textit{true} \end{array}}{\textit{true}}$ |

If the variable *a* has a value of 5:

a > 0 && *a* < 10 Evaluates as: *true* && *true*. Therefore, the result is *true*.

If the variable *a* has a value of 17:

a > 0 && *a* < 10 Evaluates as: *true* && *false*. Therefore, the result is *false*.

PRECEDENCE AND ASSOCIATIVITY

- * *Associativity* determines in which order a given operator looks for its operands.

```
b = 4;    // The = looks at the right-hand side for a
          // value, and assigns it to the variable on the
          // left-hand side.
```

- * *Precedence* determines which operator gets evaluated first when an expression contains more than one operator.

```
c = b + 7; // The + has higher precedence than = so it's
          // evaluated first, and its result is the
          // right-hand operand of the =.
```

- * Associativity also determines which operator gets evaluated first when an expression contains more than one operator with the same precedence.

```
a = b + c + 10; // +'s have the same precedence,
                // associativity is left to right. Take
                // b and add it to c first, then add the
                // result to 10.
```

- * If you want to change the order of evaluation in an expression to something other than the default, use parentheses to group subexpressions.

```
a = b + (c + 10);    // Result is the same as in the
                     // previous expression.
a = b * (c + 10);    // c + 10 is done first
```

Precedence is a ranking of which operator is more important. Operators with a higher precedence get evaluated first, while lower-precedence operators get evaluated later.

If two operators in the same expression have the same precedence, then *associativity* can determine which gets evaluated first. Associativity specifies either a right-to-left or a left-to-right evaluation. Most operators associate left-to-right. Notable exceptions are the assignment operator, which evaluates the expression on its right before assigning it to the variable on its left, and the unary negative sign, which applies to the value on its right and doesn't do anything with its left-hand side:

```
x = y * -3
```

Programmers use **precedence tables** to figure out how the computer will evaluate the expressions they write. This partial table of Java operators shows the operator, its precedence, and its associativity.

| Precedence | Operator | Meaning | Associativity |
|------------|-----------|-------------------------------|---------------|
| Highest | + - | positive & negative signs | right to left |
| | * / % | multiplication & division | left to right |
| | + - | addition & subtraction | left to right |
| | < <= > >= | less than, greater than, etc. | left to right |
| | == != | equal & not equal | left to right |
| | && | logical and | left to right |
| | | logical or | left to right |
| Lowest | = | assignment | right to left |

Using the fact that && has higher precedence than ||:

```
b < 42 || c > 50 && c < 100
```

the computer will evaluate the expression *as if* it looked like this:

```
( b < 42 ) || ( c > 50 && c < 100 )
```

So, if *b* is 10 and *c* is 25, is the final result *true* or *false*? Try to figure it out, and see if your instructor agrees!

EXAMPLE 6 - CALCULATING MILES PER GALLON

MilesPerGallon.java

```
package examples;

public class MilesPerGallon {
    public static void main(String[] args) {
        java.util.Scanner scanner = new java.util.Scanner(System.in);

        float milesTraveled = 0.0F;
        float gallonsUsed = 0.0F;
        float milesPerGallon = 0.0F;

        System.out.print("Please enter the number of miles"
            + " traveled: ");
        milesTraveled = scanner.nextFloat();
        System.out.print("Please enter the number of gallons"
            + " used: ");
        gallonsUsed = scanner.nextFloat();

        milesPerGallon = milesTraveled / gallonsUsed;

        System.out.print("You traveled " + milesPerGallon);
        System.out.println(" miles for every gallon of fuel"
            + " used.");

        scanner.close();
    }
}
```


REVIEW QUESTIONS

1. What is an expression?
2. What is an operator?
3. How do expressions get work done in a program?
4. What is meant by the "result" of an expression?
5. Describe an arithmetic expression.
6. What is the purpose of an arithmetic expression?
7. Describe a relational expression.
8. What is the purpose of a relational expression?
9. What is a boolean value?
10. Explain the purpose of the *and* and the *or* operators.
11. What is precedence?
12. What is associativity?

LABS

- 1 Write a program that prompts the user to enter the radius of a circle, and reads the user's input into a floating-point variable. Have the program calculate the circumference of the circle, and print out both the radius and the circumference.

(Solution: *Circle1.java*)

- 2 Modify your solution to *Circle1.java* so that it also prints out the circle's area.

(Solution: *Circle2.java*)

- 3 Write a program that accepts a Fahrenheit temperature as input and converts it into a Celsius temperature, which it prints out.

(Solution: *FahrToCels.java*)

- 4 Accept two integers as input, say, *i* and *j*. Write the program so that it rounds off *i* to the next largest even multiple of the other integer *j*.

```
NextMultiple = i + j - i % j
```

To test your results, use the following test data: to round off 256 days to the next largest number of days that is evenly divisible by 7-day weeks. With *i* = 256 and *j* = 7, the next largest number of days that are evenly divisible into 7-day weeks is 259.

(Solution: *NextLargestMultiple.java*)

- 5 Write a program that accepts the amount of a restaurant check and calculates the tip amount for a 10%, 15%, and 20% tip.

(Solution: *CalculateTip.java*)

- 6 Write a program that reads two integers (a dividend and a divisor) from the user into variables and divides the first integer by the second (with a statement to print the result). Test the program several times with different values. What happens when the first number isn't evenly divisible by the second? What happens when you enter zero for the second number?

(Solution: *Divide.java*)

- 7 Write a program that accepts numeric values for the total daily rainfall for a week. Print the daily rainfall and calculate and print the total rainfall for the week, as well as the average rainfall for the week.
(Solution: *DailyRainfall.java*)
- 8 Write a program that accepts five numeric grades for classes that a student has taken. Print a report card and calculate a grade point average.
(Solution: *ReportCard.java*)
- 9 Write a program that prompts for x and y coordinates for the upper left and lower right corners of a rectangle, and calculates and prints the rectangle's area.
(Solution: *RectArea.java*)

Some useful formulas:

| | |
|----------------------------|--|
| π : | 3.141592653589793 |
| Circumference of a circle: | $\pi * \text{diameter}$ or $2 * \pi * \text{radius}$ |
| Area of a circle: | $\pi * \text{radius}^2$ |
| Celsius: | $5/9 (\text{Fahrenheit} - 32)$ |
| Fahrenheit: | $(9/5 \text{ Celsius}) + 32$ |

And remember, you aren't expected to complete *all* the exercises during the lab session!

CHAPTER 7 - DECISION MAKING

OBJECTIVES

- * Explain the term *sequential execution*.
- * Describe the purpose of conditional statements.
- * Use `if` statements to control the flow of program execution.
- * Describe the purpose of the `if` statement's `else` clause.
- * Create code blocks, and explain when and why they are used.
- * Create nested blocks.
- * Describe how a switch statement works.

SEQUENTIAL EXECUTION

- * We have seen several types of statements already:
 - Data definition statements define variables to be used in your program.
 - Input statements accept input from the keyboard.
 - Assignment statements assign values into variables.
 - Output statements send output to the screen.
- * *Sequential execution* refers to the execution of your program's statements in the sequence that they appear in your source code file — top-to-bottom, beginning-to-end.
 - In sequential execution, every statement is executed.
 - In sequential execution, every statement is executed once.
 - The program doesn't stop until there are no more statements to execute.

When you write a computer program, you typically start entering statements in the “main” part of the program. You enter the first statement you want executed first, the second statement second, and so on, until the last statement.

When your program starts, the computer executes the first statement in your program. When that statement finishes, it goes on to the next one, sequentially through your program statements. This continues to the last statement in your program, after which your program ends.

WHAT IS DECISION MAKING?

- * Sometimes, you will want the *flow of execution* to depend on conditions your program will determine while it's running.
 - If a certain condition is true, execute one set of statements; otherwise, execute a different set of statements.
- * Decision making statements in a computer program are called *conditionals*. Conditions occurring as the program runs determine the flow of execution from one part of the program to the next.
- * To make a decision, a computer program poses a question; the question typically asks how two data items are related to each other.
 - Less-than, greater-than, equal-to, etc., are all "questions" that can be posed by a program to result in non-sequential execution of statements.
- * A relational expression alone is not enough; we need a statement that can evaluate a relational expression, and then decide whether to execute a group of other statements based on the true/false result.
- * This is exactly what an `if` statement does.

Some everyday conditional statements:

"If it looks like rain out, I'll take my umbrella."

"If it looks like rain out, I'll take my umbrella; otherwise, I'll just take a jacket."

"If it looks like rain out, I'll take my umbrella,
but if it looks like sunshine out, I'll take my tanning lotion,
but if it looks like snow out, I'll go skiing!,
and if I can't tell what it looks like out, I'll just stay in."

In Java, these "decision trees" might look like this (we'll get to the syntax details in a moment):

```
if (rain)
    takeUmbrella();
```

```
if (rain)
    takeUmbrella();
else
    takeJacket();
```

```
if (rain)
    takeUmbrella();
else if (sunshine)
    takeTanningLotion();
else if (snow)
    goSkiing();
else
    stayHomeAndRead();
```

There are two types of non-sequential, decision-making statements that we will discuss in this chapter: the `if` statement and the `switch` (or `case`) statement.

SIMPLE DECISIONS: IF

- * An `if` statement evaluates a conditional expression and executes a specified statement if the condition is true.
 - The `if` is sometimes called a "control statement" — it controls the execution of another statement.
 - The conditional expression results in a true/false value; relational expressions are the usual type used in `if` statements.
- * `if` statements effectively have what is called a then clause.
 - The then clause is what is executed if the evaluation of the expression results in true.
 - If the relational expression results in false, then the statements in the then clause are skipped over, not executed: non-sequential execution.

```
if (answer != 42)
    System.out.println("Wrong answer ...");
```

- * When coding an `if` statement, using indentation helps show which statements the `if` controls.

The syntax of the Java `if` statement is simple:

```
if ( expression )  
    statement;
```

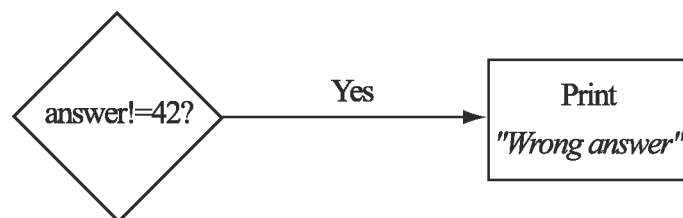
- `if` is the required keyword.
- `if` is followed by an expression, which must be enclosed by a pair of parentheses.
- The next statement after the `if` expression will be executed if the expression is true (the word "then" is not part of the actual syntax).

```
if ( answer != 42 )  
    System.out.println("Wrong answer ...");
```

Note there is no semicolon after the closing parenthesis, nor is there is a semicolon after the expression inside the parenthesis.

