*Introduction to Java: Programming and Data Structures*

**Chapter 1: Introduction to Computers, Programs, and Java**

1.5 – Java, The World Wide Web, and Beyond

Java is simple, object oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high performance, multithreaded and dynamic.

1.6 – The Java Language Specification, API, JDK, JRE, and IDE

The application program interface (API), also known as library, contains predefined classes and interfaces for developing Java programs.

Java comes in three editions:

* Java Standard Edition (Java SE) to develop client-side applications. The applications can run on a desktop.
* Java Enterprise Edition (Java EE) to develop server-side applications, such as Java servlets, JavaServer Pages (JSP), and JavaServer Faces (JSF).
* Java Micro Edition (Java ME) to develop applications for mobile devices, such as cell phones.

Oracle releases each version with a Java Development Toolkit (JDK). The JDK consists of a set of separate programs, each invoked from a command line, for compiling, running, and testing Java programs. The program for running Java programs is known as JRE (Java Runtime Environment).

Instead of using the JDK, you can use a Java development tool (e.g. IntelliJ, NetBeans)—software that provides an integrated development environment (IDE) for developing Java programs quickly.

1.7 A Simple Java Program

A Java program is executed from the ***main*** method in the class.

(The word console is an old computer term that refers to the text entry and display device of a computer. Console input means to receive input from the keyboard, and console output means to display output on the monitor.)

\*Reference Welcome.Java

Line 1 defines a class. Every Java program must have at least one class. Each class has a name. By convention, class names start with an uppercase letter.

Line 2 defines the main method. The program is executed from the main method. A class may contain several methods. The main method is the entry point where the program begins execution. A method is a construct that contains statements. Every statement in Java ends with a semi-colon (;), known as the statement terminator.

String is a programming term meaning a sequence of characters. A string must be enclosed in double quotation marks.

Reserved words, or keywords, have specific meaning to the compiler and cannot be used for other purposes in the program.

Line 3 is a comment that documents what the program is and how it is constructed. Comments help programmers to communicate and understand the program. They are not programming statements, and thus are ignored by the compiler. In Java, comments are preceded by two slashes (//) on a line, called a line comment, or enclosed between /\* and \*/ on one or several lines, called a block comment or a paragraph comment. When the compiler see //, it ignores all text after // on the same line. When it see /\*, it scans for the next \*/ and ignores any text between /\* and \*/. In addition to line comments and block comments Java supports comments of a special type, referred to as Javadoc comments. Javadoc comments begin with /\*\* and end with \*/. They can be extracted into an HTML file using the JDK’s Javadoc command. Use Javadoc comments for commenting on an entire class or an entire method. These comments must precede the class or the method header in order to be extracted into a Javadoc HTML file. For commenting on steps inside a method, use line comments.

A pair of braces in a program forms a block that groups the program’s components. In Java, each block begins with an opening brace ({) and ends with a closing brace (}). Every class has a class block that groups data and methods of the class. Similarly, every method has a method block that groups the statements in the method. Blocks can be nested, meaning one block can be placed within another. Whenever you type an opening braces, immediately type a closing brace to prevent the missing-brace error.

The most common errors you will make as you learn to program will be syntax errors. Like any programming language, Java has its own syntax, and you need to write code that conforms to syntax rules.

Table 1.2 Special Characters

|  |  |  |
| --- | --- | --- |
| Character | Name | Description |
| {} | Opening and closing braces | Denote a block to enclose statements. |
| () | Opening and closing parenthesis | Used with methods. |
| [] | Opening and closing brackets | Denote an array. |
| // | Double slashes | Precede a comment line. |
| “” | Opening and closing Quotation marks | Enclose a string (i.e. sequence of characters). |
| ; | Semicolon | Mark the end of a statement. |

The print method is identical to the println method except that println moves to the beginning of the next line after displaying the string, but print does not advance to the next line when completed.

1.8 Creating, Compiling, and Executing a Java Program

You save a Java program in a .java file and compile it into a .class file. The .class file is executed by the Java Virtual Machine (JVM).

You have to create your program and compile it before it can be executed. This process is repetitive. If your program has compile errors, you have to modify the program to fix them, then recompile it. If your program has runtime errors or does not produce the correct result, you have to modify the program, recompile it, and execute it again.