Notice how the "controlled statement" is indented from the `if`; this style of indentation is typical and shows that the `System.out.println` statement is controlled by the `if` (it shows the flow of execution). Such indentation, while not required by the language, is very important for program readability.

A **flow chart** can also help you make sure the logic of your program is correct:



Comparing Strings in Java

You should not use the `==` operator to compare Strings in Java. Instead, you should use a special method called `equals()` that is built into all Strings.

```
String fName = "John";  
String lName = "Doe";  
// if (fName == lName) // Wrong!  
if (fName.equals(lName)) { // Right!  
    System.out.println("Same first and last names");  
}
```

TWO-WAY DECISIONS: ELSE

- * if statements also have an else clause.
 - The then clause is executed if the expression results in true.
 - The else clause is executed if the expression results in false.

```
if (answer != 42)
    System.out.println("Wrong answer ...");
else
    System.out.println("You got the correct" +
        " answer!");
```

- * Only one of the two statements will execute.
- * Again, notice the indentation and how it makes the statement easier to read and understand.

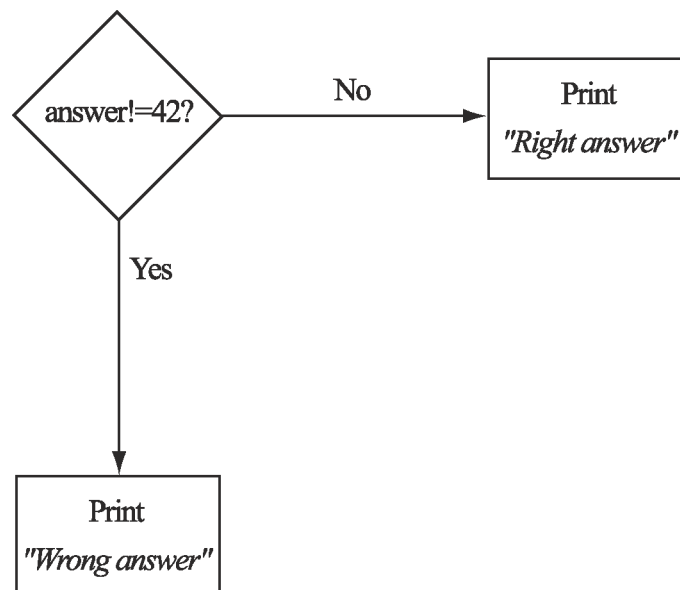
The `else` clause is an optional part of the syntax of the Java `if` statement:

```
if ( expression )
    statement;
else
    statement;
```

- `if` is the required keyword.
- `if` is followed by an expression, which must be enclosed by a pair of parentheses.
- The next statement after the `if` expression will be executed if the expression is true (the word `then` is not part of the actual syntax).
- The word `else` follows the statement controlled by the `if`.
- The next statement after the `else` will be executed if the `if` expression is false.

```
if ( answer != 42 )
    System.out.println("Wrong answer ...");
else
    System.out.println("You got the correct answer!");
```

Note there is no expression following the `else`.



CODE BLOCKS

- ✧ When you need a control statement (like an `if`) to control more than one statement, you need a way to group the controlled statements together.
- ✧ Some languages use a keyword that signals the end of the controlled statements:

```
if ( a < 10)
    statement;
    statement;
endif
```

- ✧ Some languages use special keywords that delimit the *block* of statements being controlled:

```
if a < 10
then
    statement;
    statement;
end if
```

- ✧ Other languages, like Java, use bracing characters to delimit the block:

```
if (a < 10)
{
    statement;
    statement;
}
```

A *code block* is a sequence of statements inside a pair of brace characters: { } (often called "curly braces").

Anywhere you can code a single statement, you can code a block of statements. So, if you need to have more than one statement in either your `then` or `else` clauses, enclose them in braces. Notice that braces themselves are not terminated by a semicolon.

```
if ( tempf < 32 )
{
    System.out.println("Wow, that's cold.");
    System.out.println("In fact, that's really cold!");
}
else
{
    System.out.println("We've been colder.");
    System.out.println("At least it's not freezing");
}
```

The indentation style shown is typical, but you may decide to do something else. Just be consistent.

Have you noticed that your program is actually a block of statements named `main`?

NESTING CONTROL STATEMENTS

- ✧ When you put one control statement, such as an `if`, inside another control statement, you have created *nested control statements*.
- ✧ Stated another way, a nested control statement is one that is under the control of another.

```
if ( c < 10 )
    if ( resp == 'y' )
        System.out.println("Today is a warm day");

if ( c < 10 )
    System.out.println("Today is a warm day");
else
    if ( resp == 'n' )
        System.out.println("Today is a cold day");
```

- ✧ Nested statements can be made more readable if you put the nested statement in a block.

```
if ( c < 10 )
{
    if ( resp == 'y' )
        System.out.println("Today is a warm day");
}

if ( c < 10 )
    System.out.println("Today is a warm day");
else
{
    if ( resp == 'n' )
        System.out.println("Today is a cold day");
}
```

Most languages allow unlimited nesting of control statements. You can have an `if` inside a loop which is inside of a `switch` that is inside of another `if`. There is no practical limit (other than your own ability to figure out what it all does ...).

Remember that the `if` controls another single statement. That other statement can certainly be another `if`. However, you should enclose the nested `if` statement inside of its own pair of braces — make it a block. Even though it is unnecessary (an `if` is a single statement), it makes your code easier to read and understand.

Be careful of nesting `if` statements with `else` clauses — it can be confusing. The rule is the `else` is always associated with the most recent `if` that doesn't have an `else`. In the following, notice how we associate the `else` with the outer `if` by enclosing the inner `if` in its own pair of braces (a block).

```
if ( x < y )
{
    if ( a == b )
        System.out.println("abc");
}
else
    System.out.println("sxyz");
```

This `if` and `else` go together.

Without the braces, the `else` would have been paired with the inner `if`:

```
if ( x < y )
    if ( a == b )
        System.out.println("abc");
else
    System.out.println("sxyz"); // Whose "else" IS this?
```

This `if` and `else` go together.

The compiler knows whose `else` it is, but do you?

MULTI-WAY DECISIONS: SWITCH

- ✧ Sometimes you'll compare the same variable or expression with several different values, and follow a different execution path depending on which value matches.
- ✧ A `switch` statement compares a single value to a list of possibilities.

```
switch (answer)
{
    case 0:
        statements;
        statements;
        break;
    case 1:
        statements;
        statements;
        break;
    case 2:
        statements;
        statements;
        break;
    default:
        statements;
        statements;
}
```

- ✧ A `switch` evaluates one expression to come up with a result, which it compares for equality with each case value.
 - When it finds a matching case, it starts executing the statements in that case.
- ✧ The `default` statements are executed if no matching case is found.

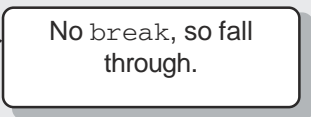
In Java, the `switch` keyword is followed by a parenthesized expression. The expression must result in a `String`, character or integer value (no floats or doubles). The block contains one or more `case` labels, consisting of the word `case`, a constant or expression (no variables), and a colon.

LeapYear.java

```
...
public class LeapYear {
    public static void main(String[] args) {
        java.util.Scanner scanner = new java.util.Scanner(System.in);
        int year = 0;

        System.out.print("Please enter the current year: ");
        year = scanner.nextInt();
        scanner.close();

        switch (year % 4) {
            case 0:
                if (year % 100 == 0) {
                    System.out.print(
                        "\"Century\" years aren't leap years.");
                    if (year % 400 == 0) {
                        System.out.println(" ..unless divisible by 400.");
                        System.out.println(year + "'s a leap year!");
                    }
                    else {
                        System.out.println(" " + year
                            + " isn't a leap year.");
                    }
                }
                else {
                    System.out.println(year + " is a leap year!");
                }
                break;
            case 3:
                System.out.print("Next year is a leap year. ");
            default:
                System.out.println(year + " isn't a leap year.");
                break;
        }
    }
}
```



The `break` statement breaks out of the `switch`. The cases in the Java `switch` statement *fall through*: once a matching case is found, execution continues until a `break` statement is reached — even the statements in other cases! If no equal comparison is found, the statements of the `default` case are executed — a sort of `else` clause for the `switch`.

EXAMPLE 7 - PRINTING LETTER GRADES BASED ON SCORES

LetterGradesCase.java

```
package examples;

public class LetterGradesCase {
    public static void main(String[] args) {
        java.util.Scanner scanner = new java.util.Scanner(System.in);

        int num1 = 0; // switch can't test floating point data types
        char grade = ' ';

        System.out.print(
            "Please enter the numeric grade for the class: ");
        num1 = scanner.nextInt();
        scanner.close();

        // Determine the letter grade
        switch (num1) {
            case 100: // Each of these cases falls through.
            case 99: case 98: case 97: case 96: case 95:
            case 94: case 93: case 92: case 91: case 90:
                grade = 'A';
                break;
            case 89: case 88: case 87: case 86: case 85:
            case 84: case 83: case 82: case 81: case 80:
                grade = 'B';
                break;
            case 79: case 78: case 77: case 76: case 75:
            case 74: case 73: case 72: case 71: case 70:
                grade = 'C';
                break;
            case 69: case 68: case 67: case 66: case 65:
            case 64: case 63: case 62: case 61: case 60:
                grade = 'D';
                break;
            default:
                grade = 'F';
                break;
        }
    }
}
```

```
        System.out.print("The student earned a " + num1);  
        System.out.print(", which is equivalent to a letter grade of ");  
        System.out.println(grade + ".");  
    }  
}
```


REVIEW QUESTIONS

1. Explain the term *sequential execution*.
2. What kind of statement is used for decision making in a computer program?
3. What kind of expression does an `if` statement evaluate?
4. When is an `if` statement's `else` clause executed?
5. Explain code blocks — when and why they are used.
6. Explain what is meant by nesting.
7. How many expressions does a `switch` statement evaluate?

LABS

- ❶ Write a program that accepts a number as input and prints a message stating whether the number is positive or negative.
(Solution: *NumSign.java*)
- ❷ Modify your solution to ❶ so that if zero is entered, it prints a message saying so.
(Solution: *NumSignZero.java*)
- ❸ Write a program that accepts a number and a letter as input. The number is a temperature, and the letter is an "F" or a "C" representing Fahrenheit or Celsius:

```
Please enter your temperature: 32
Please enter the scale: F
```

The program will then look at the letter, and based on whether it's an "F" or a "C" will perform the appropriate conversion calculation and output, for example:

```
32 degrees Fahrenheit is 0 degrees Celsius
(Solution: IfCelsFahr.java)
```

- ❹ Write a program that accepts two numbers and a character as input. The character will represent a mathematical operation to perform on the numbers:

```
Please enter a number: 8.7
Please enter an operator: *
Please enter another number: 3.2
```

Use a switch statement to determine whether the character is a +, -, *, or /, and perform the desired calculation on the numbers and print the result:

```
The result of 8.7 * 3.2 is 27.84
(Solution: BasicCalculator.java)
```

- ⑤ Write a program that reads a single character of input, and uses `if` statements to determine whether the character is a digit, a vowel, some other letter, or a non-alphanumeric character.

Hint: It may be helpful to convert the `String` you get from `scanner.next()` to a `char`:

```
String s = scanner.next();  
char c = s.charAt(0);
```

(Solution: *CharClass.java*)

- ⑥ Modify your solution to ⑤ so that it uses a `switch` statement rather than `if` statements. Have it determine only whether the character input is a digit, a vowel, or something other than a digit or a vowel.

(Solution: *CharClassSwitch.java*)

CHAPTER 8 - LOOPING

OBJECTIVES

- * Explain what looping is.
- * Write programs using iterative loops.
- * Use code blocks to contain statements in loops.
- * Choose when to use nested loops in a program.
- * Write programs using conditional loops.
- * Explain what an infinite loop is.
- * Use a break to exit a loop early.

KINDS OF LOOPS

- * A *loop* is a control statement — it controls the execution of other statements.
- * A loop executes its controlled statement(s) many times, over and over again.
- * There are two types of loops:
 - An *iterative loop* executes its statement(s) a predefined number of times.
 - A *conditional loop* executes its statement(s) as long as some condition is true, evaluating a conditional expression.
- * When your program reaches a loop statement, it stops sequential execution and "loops through" the controlled statements.
- * When the looping finishes, your program picks back up with sequential execution, starting at the next statement after the loop.

An everyday loop:

1. Lather.
2. Rinse.
3. Repeat (once).

Another:

While there's still at least one dirty dish in the pile:

1. Get a dirty dish from the pile and place it in the soapy water.
2. Wash the dish clean.
3. Place the dish in the rinse water.
4. Rinse the dish.
5. Remove the dish from the rinse water and place it in the drying rack.
6. Check the pile for dirty dishes.

Have a drink (the best wine is in the dusty bottles in the back).

ITERATIVE LOOPS

- * An iterative loop executes its controlled statement a specific number of times.
- * Iterative loops are often called `for` loops because they repeat for a certain number of times.

- In some languages, the `for` loop automatically keeps track of which iteration it's on.

```
-- Iterative loop in Oracle PL/SQL:  
FOR i IN 1 .. 10 LOOP  
    DBMS_OUTPUT.PUT_LINE(i);  
END LOOP;
```

- In other languages, like Java, you use expressions to keep the count and to check whether you've reached the end.

```
// Iterative loop in Java:  
int i;  
for ( i = 1; i <= 10; i = i + 1 )  
    System.out.println(i);
```

- The variable in which you keep track of the count is called the *loop control variable*.

- * The loop control variable can be *incremented* (increased by one) or *decremented* (decreased by one).

```
int t;  
for ( t = 10; t > 0; t = t - 1 )  
    System.out.println("T minus " + t);  
System.out.println("Liftoff!");
```

The Java `for` loop is a classic iterative loop. You determine how many times it executes, using three expressions and a loop control variable:

```
int loop-control-variable;  
for (initialization; test-expression; increment-expression)  
    statement;
```

The first expression initializes the loop control variable to a beginning value; the second expression tests to see if the control variable is within limits; and the third expression increments the control variable so that it eventually fails the second test, stopping the loop.

```
int i;  
for ( i = 0; i < 10; i = i + 1 )  
    System.out.println(i);
```

Note the two semicolons, which separate the three expressions inside the parentheses. You cannot leave out either of the semicolons. The first expression is executed only once, when your program reaches the `for` loop. Then, each time, before the loop's controlled statement is executed, the second expression is evaluated. If it's true, the loop's controlled statement is executed; if false, the loop ends. Finally, if the loop's controlled statement was executed, the `for` loop then “loops back up” and evaluates the third expression.

Hands On:

Write a new program using a `for` loop to print the numbers from 1–10 on your terminal screen. Get this to work before you go on to the next page.

Investigate:

In the expression that increments your loop control statement, instead of using syntax like `i = i + 1`, try changing the expression to look something like `i += 1`. If that works, try changing it to `i++`.

What do you think these new operators do?

CODE BLOCKS AND LOOPS

- * Loops that need to execute more than one statement per iteration use a block.

```
for ( n = 1; n < 10; n = n + 1 ) {  
    System.out.print("Value is: " + n);  
    System.out.print("\t Square is: " + (n * n));  
    System.out.println("\t Cube is: " + (n * n * n));  
}
```

- * If the *loop body* contains only one statement, braces may not be needed, but it looks nice:

```
for (a = 0; a < 10; a = a + 1) {  
    System.out.println("The count is: " + a);  
}
```

Anywhere you can code a single statement, you can code a block of statements. If you need to have more than one statement in your loop body, enclose them in braces. Notice that braces themselves are not terminated by a semicolon.

```
for (i = 10; i >= 0; i = i - 1)
{
    System.out.println("———");
    System.out.println("Count down: " + i);
}
```

The indentation style shown is typical. You may decide to do something else — just be consistent.

Investigate:

Notice also that there's no semicolon after the parentheses containing the loop control expressions. If you make a mistake and put a semicolon there,

```
for (i = 10; i >= 0; i = i - 1);
{
    System.out.println("———");
    System.out.println("Count down: " + i);
}
```

... your program will still compile (it's not a compiler error!). But when your program runs, what will happen? The compiler knows what it will do. Do you? Try it and see if you can figure out what happened.

NESTED LOOPS

- * When you put one loop statement, such as a `for` loop, inside another loop statement, you have created a *nested loop*.
- * Stated another way, a nested loop is one that is under the control of another loop.

```
for (num = 1; num <= 10; num = num + 1) {  
    for (factor = 1; factor <= 10; factor = factor + 1)  
        System.out.print(num * factor + "\t");  
    System.out.println();  
}
```

- How many times does the second output statement execute?
- * Notice how the inner loop's control variable "starts over," i.e., gets re-initialized, every time the outer loop's control variable increments.
- * The outer loop is like a big sprocket, and the inner loop like a little sprocket.
 - For each turn of the outer loop, the inner loop turns many times.

MultTable.java

```
package examples;

public class MultTable {
    public static void main(String[] args) {
        int num = 0;
        int factor = 0;

        for (num = 1; num <= 10; num = num + 1) {
            for (factor = 1; factor <= 10; factor = factor + 1) {
                System.out.print((num * factor) + "\t");
            }
            System.out.println("");
        }
    }
}
```

Notice how the braces and the indentation style show which statements are controlled by which loop.

Try It: This example prints a multiplication table.

CONDITIONAL LOOPS

- * Conditional loops are often called `while` loops because they keep repeating while a condition is true.
- * A conditional loop has only one expression — typically a relational expression — to control the operation of the loop.

```
while (number != 0) {  
    sum = sum + number;  
    System.out.print("Enter another number (0 to quit): ");  
    number = scanner.nextInt();  
}
```

- * The loop evaluates its test expression each time it executes:
 - The loop continues looping if the result is true.
 - The loop stops looping if the result is false.
- * The expression is evaluated before the loop is first executed, and it is possible that the body of the loop may never execute at all.

```
int number = 0; // Oops, loop will never happen ...  
while (number != 0) {  
    sum = sum + number;  
    System.out.print("Enter another number (0 to quit): ");  
    number = scanner.nextInt();  
}
```

- * `do` loops run the test expression at the bottom, after executing once.

```
do {  
    sum = sum + number;  
    System.out.print("Enter another number (0 to quit): ");  
    number = scanner.nextInt();  
} while (number != 0); // Note the semicolon here.
```

The Java `while` loop is a classic conditional loop. It loops through and executes the statements it controls — its body — as long as the expression evaluates as true.

```
while ( expression )  
    statement;
```

- `while` is the required keyword.
- `while` is followed by an expression, always enclosed by a pair of parentheses.
- The statement after the `while` expression will be executed if the expression is true.
- After the statement is executed, the expression is evaluated again.
- This continues until expression is false.

As always, you can use a code block as the statement. It's called the *loop body*, whether it's a single statement or a block.

Sum.java

```
package examples;  
  
public class Sum {  
    public static void main(String[] args) {  
        java.util.Scanner scanner = new java.util.Scanner(System.in);  
  
        int sum = 0;  
        int number = 0;  
  
        System.out.print("Enter a number: ");  
        number = scanner.nextInt();  
  
        while (number != 0) {  
            sum = sum + number;  
            System.out.print("Enter another number (zero to quit): ");  
            number = scanner.nextInt();  
        }  
        System.out.println("The total is: " + sum);  
        scanner.close();  
    }  
}
```

Try It: Compile and run *Sum.java*. When you're ready for the sum to be displayed, enter 0.

INFINITE LOOPS

- ✴ An *infinite loop* is one that doesn't know when to stop; it loops forever.
- ✴ Since an infinite loop will make your program run forever, there has to be a way to get out of the loop.
 - A `break` statement tells a loop to end, and resume sequential execution at the next statement after the loop body.

```
while (true) {  
    ...  
    if (choice == 'q' || choice == 'Q')  
        break;  
}  
System.out.println("You're out of the loop.");
```

- A `break` statement is usually controlled by an `if` statement.
- ✴ You may wonder "Why?"; but infinite loops have many uses.
 - If a program presents a menu, you choose an option, and then you are presented with the menu again, you are using an infinite loop.
 - The loop will continue until you break out of it by entering the "quit" option.
 - Your operating system command interpreter operates in an infinite loop while prompting you for commands.
 - The loop will continue until you break out of it by entering the command to exit.