You can use any text editor or IDE to create and edit a Java source-code file. From the command window, you can use a text editor such as notepad to create the Java source-code file. The source file must end with the extension .java and must have the same exact name as the public class name.

If there aren’t any syntax errors, the compiler generates a bytecode file with a .class extension. The Java language is a high-level language, but Java bytecode is a low-level language. The bytecode is similar to machine instructions but is architecture neutral and can run on any platform that has a Java Virtual Machine (JVM). Rather than a physical machine, the virtual machine is a program that interprets Java bytecode. This is one of the primary advantages of Java: Java bytecode can run on a variety of hardware platforms and operating systems. To execute a Java program is to run the program’s bytecode. You can execute the bytecode on any platform with a JVM, which is an interpreter. It translates the individual instructions in the bytecode into the target machine language code one at a time, rather, than the whole program as a single unit. Each step is executed immediately after it is translated.

Caution – Do not use the extension .class in the command line when executing the program. Use java ClassName to run the program.

When executing a Java program, the JVM first loads the bytecode of the class to memory using a program called the class loader. If your program uses other classes, the class loader dynamically loads them just before they are needed. After a class is loaded, the JVM uses a program called the bytecode verifier to check the validity of the bytecode and to ensure that the bytecode does not violate Java’s security restrictions.

*1.9 Programming style and Documentation*

1.9.2 Proper Indentation and Spacing

A consistent indentation style makes programs clear and easy to read, debug, and maintain. Indentation is used to illustrate the structural relationships between a program’s components or statements. Java can read the program even if all of the statements are on the same long line, but humans find it easier to read and maintain code that is aligned properly. Indent each subcomponent or statement at least two spaces more than the construct within which it is nested.

1.10 Programming Errors

Programming errors can be categorized into three types: syntax errors, runtime errors, and logic errors.

1.10.1 Syntax errors

Errors detected by the compiler are called syntax errors or compile errors. Syntax errors result from errors in code construction, such as mistyping a keyword, omitting some necessary punctuation, or using an opening brace without a closing brace. These errors are usually easy to detect because the compiler tells you where they are and what caused them. Since a single error will often display many lines of compile errors, it is good practice to fix errors from the top line and work downward. Fixing errors that occur earlier in the program may also fix additional errors that occur later.

1.10.2 Runtime Errors

Runtime Errors are errors that cause a program to terminate abnormally. They occur while a program is running if the environment detects an operation that is impossible to carry out. Input mistakes typically cause runtime errors. An input error occurs when the program is waiting for the user to enter a value, but the user enters a value the program cannot handle. For instance, if the program expects to read a number, but the user enters a string, this causes data-type errors to occur in the program.

1.10.3 Logic Errors

Logic errors occur when a program does not perform the way it was intended to. Errors of this kind occur for many different reasons. In general, syntax errors are easy to find and easy to correct because the compiler gives indications as to where the errors came from and why they are wrong. Runtime errors are not difficult to find, either, since the reasons and locations for the errors are displayed on the console when the program aborts. Finding logic errors, on the other hand, can be very challenging.

**Chapter 2: Elementary Programming**

2.2 Writing a Simple Program

Writing a program involves designing algorithms and translating algorithms into programming instructions or code. An algorithm lists the steps you can follow to solve a problem. Algorithms can help the programmer plan a program before writing it in a programming language. Algorithms can be described in natural languages or in pseudocode (natural language mixed with some programming code).

When you code—that is, when you write a program—you translate an algorithm into a program.

A variable represents a value stored in the computer’s memory. To let the compiler, know what the variable is specify its data type. That is the kind of data stored in a variable, whether an integer, real number, or something else. This is known as declaring variables. Java provides simple data types for representing integers, real numbers, characters, and Boolean types. These types are known as primitive data types or fundamental types.

Real numbers (i.e. numbers with a decimal point) are represented using a method known as floating-point in computers. Therefore, the real numbers are also called floating-point numbers. In Java, you can use the keyword double to declare a floating-point variable.

Variables correspond to memory locations. Every variable has a name, a type, and a value. The value is not defined until you assign a value.

Reviewing how a program works is called tracing a program. Tracing programs are helpful for understanding how programs work, and they are useful tools for finding errors in programs.