You can make an infinite `for` loop by leaving out all of the expressions:

```
for (;;) { // An infinite for loop has no expressions
    ...
}
```

You can make an infinite `while` loop by using any constant, true value for the conditional expression.

```
while (1 != 0) { // This is always true! 1 is never equal to 0.
    ...
}
```

You may see Java infinite `while` loops written like this:

```
while (true) {
    ...
}
```

The words `true` and `false` are constants in the Java language — essentially keywords or reserved words. They are the only two values that a boolean variable can be assigned or can result from the evaluation of a boolean expression. Other languages may use integer values such as 1 and 0 to represent true and false.

A `break` statement alters the normal execution of the body of the loop by telling the loop to quit — right now, right away. Execution will pick back up with the first statement that occurs after the loop.

EXAMPLE 8 - A SIMPLE MENU PROGRAM

SimpleMenu.java

```
package examples;

public class SimpleMenu {
    public static void main(String[] args) {
        java.util.Scanner scanner = new java.util.Scanner(System.in);

        int resp = 0;
        float num1 = 0.0F, num2 = 0.0F;

        while (true) {
            System.out.println("                Arithmetic Menu");
            System.out.println("                _____");
            System.out.println("Option  Description");
            System.out.println("_____");
            System.out.println("0      Exit");
            System.out.println("1      Add two numbers");
            System.out.println("2      Subtract two numbers");
            System.out.println("3      Multiply two numbers");
            System.out.println("4      Divide two numbers\n");

            System.out.print("                Please choose an option: ");
            resp = scanner.nextInt();

            if (resp == 0)
                break;

            switch (resp) {
                case 1:
                    System.out.println("Adding two numbers ...");
                    System.out.print("Enter the first number: ");
                    num1 = scanner.nextFloat();
                    System.out.print("Enter the second number: ");
                    num2 = scanner.nextFloat();
                    System.out.println(num1 + " + " + num2 + " = "
                        + (num1 + num2));
                    break; // only breaks out of the switch
            }
        }
    }
}
```

```
        case 2:
            System.out.println("Subtracting two numbers ..");
            System.out.print("Enter the first number: ");
            num1 = scanner.nextFloat();
            System.out.print("Enter the second number: ");
            num2 = scanner.nextFloat();
            System.out.println(num1 + " - " + num2 + " = "
                               + (num1 - num2));
            break;
        case 3:
            System.out.println("Multiplying two numbers ...");
            System.out.print("Enter the first number: ");
            num1 = scanner.nextFloat();
            System.out.print("Enter the second number: ");
            num2 = scanner.nextFloat();
            System.out.println(num1 + " * " + num2 + " = "
                               + (num1 * num2));
            break;
        case 4:
            System.out.println("Dividing two numbers ...");
            System.out.print("Enter the first number: ");
            num1 = scanner.nextFloat();
            System.out.print("Enter the second number: ");
            num2 = scanner.nextFloat();
            System.out.println(num1 + " / " + num2 + " = "
                               + (num1 / num2));
            break;
        default:
            System.err.println("Invalid response. ");
            break;
    }
}
scanner.close();
}
```

REVIEW QUESTIONS

1. What's the difference between an iterative loop and a conditional loop?
2. What is the common name for an iterative loop?
3. What is the common name for a conditional loop?
4. What is the term for the statement or group of statements controlled by a loop?
5. What is a nested loop?
6. What is an infinite loop? How is an infinite loop stopped?

LABS

You've now learned enough programming fundamentals to start writing some more interesting programs. Conditionals, loops, variables and data types, input and output — putting these together so that the computer does what you want it to do is the programmer's craft and profession.

This chapter includes a large number of lab exercises for you to work on both during the lab session in class, and later, on your own time. You'll probably finish just a few of these during the lab session. Doing more of these exercises on your own, at your own pace, is the best way to reinforce and practice both the syntax of the language you're writing your programs in and the kind of logical thinking and problem solving all programmers must master.

- ①
 - a. Write a program that uses a for loop to print out the first 16 multiples of 2.
(Solution: *Mult2.java*)
 - b. Make a copy of your solution and modify it so that, in addition to printing the first 16 multiples of 2, it also prints the first 16 powers of 2. Have both the multiple and the power print on the same line.
(Solution: *Pow2.java*)
- ② Modify your temperature conversion program from the previous chapter and use a loop that will allow for 5 different temperatures to be input and converted.
(Solution: *LoopFahrCels.java*)
- ③
 - a. The Fibonacci series is a series of numbers in which the first two numbers are 0 and 1. Each subsequent number is the sum of the previous two. So, the third Fibonacci number is $(0 + 1) = 1$. The next is $(1 + 1) = 2$. Then $(1 + 2) = 3$. Then $(2 + 3) = 5$. Then $(3 + 5) = 8$. And so on. The Fibonacci series has many interesting mathematical properties and shows up in surprising places in nature.

Write a program that uses a for loop to print the first 20 Fibonacci numbers. Remember that the first two (known as f_0 and f_1) are predefined as 0 and 1. All subsequent numbers are calculated from there.
(Solution: *Fib.java*)

- b. Modify your solution (if necessary) so that the sequence number is printed out next to the fibonacci value. Make sure that sequence numbers 0 and 1 match up with the first two predefined values:

```
0 : 0
1 : 1
2 : 1
3 : 2
4 : 3
5 : 5
6 : 8
7 : 13
...
```

(Solution: *FibNum.java*)

- ④ Write a program that uses nested `for` loops to print a multiplication table for the integers -5 through 5. For each integer, print all of its multiples on a single line. You might want to start with a copy of *MultTable.java*.
(Solution: *MultTable5.java*)
- ⑤ Write a program that uses nested `for` loops to print a table of the first 8 powers of the first 10 integers.
Hint: You might want to start with a copy of *MultTable.java*.
(Solution: *PowTable.java*)
- ⑥ Modify your calculator program from the previous chapter, using a loop to allow for as many different calculations as the user wants to do. When both input numbers are 0, then break out of the loop and end the program.
(Solution: *LoopCalc.java*)
- ⑦ (Optional) Make a copy of your solution to ③, the Fibonacci number exercise. Modify the copy to calculate only the first 10 Fibonacci numbers. Instead of printing the value of the Fibonacci number, use a nested `for` loop to print a row of asterisks whose length is the current Fibonacci number.

If you get this working, then experiment. What happens if you calculate 15 Fibonacci numbers? How about 20?

(Solution: *FibGraph.java*)

Continued . . .

LABS (CONTD.)

- 8 Write a program that prints the factorials of the integers from 1 to 10. The factorial of a number N is the product of all numbers less than or equal to N : factorial 1 ($1!$) is just one. $2!$ is $(1 * 2) = 2$. $3!$ is $(1 * 2 * 3) = 6$. $4!$ is $(1 * 2 * 3 * 4) = 24$. And so on. Have your program print the results in a tabular format:

```
1! = 1
2! = 2
3! = 6
4! = 24
...
```

When you get your program working, experiment: What's the highest factorial the computer can correctly calculate?

(Solution: *Factorial.java*)

- 9 a. Write a program with an infinite loop, reading a floating-point number from the user at each iteration. The program should maintain the sum of the numbers entered so far. Have the program exit when the user enters zero. At each iteration, print the last number entered, and the current sum. Print the data in a tabular format:

```
Entered    Sum
12         12
9          21
14.3       35.3
...
```

(Solution: *RunningSum.java*)

- b. Modify your solution so that in addition to the running sum, the program also calculates the average of the numbers so far. Have it print the average, as well as the count of the numbers entered so far.

(Solution: *RunningSumAvg.java*)

- c. Modify your solution so that it also prints the largest and smallest numbers entered so far.

(Solution: *RunningStats.java*)

- d. An alternative algorithm for calculating a running average is:

$$\text{average} = (\text{previous average}) + ((\text{new number} - \text{previous average}) / \text{count})$$

Modify your program to maintain the average calculated this way, in addition to the more traditional way. Compare the results. Is there any situation in which one algorithm is better than the other?

(Solution: *RunningAvg.java*)

CHAPTER 9 - METHODS

OBJECTIVES

- * Use methods to handle repetitive tasks in a program.
- * Call a method from within your program.
- * Pass values as parameters to your methods.
- * Return a value from your method and use it within your program.
- * Use both locally and class scoped variables.
- * Use method stubs during the development of an application.
- * Describe the purpose of a library.

PROGRAMMING WITHOUT METHODS

- * Often your program will need to perform the same set of instructions at different points in the program.
- * You can simply rewrite the needed statements in your program.
- * But duplicating code is not the best approach.
 - You might make a mistake and copy that same mistake several times.
 - You might later find a better way to do the task and need to update all of the copied statements.
- * Computers are well suited to repetitive tasks, and we want to take advantage of that.
 - If we can tell the computer how to do a task once, we should be able to ask it to perform the task over and over.
 - You might give someone detailed directions to get to your house once, but after that, you just tell them to come over!

Interest1.java

```

package examples;

public class Interest1 {
    public static void main(String[] args) {
        java.util.Scanner scanner = new java.util.Scanner(System.in);

        double principal = 0.0;
        double rate = 0.0;
        double payment = 0.0;

        System.out.print("\nEnter the principal amount of the loan ");
        principal = scanner.nextDouble();

        System.out.print("Enter the interest rate for the loan ");
        rate = scanner.nextDouble();
        scanner.close();

        payment = (principal + 30 * principal * (rate / 100))
            / (30 * 12);

        System.out.println(
            "\nPrincipal\tInterest Rate\tMonthly Payment");

        System.out.println("-----\t\t-----\t"
            + "\t-----");

        System.out.println(principal + "\t\t\t" + rate + "\t\t\t"
            + payment);
        payment = (principal + 30 * principal * ((rate - .5) / 100))
            / (30 * 12);

        System.out.println(principal + "\t\t\t" + (rate - .5)
            + "\t\t\t" + payment);
        payment = (principal + 30 * principal * ((rate + .5) / 100))
            / (30 * 12);

        System.out.println(principal + "\t\t\t" + (rate + .5)
            + "\t\t\t" + payment);
    }
}

```

Notice the repeated code.

Try It: This example asks for a principal amount and interest rate, then calculates the monthly payment. It then calculates the payment for interest rates a ½ point below and a ½ point above the entered rate.

REUSABLE CODE IN A METHOD

- * *Methods* make your program smaller by placing common statement sequences in a reusable place.
 - A method has a name, which you use in your program to execute the statements in the method.
- * Well-structured, modular applications make programming more manageable by letting you concentrate on smaller and easier pieces of code one at a time.
- * Using methods also makes your program shorter, more readable, and more maintainable.
 - Your program is shorter because you are not rewriting several lines of code each time you need to perform the same task.
 - Your program is more readable because you have given the method a name which describes the task it performs.
 - Your program is more maintainable because each method can be tested and modified independently.
- * Methods are called different things in different programming languages.
 - *Procedure*, *function*, and *subroutine* are other names for a method.
 - All of these are named blocks of code that are called to perform some task.

Interest2.java

```

package examples;

public class Interest2 {
    public static void main(String[] args) {
        ...
        System.out.print("\nEnter the principal amount of the loan ");
        System.out.print("(such as 25000): ");
        principal = scanner.nextDouble();

        System.out.print("Enter the interest rate for the loan ");
        System.out.print("(such as 7.5): ");
        rate = scanner.nextDouble();
        scanner.close();

        printHeading();
        payment = (principal + 30 * principal * (rate / 100))
            / (30 * 12);
        System.out.println(principal + "\t\t\t" + rate + "\t\t\t"
            + payment);
        System.out.println("\nTry some other rates:");

        printHeading();
        payment = (principal + 30 * principal * ((rate - .5) / 100))
            / (30 * 12);
        System.out.println(principal + "\t\t\t" + (rate - .5)
            + "\t\t\t" + payment);

        payment = (principal + 30 * principal * ((rate + .5) / 100))
            / (30 * 12);
        System.out.println(principal + "\t\t\t" + (rate + .5)
            + "\t\t\t" + payment);
    }

    public static void printHeading() {
        System.out.println(
            "\nPrincipal\t\tInterest Rate\t\tMonthly Payment");
        System.out.println(
            "-----\t\t-----\t\t-----");
    }
}

```

Try It: This version of the Interest program moves the heading printing to a method.

THE STARTING POINT

- ✴ Every programming language has a different way of determining where a program should start executing:
 - In COBOL, the program starts executing with the first statement in the PROCEDURE-DIVISION.
 - In Fortran, the program starts executing with the first executable statement in the source code.
 - In Java, the program begins execution with the program's `main()` method.
- ✴ *Calling* a method causes the code in that method to execute, and then control moves back to the place where it was called from.

```
...
payment = (principal + 30 * principal *
           (rate / 100)) / (30 * 12);

printHeading();

System.out.println(principal + "\t\t\t" + rate + "\t\t\t" +
                   payment);
...
```

- ✴ If the first method never calls any other methods, then the code in those methods will never be used.
- ✴ Methods can call methods which call methods which call methods which call methods . . .

Java programmers use the term "method." Other languages, such as C++ and Pascal, use the term "function" in place of "method."

VARIABLE VISIBILITY: SCOPE

- * A variable's *scope*, or *visibility* specifies where the variable can be used in your program.
- * The variables that we have been using so far are all *local* variables:

```
public static float calculatePayment(double prin,  
    double rate){  
    double payment = 0.0;  
    ...  
}
```

- They are local to the method they were defined in; only that method is able to use them in its statements and expressions.
 - A local variable is only visible to the block of code that it is defined in.
- * A *field* is defined outside of any method definition.

```
private static float maxRate = 10.0F;  
  
public static float calculatePayment(double prin,  
    double rate){  
    double payment = 0.0;  
    ...  
    if(rate > maxRate)  
        rate = maxRate;  
    ...  
}
```

- Fields can be defined above, between, and below the method definitions, but are usually placed at the top of the file.
- A field variable is visible to all methods.

Local variables are those that a specific method is concerned with. They have some meaning that is specific to a particular method. Fields are those that have meaning to the entire class.

PARAMETERS

- * *Parameters* are the "input data" to a method — the data we want the method to process.

```
public static void printPayment(double prin, double rate,
    double pay) {
    System.out.println(prin + "\t\t\t" + (rate+.5) +
        "\t\t\t" + pay);
}
```

- A method can take zero, one, or many parameters.

- * When you call a method, you must use the same number and type of data items in your method call as were specified in the method definition.

```
printPayment(principal, rate - .5, payment);
```

- * Copies of the data being passed, the *arguments*, are sent to the method.
- * The method uses its own parameter names for the copied data.
 - Parameters within the method act just like any other local variable and can only be used within the method itself.
- * Sometimes the terms argument and parameter are used interchangeably.

Interest3.java

```

package examples;

public class Interest3 {
    public static void main(String[] args) {
        ...
        System.out.print("\nEnter the principal amount of the loan ");
        System.out.print("(such as 25000): ");
        principal = scanner.nextDouble();

        System.out.print("Enter the interest rate for the loan ");
        System.out.print("(such as 7.5): ");
        rate = scanner.nextDouble();
        scanner.close();

        printHeading();
        payment = (principal + 30 * principal * (rate / 100))
            / (30 * 12);
        printPayment(principal, rate, payment);
        System.out.println("\nTry some other rates:");

        printHeading();
        payment = (principal + 30 * principal * ((rate - .5) / 100))
            / (30 * 12);
        printPayment(principal, rate - .5, payment);
        payment = (principal + 30 * principal * ((rate + .5) / 100))
            / (30 * 12);
        printPayment(principal, rate + .5, payment);
    }

    public static void printHeading() {
        System.out.println(
            "\nPrincipal\t\tInterest Rate\t\tMonthly Payment");
        System.out.println(
            "-----\t\t-----\t\t-----");
    }

    public static void printPayment(double princ, double rate,
        double pay) {
        System.out.println(princ + "\t\t\t" + rate + "\t\t\t" + pay);
    }
}

```

Try It: This version of the Interest program adds a method for printing the payment.

RETURNING A VALUE

- * Methods can perform a task and return a result.

```
public static double calculatePayment(double prin,  
    double rate) {  
    double pay = 0.0;  
    pay = (prin + 30 * prin * (rate/100) / (30*12));  
    return pay;  
}
```

- The return statement must return a value of the appropriate data type, whether it's a variable or a literal.

- * In most languages, a method can return only one value.

- * When you call a method, you must use the method in a context where the expected return value will be accepted correctly.

```
payment = calculatePayment(principle, rate);
```

- If a method returns a `float`, you will use the method call where a `float` would be expected and useful.

- * When the method returns, the data being returned will essentially "replace" the method call.

```
printPayment(pr, rt, calculatePayment(pr, rt));
```

Interest4.java

```
package examples;

public class Interest4 {
    public static void main(String[] args) {
        ...
        System.out.print("\nEnter the principal amount of the loan ");
        System.out.print("(such as 120000): ");
        principal = scanner.nextDouble();

        System.out.print("Enter the interest rate for the loan ");
        System.out.print("(such as 7.5): ");
        rate = scanner.nextDouble();
        scanner.close();

        printHeading();
        payment = calculatePayment(principal, rate);
        printPayment(principal, rate, payment);
        System.out.println("\nTry some other rates:");

        printHeading();
        payment = calculatePayment(principal, rate - .5);
        printPayment(principal, rate - .5, payment);
        payment = calculatePayment(principal, rate + .5);
        printPayment(principal, rate + .5, payment);
    }

    public static void printHeading() {
        System.out.println(
            "\nPrincipal\t\tInterest Rate\t\tMonthly Payment");
        System.out.println(
            "-----\t\t-----\t\t-----");
    }

    public static void printPayment(double princ, double rate,
        double pay) {
        System.out.println(princ + "\t\t\t" + rate + "\t\t\t" + pay);
    }

    public static double calculatePayment(double princ, double rate) {
        double pay = 0.0;
        pay = (princ + 30 * princ * (rate / 100)) / (30 * 12);
        return pay;
    }
}
```

Try It: This example calculates and returns the payment from a method.

METHOD STUBS

- * When you design a computer program, you may come up with some ideas of methods that you would like to write and use.
- * Most languages allow you to code a "skeleton" method — one that is callable but has no actual statements in it yet.
- * This type of method skeleton is usually referred to as a *method stub*.
 - Method stubs have only the basic method syntax necessary to get the program to compile and run without error.
- * The advantage of stubs are twofold:
 - You don't have to finish coding a method until you are ready.
 - The method "works" — it is callable — and your program will still compile and run.
- * Method stubs let you write your program in step-by-step pieces and help tremendously in the debugging process.

MenuStub.java

```
package examples;

public class MenuStub {
    public static void main(String[] args) {
        ...
        while (true) {
            System.out.println("\nCalculator Menu");
            System.out.println("-----");
            System.out.println(" (A)dd");
            System.out.println(" (S)ubtract");
            System.out.println(" (M)ultiply");
            System.out.println(" (D)ivide");
            System.out.println(" (Q)uit");
            System.out.print("\nYour choice: ");
            input = scanner.next();
            if (input.equals("Q") || input.equals("q"))
                break;

            switch (input.charAt(0)) {
                case 'a': case 'A':
                    add();
                    break;
                case 's': case 'S':
                    subtract();
                    break;
                case 'm': case 'M':
                    multiply();
                    break;
                case 'd': case 'D':
                    divide();
                    break;
                default:
                    System.err.println("\nInvalid option.");
                    break;
            }
        }
        scanner.close();
    }

    // Use method stubs until we get the menu looking and working right.
    public static void add() { }
    public static void subtract() { }
    public static void multiply() { }
    public static void divide() { }
}
```

LIBRARIES

- ✧ Many organizations have developed a number of company-specific classes and methods that other programmers will find useful in writing programs for the organization.
- ✧ Most compilers also come with a number of language-specific classes and methods useful in writing programs in that language.
- ✧ To make these methods available to everybody, they are usually stored in a library.
 - A *library* is nothing more than a file, directory, or folder that contains the executable code for one or more classes and methods.
- ✧ In Java, you can use an `import` statement to specify which classes you will be using from a library so that you can reference those classes more easily.

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

By default, the Java compiler knows how to locate and use some standard classes and methods. There are a number — a large number — of utility methods stored there that can help you perform common tasks from input/output, to string manipulation, to date and time processing, and much more.

| | |
|----------------------|---|
| java.lang: | |
| System.out.print() | - print a string to standard output. |
| System.err.print() | - print a string to standard error. |
| System.out.println() | - print a string to standard output w/newline. |
| System.err.println() | - print a string to standard error w/newline. |
| System.exit() | - exit the program with an error code. |
| java.lang.String: | |
| String.equals() | - compare strings for equality. |
| String.charAt() | - retrieve a character by position from string. |
| java.lang.Math: | |
| Math.sqrt() | - calculate a square root. |

The list of methods that a system provides is called the system's *Application Programming Interface* (API). That is, it's the list of methods that an application program can use to interface with the system.

There are also many Java libraries available on the Internet or with tools that you may purchase. Don't forget to ask around the office. There may be company-specific libraries containing methods that you can use.