The plus sign (+) has two meanings: one for addition, and the other for concatenating (combining) strings. A string concatenation operator combines two strings into one. If a string is combined with a number, the number is converted into a string and concatenated with the other string.

Caution – A string cannot cross lines in the source code and would result in a compile error.

2.3 Reading Input from the Console

Reading input from the console enables the program to accept input from the user.

Java uses System.out to refer to the standard output device, and System.in to the standard input device. By default, the output device is the display monitor, and the input device is the keyboard. To perform console output, you simply use the println method to display a primitive value or a string to the console. To perform console input, you need to use the Scanner class to create an object to read input from System.in, as follows:

Scanner input = new Scanner(System.in);

The syntax new Scanner(System.in) creates an object of the Scanner type. The syntax Scanner input declares that input is a variable whose type is Scanner. The whole line Scanner input = new Scanner(System.in) creates a Scanner object and assigns its reference to the variable input. An object may invoke its methods. To invoke a method on an object is to ask the object to perform a task. You can invoke the nextDouble() method to read a double value as follows:

Double radius = input.nextDouble();

This statement reads a number from the keyboard and assigns the number to radius.

The Scanner class is in the java.util package. Note the import statement can be omitted if you replace Scanner by java.util.Scanner.

A prompt is something that directs the user to enter an input. Your program should always tell the user what to enter when expecting input from the keyboard.

There are two types of import statements: specific import and wildcard import. The specific import specifies a single class in the import statement. For example, the following statement imports the Scanner from package java.util.

Import java.util.Scanner;

The wildcard import imports all the classes in a package by using the asterisk as the wildcard. For example, the following statement imports all the classes from the package java.util:

Import java.util.\*;

The information for the classes in an imported package is not read in at compile time or runtime unless the class is used in the program. The import statement simply tells the compiler where to locate the classes. There is no performance difference between a specific import and a wildcard import declaration.

2.4 Identifiers

Identifiers are the names that identify the elements such as classes, methods, and variables in a program.

All identifiers must obey the following rules:

* An identifier is a sequence of characters consisting of letter, digits, underscores, and dollar signs.
* An identifier must start with a letter, and underscore, or a dollar sign. It cannot start with a digit.
* An identifier cannot be a reserved word.
* An identifier cannot be true, false, or null.
* An identifier can be of any length.

The java compile detects illegal identifiers and reports syntax errors. Also, since Java is case sensitive, area, Area, and AREA are all different identifiers. Identifiers are for naming variables, methods, classes, and other items in a program. Descriptive identifiers make programs easy to read. Avoid using abbreviation for identifiers. Using complete words is more descriptive. Do not name identifiers with the $ character. By convention, the $ character should be used only in mechanically generated source code.

2.5 Variables

Variables are used to represent values that may be changed in the program. They are called variables because their values can be changed. Variables are for representing data of a certain type. To use a variable, you declare it by telling the compiler its name as well as what type of data it can store. The variable declaration tells the compiler to allocate appropriate memory space for the variable based on its data type. The syntax for declaring a variable is:

Datatype variableName;

If variables are of the same type, they can be declared as follows:

Datatype variable1, variable2, …, variableN

Variables often have initial values. You can declare a variable and initialize it in one step. Consider the following code:

Int count = 1;

This is equivalent to the next two statements:

Int count;

Count = 1;

You can use a shorthand form to declare and initialize variables of the same type together. For example,

Int I =1, j = 2;

A variable must be declared before it can be assigned a value. A variable declared in a method must be assigned a value before it can be used.

Whenever possible, declare a variable and assign its initial value in one step. This will make the program easy to read and avoid programming errors.

Every variable has a scope. The scope of a variable is the part of the program where the variable can be referenced.

2.6 Assignment Statements and Assignment Expressions

An assignment statement designates a value for a variable. An assignment statement can be used as an expression in Java.

After a variable is declared, you can assign a value to it by using an assignment statement. In Java, the equal sign (=) is used as the assignment operator. The syntax is as follows:

Variable = expression;

An expression represents a computation involving values, variables, and operators that, taking them together, evaluates to a value. You can use a variable in an expression. A variable can also be used in both sides of the = operator. For example:

X = x + 1;

In this assignment statement, the result of x + 1 is assigned to x. If x is 1 before the statement is executed, then it becomes 2 after the statement is executed. To assign a variable, you must place the variable name to the left of the assignment operator.