EXAMPLE 9 – SQUARE AND SQUARE ROOT

SquareAndRoot.java

```
package examples;

import java.util.*;

public class SquareAndRoot {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        double aNum = getNumber(true, scanner);
        while (true) {
            calcSquareAndRoot(aNum);
            aNum = getNumber(false, scanner);
        }
    }

    public static double getNumber(boolean firstTime, Scanner sc) {

        String theResp = null;
        double theNum = 0.0;
        if (firstTime == false) {
            System.out.print("Would you like to do another? (y/n): ");
            theResp = sc.next();

            if (theResp.charAt(0) == 'n' || theResp.charAt(0) == 'N') {
                sc.close();
                System.exit(0);
            }
        }

        System.out.print("Please enter a number: ");
        theNum = sc.nextDouble();
        return theNum;
    }

    public static void calcSquareAndRoot(double theNum) {
        System.out.println("You entered the number: " + theNum);
        System.out.println("\tThe square of that number is: "
            + square(theNum));
        System.out.println("\tThe square root of that number"
            + " is: " + Math.sqrt(theNum));
    }
}
```

Since we are importing
java.util.*, we no longer need to
fully qualify the Scanner class.

```
public static double square(double x) {  
    double result;  
    result = x * x;  
    return result;  
}  
}
```

REVIEW QUESTIONS

1. What is a method?
2. Why do you use methods?
3. What is the syntax necessary to call a method from within your program?
4. What is the difference between a field and a local variable?
5. How do you pass parameters to the methods in your program?
6. Explain how to accept a return value from a method in your program.
7. Where does a program actually start executing?
8. How do you define a method?
9. What is a method stub?
10. Explain the purpose of a library.

LABS

- ❶ Write a program that contains a method named `printHello()`. The `printHello()` method should print a simple message on the screen. Call your `printHello()` method from `main()`. (Solution: *MethodHello.java*)
- ❷ Write a program with a method that calculates a power — a base number raised to the power of an exponent. The method should take two parameters: a `double` (the base) and an `int` (the exponent). The method should return a `double`: the base raised to the power of the exponent. In the `main()` method, prompt the user to enter the base and the exponent; use the method to perform the calculation; then print the result returned by the method. (Solution: *PowerMethod.java*)
- ❸ Find your temperature conversion program from a previous chapter and put the temperature calculations in a method. In your `main()` method, prompt for a number to be converted and pass the number to the method. Let the method print the results of the conversion. (Solution: *MethodTempConv.java*)
- ❹ Find your "next largest multiple" program from a previous chapter and put the calculation in a method. In your `main()` method, prompt for the numbers to be used in the calculation and pass them to the method. Return the result back to `main()`, where the result will be printed out. (Solution: *MethodNextMult.java*)
- ❺ Write a program that prompts the user to enter the X and Y coordinates of two points. The program should define and use a method that will then calculate the distance between the two points. The distance between two points is found with:
$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$
Have `main()` get the points from the user, and print the result. You can use the power method you created in the earlier exercise to do the squaring. Use the standard `java.lang.Math` method `sqrt()` to take the square root. (Solution: *PointDistance.java*)
- ❻ Look up the trigonometric methods for sine, cosine, and tangent in the `java.lang.Math` class (`sin()`, `cos()`, `tan()`). Write a program that prints the values of the three trigonometric methods for angles between 0 and 3.5 radians in 0.1 radian increments. (Solution: *TrigTable.java*)

CHAPTER 10 - DATA COLLECTIONS – ARRAYS

OBJECTIVES

- * Explain the difference between scalar data and data collections.
- * Define array variables to hold several pieces of data.
- * Use an index or subscript with an array to access individual data items.

SCALAR DATA VS. DATA COLLECTIONS

- * Up to this point, we have been using what is usually called *scalar* data in our programs:
 - A scalar variable contains a single value.
- * Data *collections*, as the name implies, are collections of multiple data items:
 - Just like a scalar variable, a variable that contains a data collection has a single name.
 - But that single name refers to multiple, independent data items.
 - You can work with the collection as a whole, or with its data items individually.
- * We'll look at two types of data collection in this course:
- * An *array* is a collection of data items that are all of the same data type.
- * A *class* is a collection of data items that may be of different data types.
 - A class can also include methods.
- * We'll look at arrays in this chapter, and classes later.

A scalar variable has room for a single piece of data. All of the variables we have been using in our programs so far have been scalars.

```
int sum = 0;
char middleInitial = 'Q';
float volume = 65.4498F;
double pi = 3.1415926535898;
```

| | | | |
|-----|---------------|---------|-----------------|
| sum | middleInitial | volume | pi |
| 0 | Q | 65.4498 | 3.1415926535898 |

An array is a collection of related data items with a single variable name. You access the individual elements of the array using a numeric index or subscript.

A class is a collection of related data items with a single name. You access the individual members of a class using an object declared to be of the class type.

WHAT IS AN ARRAY?

- * An array is a data collection that contains a number of data items that are all of the same data type.
- * The data items, or *elements*, of the array are stored in consecutive memory locations, and each is assigned a number: its *index*.

| | | | | | | | | | | | | |
|-----------|---|---|---|---|---|---|---|---|---|---|----|----|
| Elements: | | | | | | | | | | | | |
| Index: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |

- * You use the index to access a particular element of the array.
- * Any time you have a number of related data items that are all of the same type, you should consider using an array:
 - The number of milk bottles going through a filling machine each weekday could be stored in an array of 7 integers.
 - Student letter grades for each of six classes they took could be stored in an array of 6 characters.
 - The average monthly precipitation for a year could be stored in an array of 12 floating-point elements.
- * When you define an array, you have to tell the compiler how many elements it will contain.

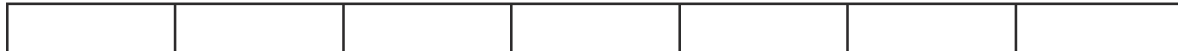
```
int bottleCount[] = new int[7];
```

To declare an array in Java, you follow the array's name with an empty pair of square brackets. Then the new operator is used with the number of elements you want to store in the array (the array's *size* or *length*). Arrays are always declared and accessed with square brackets:

```
int bottleCount[] = new int[7];
char courseGrade[] = null;
float monthlyPrecip[] = new float[12];

courseGrade = new char[6];
```

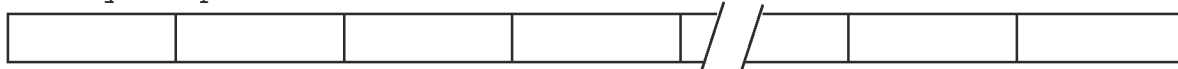
bottleCount



courseGrade



monthlyPrecip



ACCESSING ARRAY ELEMENTS

- * Using a numeric index, you can store data into or retrieve data from a particular array element.
- * The index number must be attached to the array name to tell the compiler which element of the array you want to use:

```
bottleCount[1] = 5733;
```

- * In many languages, the first element has index 0.

```
courseGrade[0] = 'B'; // the first element
```

- Think of the index as the number of elements you have to skip to get to the one you want.

```
courseGrade[3] = 'A'; // the FOURTH element  
System.out.print(courseGrade[5]); // the LAST element
```

- * Use individual elements of an array just like scalar variables.

```
seasonDiff = (monthlyPrecip[7]-monthlyPrecip[1]);
```

- * Iterative loops, with their loop control variable, are often used to access elements of an array in turn, one after the other.

```
for(i = 0; i < 7; i++)  
    System.out.println(i + ": " + bottleCount[i]);
```

- * Trying to assign a value beyond the end of an array causes a catastrophic error in most languages.

```
courseGrade[8] = 'F'; // No such element!
```

Java and its relatives (C++, C#, C, Perl, etc.) use *zero-based arrays* — the first index is 0, and the index of the last element is always one number less than the array's length.

```
int bottleCount[] = new int[7];
```

bottleCount

| | | | | | | |
|---|------|---|---|---|---|---|
| | 5733 | | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |

courseGrade

| | | | | | |
|---|---|---|---|---|---|
| B | B | C | A | B | A |
| 0 | 1 | 2 | 3 | 4 | 5 |

monthlyPrecip

| | | | | | | |
|---|---|---|---|---|----|----|
| | | | | | | |
| 0 | 1 | 2 | 3 | 9 | 10 | 11 |

MULTIDIMENSIONAL ARRAYS

- * Most languages permit you to define *multidimensional arrays*.
- * Two-dimensional arrays, which are quite common, are used to represent tables of data, points on a terminal screen, locations on a map, etc.

```
char TicTacToe[][] = new char[3][3]; // 3 rows and 3 columns
```

- * You must use two indexes to access an element of a two-dimensional array:

```
TicTacToe[1][1] = 'X'; // Put an X in the center.  
TicTacToe[0][2] = 'O'; // O in the top right.
```

➤ Think of the indexes as row and column numbers.

- * Three-dimensional arrays can be used to represent points in space (in engineering programs, for example), and even higher-dimension arrays are used in scientific and mathematical programs.
- * Every Java array has an attribute called `length` that will contain the number of elements in the array.

```
System.out.println("bottleCount has " +  
    bottleCount.length + "elements");
```

```
char TicTacToe[][] = new char[3][3];
```

TicTacToe

| | | | |
|---|---|---|---|
| 0 | | | O |
| 1 | | X | |
| 2 | | | |
| | 0 | 1 | 2 |

```
String workWeek[][] = null;
```

```
workWeek = new String[8][5];
```

workWeek

| | | | | | |
|-------|---------|------------|-----------|----------|------------|
| 9:00 | | | | "Barber" | |
| 10:00 | | | | | |
| 11:00 | | | "Dentist" | | |
| 12:00 | "Lunch" | | "Lunch" | | "Lunch" |
| 1:00 | | | | | "Nap" |
| 2:00 | | "Tee Time" | | | |
| 3:00 | | | | | |
| 4:00 | | | | | "Timecard" |
| | Monday | Tuesday | Wednesday | Thursday | Friday |

ARRAY INITIALIZATION

- * Attributes are initialized by the Java compiler; local variables must be initialized by the Java programmer:

```
public class CalcAverages{
    int totalNumber;

    public static void main(String[] args){
        int theCount = 0;
        ...
    }
}
```

- * When the programmer (you) supplies initial values for array elements, individual assignment statements are cumbersome:

```
message[0] = 'H';
message[1] = 'e';
message[2] = 'l';
...
```

- * Java provides a shorthand for allocating and initializing an array at the same time:

```
char message[] = {'H', 'i', '!'};
int dailyTotal[] = {0, 0, 0, 0, 0, 0, 0};
```

- * To initialize a two-dimensional array, initialize the elements of each row:

```
// Three stores each reported monthly number of
// employees for the past year. 3 rows, 12 cols.

int empsPerLocation[][] = {
    { 9, 8, 8, 9, 9,10,10,10,10,11,11,11},
    { 5, 5, 5, 5, 5, 4, 4, 4, 4, 3, 3, 2},
    {14,15,15,16,17,21,21,21,21,21,21,23}
};
```


EXAMPLE 10 - CALCULATING AVERAGE RAINFALL

RainFall.java

```
package examples;

import java.util.*;

public class RainFall {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        String dayNames[] = { "Sunday", "Monday", "Tuesday",
                               "Wednesday", "Thursday", "Friday", "Saturday" };
        float rainFall[] = new float[7];
        float total = 0.0F;
        float average = 0.0F;
        int i = 0;

        for (i = 0; i < 7; i = i + 1) {
            System.out.print("Enter the rainfall level for "
                             + dayNames[i] + ": ");
            rainFall[i] = scanner.nextFloat();
        }
        scanner.close();

        for (i = 0; i < 7; i = i + 1)
            total = total + rainFall[i];

        average = total / 7;

        System.out.println("The total rainfall for the week was: "
                           + total);
        System.out.println("The average rainfall for the week was: "
                           + average);
    }
}
```


REVIEW QUESTIONS

1. Describe the difference between a scalar variable and a data collection.
2. What is an array?
3. How do you access an array element?

LABS

- ①
 - a. Write a program that declares an array of ten integers. Initialize the array (when you declare it) with the first ten positive odd numbers. Print the first and last elements of the array.
(Solution: *TenOdds1.java*)
 - b. Modify the solution to use a loop to print all of the elements of the array, showing each element on its own line in the output.
(Solution: *TenOdds2.java*)
 - c. Modify the solution so that all the elements of the array are printed on the same line, with a single space between each one.
(Solution: *TenOdds3.java*)
- ②
 - a. Write a program that declares an array of ten floating-point numbers. Use a loop to read in the values of the array. After all the elements have been read, print them all on one line.
(Solution: *TenFloats.java*)
 - b. Modify the solution. After all of the elements have been read and printed, use a loop to find the largest number entered. Print the array index of the number and the number.
(Solution: *TenFloatsLargest.java*)
- ③ Find your "test scores" program that you wrote back in Chapter 4. Convert it to use an array of integers instead of individual integer variables for the test scores.
(Solution: *ArrayStudentScores.java*)
- ④
 - a. Write a program that uses a 2-D `char` array to represent a tic-tac-toe board. Initialize the first and last rows with X's and the middle row with O's:

```
XXX
OOO
XXX
```

Use a nested loop to print the board.
(Solution: *Ttt.java*)

- b. Modify your solution. Create a method called `printBoard` whose only parameter is a 3x3 `char` array. The method should contain the loops to print the array.
(Solution: *TttMethod.java*)

- c. Modify your solution. Initialize the array to all asterisks when it is declared. Prompt for an x or an o, as well as the row (1 to 3) and the column (1 to 3) of a square. Assign the x or o to the correct element and print the board.
(Solution: *TttInput.java*)
- d. Modify your solution to make sure both the row and column entered are within proper range. If either is out of range, then print an error message and terminate the program.
(Solution: *TttCheck.java*)
- e. Modify your solution to put the program in a loop. Call `printBoard` each time the user provides input, and quit the program when a q is input. Also, modify the row/column range, checking so that after the error message, instead of quitting the program, just loops to let the user continue building the board.
Hint: The `continue` statement causes execution to go back to the top of the loop, whereas `break` exits the loop.
(Solution: *TttLoop.java*)

CHAPTER 11 - DEBUGGING

OBJECTIVES

- * Explain what is meant by the term *debugging*.
- * Do some simple debugging by commenting out statements.
- * Debug your program by adding extra print statements.
- * Make your debugging print statements conditional.
- * Describe the basic concepts and operation of a debugger program.

WHAT IS DEBUGGING?

- ✧ The term *bug* is used to describe an error in your program that makes it produce incorrect results.
- ✧ Debugging is the process of finding the bugs in your program and fixing them.
- ✧ There are two general categories of bug:
 - Bugs caused by using the incorrect syntax in a program statement.
 - Bugs caused by using incorrect logic in designing and implementing your program.
- ✧ There are many different tools and techniques used to help programmers debug programs.

Back in 1947, when computers were as much mechanical as electronic, a computer at a Naval research station started producing incorrect results. A technician started looking through the maze of relays, switches and vacuum tubes, and found that a moth had found its way into the computer's circuitry, become caught in one of the relays and had died. This caused the relay to work incorrectly. When they "debugged" the computer — when they took the bug out — it started working correctly again. They taped the moth in the logbook with the entry "First actual case of bug being found." (The term "bug" for a mechanical or design defect had been in use at least since the days of Thomas Edison.)

So, "debugging" your computer program is the process of finding problems in your program and correcting them — getting the "bugs out."

There are two general types of bug: *syntax bugs* (compiler errors) that are caused by typing a statement into your program incorrectly, and *logic bugs* (runtime errors), in which the program compiles but does not produce the correct results when it runs.

Syntax bugs are fairly easy to find and, therefore, to fix. Logic bugs can be much more difficult. One technique is to very carefully read the text of your program's source code. Step through each statement, exactly as the computer would. Do exactly what each statement says and only what each statement says (not what you "know" the program is supposed to do). This is a skill every programmer must practice from time to time. For large programs, though, this technique can take too long. Therefore, many different tools and techniques have been developed over the years to aid in the debugging process.

COMMENTING OUT CODE

- * One of the oldest debugging techniques is to *comment out code*.
 - Locate where you think the problem is occurring in your program.
 - Put comment characters in front of or around the statements that might be causing the problem.
 - Recompile and reexecute the program.
 - Determine if your program is now working correctly.
 - Repeat the process until you've identified the statement causing the error.

* Example:

```
...
if (scale == 'C' || scale == 'c')
{
    celsius = temp;
    fahrenheit = (celsius * 1.8) + 32;
}
/* The following code is commented out
else
{
    // System.err.println("Invalid scale. Use 'F' or 'C'.");
    return;
}
*/
...
```

Commenting out code is probably the oldest debugging technique. It involves using the language's comment characters on statements that you would like to temporarily take out of your program to see if they are contributing to a bug.

Since the compiler ignores comments as it compiles your program, the commented lines will not be compiled. This allows you to keep the problem statements in your program without having to take them out temporarily and put them back in later.

```
...
if(scale == 'C' || scale == 'c')
{
    celsius = temp;
    fahrenheit = (celsius * 1.8) + 32;
}
// else
// {
//     System.err.println("Invalid scale. Use 'F' or 'C'.");
//     return;
// }
...
```

You then recompile the program, reexecute it, and see if the problem is still bugging your program. You continue following the same steps until you identify the problem statements.

Remember: in Java you cannot “nest” comments using the `/* */` style of commenting. This problem does not occur with the Java `//` comment style:

```
/* Trying to comment these two lines out
a = a + 3;    /* Add 3 to the base value */
sum = calcSum(a);
*/
```

The following will work a lot better:

```
/* Trying to comment these two lines out
a = a + 3;    // Add 3 to the base value
sum = calcSum(a);
*/
```

SIMPLE DEBUGGING WITH PRINT STATEMENTS

- * Another tried-and-true debugging technique involves putting extra print statements in your program.

- Print a message to indicate that execution control has reached some specific point in the program correctly.

```
amountDue = amountDue - payment;  
System.out.println("We got past the calculation");
```

- Print the contents of variables to see if they contain the values you expect at a particular point in your program.

```
amountDue = amountDue - payment;  
System.out.println("value: " + amountDue);
```

- * Put the word DEBUG, or something like it, in your debug statements so you can distinguish them from the program's normal output.

```
System.out.println("DEBUG: Got past calculation");  
System.out.println("DEBUG: amount due: " + amountDue);
```

- * Once the print statements have served their purpose, you remove them and recompile the program.

- Your program's users probably don't want to see a bunch of "DEBUG: " stuff.

Adding extra print statements is another debugging technique. It's quick, easy, and provides simple, useful information.

You may be wondering if your program ever reaches a certain point in the execution of its statements. Just add an extra print statement at that point in your program. If it prints, then it got there.

You may also be curious as to whether or not a variable contains an appropriate value when your program is executing. Do the same thing — add another print statement to print the contents of the variable just before it's used.

You should exercise a little bit of care when placing these debugging prints in your program. Understand that if you place a debugging print in a loop, it could print your debugging information many times. And, if you put your debugging print in a decision making statement (an `if`), it may or may not print, depending on the result of the conditional expression.

Example:

```
...
System.out.println("Debug: Ready to test temperature scale.");
if (scale == 'C' || scale == 'c')
{
    System.out.println("Debug: We're in celsius-to-fahrenheit.");
    celsius = temp;
    fahrenheit = (celsius * 1.8) + 32;
}
else
{
    System.out.println("Debug: We've found an error.");
    System.err.println("Invalid scale. Use 'F' or 'C'.");
    return;
}
System.out.println("Debug: Finished testing temperature scale.");
...
```

MAKING DEBUGGING PRINT STATEMENTS CONDITIONAL

- * An annoyance of debugging with print statements is:
 - When you're debugging, you put the statements in your program.
 - When you're done debugging, you take the extra statements out.
 - If you find a problem later on, you put them back in . . .
 - When you've fixed the problem, you take them back out . . .
 - If you find another problem later on, you put them back in . . .
 - Is this beginning to sound like a bit of a chore to you?
- * To fix this problem, put your debug print statements in a conditional statement:
 - Have the conditional check a static variable to determine whether to print the debug message.

Adding a static boolean variable to determine whether or not to print debug information is another common debugging technique.

```
private static boolean debug = true;

// Somewhere in the program...
if (debug == true)
    System.out.println("DEBUG: Ready to test scale.");
...
// Somewhere else...
if (debug == true)
    System.out.println("DEBUG: Value of somevar is:" + somevar);
...
```

Now you just have to change the static variable from `true` to `false` and recompile; none of the conditional debug statements will print.

Another technique is to define a debug method:

```
public static void debug(String msg){
    boolean debugging = true;
    if(debugging)
        System.out.println(msg);
}

// Somewhere in the program...
debug("Debug: Ready to test scale.\n");
...
```

You can keep the debug method definition in a separate file and recompile just that file to turn debugging on or off.

PROGRAMS THAT HELP YOU DEBUG PROGRAMS

- ✴ Some computer systems come with special software programs that can help you in the debugging process.
- ✴ These programs are called *debuggers*, and they help you follow the execution of your program without commenting out code or adding debugging print statements.
 - A debugger reads your executable program into memory.
 - It allows you to set breakpoints in your program where you want the program to stop while it's executing.
 - It can start the program running and let it go until it reaches a breakpoint.
 - When the program stops, you can examine the contents of a variable to see if it contains what it is supposed to contain.
 - You can then single-step through the statements in the program — watching how and when they execute — or continue execution to the next breakpoint.
- ✴ There are a wide variety of debuggers on most major systems.
 - The debugger is an important part of most Integrated Development Environment (IDE) products.

Usually, debuggers have their own language. They are interactive software systems like an operating system command interpreter or an editor. They prompt you for a command, you enter the debugging command to tell the debugger what you want to do, and then you start running the program using other commands.

Some concepts and terminology:

A *breakpoint* is a place in your program that you want the program to stop executing.

Single-stepping is the process of telling the debugger program that you want to execute the statements in your program one at a time and stop after each one.

A *trace* tells you what subroutines or functions have been called, in what order, and also what parameters have been passed.

Once your program has stopped executing, after reaching a breakpoint or while single-stepping, you can use debugger commands to display the contents of your variables.

Every debugger is different — they use different commands. But once you understand the debugging process, you'll find that they all operate in a similar manner.

EXAMPLE 11 - DEBUG STATEMENTS

SquareAndRootDebug.java

```
package examples;
import java.util.*;
public class SquareAndRootDebug {

    private static boolean debug;

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        debug = true;
        if (debug == true)
            System.out.println("Debug: The program is starting.");

        double aNum = getNumber(true, scanner);
        while (true) {
            if (debug == true)
                System.out.println("Debug: In main, aNum: " + aNum);

            calcSquareAndRoot(aNum);
            aNum = getNumber(false, scanner);
        }
    }

    public static double getNumber(boolean firstTime, Scanner sc) {

        String theResp = null;
        double theNum = 0.0;
        if (debug == true) {
            System.out.println("Debug: Have entered the getNumber"
                + " method.");
            System.out.println("Debug: Was passed theFirst: "
                + firstTime);
        }

        if (firstTime == false) {
            System.out.print("Would you like to do another? (y/n): ");
            theResp = sc.next();

            if (debug == true)
                System.out.println("Debug: In getNumber, theResp: "
                    + theResp);
        }
    }
}
```

```
        if (theResp.charAt(0) == 'n' || theResp.charAt(0) == 'N') {
            sc.close();
            System.exit(0);
        }
    }

    System.out.print("Please enter a number: ");
    theNum = sc.nextDouble();

    if (debug == true)
        System.out.println("Debug: In getNumber, theNum: "
            + theNum);

    return theNum;
}

public static void calcSquareAndRoot(double theNum) {

    if (debug == true) {
        System.out.println("Debug: Have entered"
            + " calcSquareAndSquareRoot method.");
        System.out.println("Debug: Was passed theNum: " + theNum);
    }

    System.out.println("You entered the number: " + theNum);
    System.out.println("\tThe square is: " + square(theNum));
    System.out.println("\tThe square root is: "
        + Math.sqrt(theNum));
}

public static double square(double x) {
    double result = 0.0;

    if (debug == true) {
        System.out.println("Debug: Have entered the square"
            + " method.");
        System.out.println("Debug: Was passed x: " + x);
    }

    result = x * x;

    if (debug == true)
        System.out.println("Debug: In square, result: " + result);

    return result;
}
}
```

REVIEW QUESTIONS

1. Explain the term *debugging*.
2. How do you "comment out" statements in your program? How does that help you debug the program?
3. How can adding extra print statements to your program help you debug?
4. What is useful about making your debugging print statements conditional?
5. What is a debugger?

LABS

- ❶ In the files that accompany this course for this chapter is a program in a file called *BuggyProg.java*. The program has had several bugs introduced in it. Use your newly acquired debugging techniques to fix this program and make it run correctly.
(Solution: *FixBuggyProg.java*)
- ❷ Find your Fahrenheit/Celsius conversion program from a previous chapter and try commenting out sections of code to see how this debugging technique might be used.
- ❸ Find your "next largest even multiple" program from a previous chapter and try adding some debugging print statements to see how this debugging technique might be used.
(Solution: *DebugNextMult.java*)
- ❹ Modify the program from ❸ and conditionalize your debugging print statements.
(Solution: *DebugNextMult2.java*)

APPENDIX A - ADDITIONAL EXERCISES

MAKING CHANGE

- ① Write a program that accepts an amount due and an amount tendered, and calculates the change due.
(Solution: *GiveChange.java*)
- ② Have your program calculate the number of each type of coin necessary to make the change.
Hint: Integer division and the modulus operator will help.
(Solution: *GiveChangeCoins.java*)

INDEX

SYMBOLS

53
% 102
() 110
* 44, 53, 102
+,-,*,/,% 102
.class 25
.java 21
/* 52, 53
// 52
; 21
<,<=,>,>= 104
= 105
== 105
\" 82
\\ 82
\\b 82
\\n 82
\\t 82, 83
{ } 129

A

addition 102
algorithm 19
API 179
Application Programming Interface 179
application programs 26
arguments 172
arithmetic expression 98, 102
arithmetic skills 12
array 188, 189, 190
array declaration 190
array elements 192
array length 191
array size 191
ASCII character set 83
Assembler 30, 55
assignment 70, 71
assignment expression 98
assignment operator 105
assignment statements 196
associativity 110, 111
Awk 31, 53

B

backslash 82
backspace 83
bel 83
binary digits 63
bit 62, 63
bit pattern 63, 81, 83
block 128, 130, 146, 147, 151, 166
boolean 104
braces 129, 131, 147, 149
break 133, 152, 153
break statement 152
breakpoints 214
bug 206
byte 62
byte code 24, 25

C

C 11, 30, 31, 53, 55
C++ 11, 21, 30, 31, 55, 193
calling a method 168
case 132, 133
case statement 123
case-sensitive 67
char 66
character data 64
character literal 69
class file 24
COBOL 30, 53, 55, 168
code block 129
collections 188
comment out code 208
commenting 209
comments 52, 208, 209
compilation system 34
compile 22
compiler 32
compiler error 32, 207
computational procedure 19
computer program 18
computer programming 19
conditional 48, 122
conditional expression 124
conditional loop 142, 150
consistent style 55

control statement 142
core 63
"curly braces" 129
current working directory 25

D

data type 64
debug 22
debug statements 210
debuggers 34, 214
debugging 32
decision making 48, 122, 123
declaring a variable 67
decrement 144
default 132
defining a variable 67
development environment 34
digit 83
division 102
double 66
double-precision floating point 64
duplicating code 164

E

editor 34
elements, array 190
else 131
end of array 192
end of line 46
escape sequences 82
executable code 23
execute 22
experience 12
expression 98
 evaluation 100

F

fall through 133
false 153
files 12
 created during compilation 24
flarp 67
float 66
floating point data 64
flow chart 125
flow of execution 122
for loop 144
formatted output 46
formatting 84
Fortran 30, 53, 55, 168
free-format languages 54

function 166, 169

G

global variable 170

H

"Hello, world!" 20
Hello.java 21
horizontal tab 83

I

IDE 34, 214
if statement 122, 123, **124**, 131
increment 144
indentation 54, 125, 126, 147, 149
Index 227
index, array 190, 192, 194
infinite loop 152, 153
Information Systems 61
Information Technology 61
Information Theory 61
input 28, 42
Input - Process - Output 28, 47
input validation 88
instructions 18
int 66
integer 64, 102
integer data 64
Integrated Development Environment 34
iteration 50
iterative loop 142, 144

J

Java 11, 30, 31, 168, 193
Java for loop 145
Java Virtual Machine 24
Java while loop 151

K

keyboard input 86, 88
keywords 67

L

lather 143
left-hand side 70
lhs 70
libraries 178
library 178
line feed 83
literal 68

local variables 170, 171
 parameters as 172
 logic 18, 206, 207
 logical and 108
 logical expression 98, 108
 logical operators 109
 logical or 108
 logical thinking 12
 loop 106, 142
 loop body 146, 151
 loop control variable 144, 148
 looping 51
 loops 192

M

main memory 63
 main() method 121, 168
 Management Information Systems 61
 memory 62
 memory location 18
 method 169
 method call 174
 method definition 172
 method stub 176
 modular applications 166
 modulus 102
 moth 207
 MS-DOS 53
 multidimensional arrays 194
 multiplication 44, 102

N

nested control statements 130
 nested loop 148
 newline 84
 nul byte 83
 numerals 81
 numeric calculation 44

O

object-oriented 21
 operands 98
 operators 98, 100, 103
 order of evaluation 110
 output 28, 80

P

parameters 172
 parentheses 110
 Perl 31, 53, 193

PL/1 30
 precedence 110, 111
 precedence tables 111
 print statements 210
 printable character 83
 procedure 166
 process 28
 programming 20
 programming language 30, 35
 programming style 54
 programs
 application 26
 system 26
 utility 26
 prompting the user 42, 88

R

RAM 63
 relational expression 98, 104, 122, 124, 150
 relational operator 105
 rem 53
 remainder 102
 repetitive tasks 164
 return 174
 return statement 174
 return value 174
 rhs 70
 right-hand side 70
 rinse 143
 row and column numbers 194
 run-time errors 32
 runtime error 207

S

scalar variable 188, 189
 scope 170
 semicolon 21
 sequential execution 120
 Shell 53
 single-stepping 215
 source code 23, 43
 source code file 24
 special characters 82
 square brackets 191
 Squares.java 43
 standard error 80
 "standard in" 86
 standard input 86
 "standard out" 80
 standard output 80
 start of execution 168

- steps, in writing programs 22
- string literal 69
- structure 188, 189
- stubs 176
- subexpressions 102
- subtraction 102
- switch 133
- switch statement 123, 132
- syntax 206, 207
- system programs 26
- System.out 81

T

- text editor 23
- “then clause” 124
- Thomas Edison 207
- three-dimensional arrays 194
- trace 215
- true 153
- two-dimensional array 194, 196
- typing skills 12

U

- unary operators 102
- Unix 53
- utility programs 26

V

- variable 42, 44, 66, 72
- vendors 34
- visibility 170

W

- while loop 150
- writing a program 22

Z

- zero-based arrays 193

Skill Distillery

7400 E. Orchard Road, Suite 1450 N
Greenwood Village, Colorado 80111
303-302-5234
www.SkillDistillery.com