In java, an assignment statement is essentially an expression that evaluates to the value to be assigned to the variable on the left side of the assignment operator. For this reason, an assignment statement is also known as an assignment expression. For example:

System.out.println(x=1);

Which is equivalent to:

x=1;

System.out.println(x);

2.7 Named Constants

A named constant is an identifier that represents a permanent value.

The value of a variable may change during the execution of a program, but a named constant, or simply constant, represents permanent data that never changes. A constant is also known as a final variable in Java. Here is the syntax for declaring a constant:

Final datatype CONSTANTNAME = value;

A constant must be declared and initialized in the same statement. The word final is a Java keyword for declaring a constant. By convention, all letters in a constant are in uppercase. There are three benefits of using constants: (1) you don’t have to repeatedly type the same value if it used multiple times; (2) if you have to change the constant value you need to change it only in a single location in the source code; and (3) a descriptive name for a constant makes the program easy to read.

2.8 Naming Conventions

Sticking with the Java naming conventions makes your program easy to read and avoids errors.

Make sure you choose descriptive names with straightforward meanings for the variables, constants, classes, and methods in your program. Listed below are the conventions for naming variables, methods, and classes.

* Use lowercase for variables and methods. If the name consists of several words, concatenate them into one, making the first word lowercase and capitalizing the first letter of each subsequent word. This naming style is known as camelCase.
* Capitalize the first letter of each word in a class name
* Capitalize every letter in a constant, and use underscores between words.

Caution: Do not choose class names that are already used in the Java library.

2.9 Numeric Data Types and Operations

Java has six numeric types for integers and floating-point numbers with operators +, -,\*,/,and %.

2.9.1 Numeric Types

Every data type has a range of values. The compiler allocates memory space for each variable or constant according to its data type. Java provides eight primitive data types for numeric values, characters, and Boolean values.

Java uses four types of integers: byte, short, int, and long. Choose the type that is most appropriate for your variable. Java uses two types for floating-point numberds: float and double. The double type is twice as big as float, so double is known as double precision, and float as single precision. Normally, you should use the double type, because it is more accurate than the float type.

2.9.2 Reading Numbers from the Keyboard

You know how to use the nextDouble() method in the Scanner class to read a double value from the keyboard. You can also use the methods listed in table 2.2 to read a number of the byte, short, int, long, and float type.

Table 2.2 Methods for Scanner Objects

|  |  |
| --- | --- |
| Method | Description |
| nextByte() | Reads an integer of the byte type. |
| nextShort() | Reads an integer of the short type |
| nextInt() | Reads an integer of the int type. |
| nextLong() | Reads an integer of the long type. |
| nextFloat() | Reads a number of the float type. |
| nextDouble() | Reads a number of the double type. |

If you enter a value with an incorrect range or format, a runtime error would occur.

2.9.3 Numeric Operators

The operators for numeric data types include the standard arithmetic operators: addition, subtraction, multiplication, and division, and remainder. The operands are the values operated by an operator, When both operands of a division are integers, the result of the division is the quotient and the fractional part is truncated. To perform a floating-point division, one of the operands must be a floating-point number. The % operator, known as remainder, yields the remainder after division. The operand on the left is the dividend, and the operand on the right is the divisor. The % operator is often used for positive integers, but it can also be used with negative integers and floating-point values. The remainder is negative only if the dividend is negative. The + and – operators can be both unary and binary. A unary operator has only one operand; a binary operator has two. For example, the – operator in -5 is a unary operator to negate number 5, whereas the – operator in 4-5 is a binary operator for subtracting 5 from 4.

2.9.4 Exponent Operations

The Math.pow(a,b) method can be used to compute a^b. The pow method is defined in the Math class in the Java API. You invoke the method using the syntax Math.pow(a,b), which returns the result a^b. Here, a and b are parameters for the pow method.

2.10 Numeric literals

A literal is a constant value that appears directly in a program.

2.10.1 Integer Literals

An integer literal can be assigned to an integer variable as long as it can fit into the variable. A compile error will occur if the literal is too large for the variable to hold. An integer literal is assumed to be of the int type, whose value is between -2^31 and 2^31. To denote an integer literal of the long type, append the letter L to it. For example, to write integer 2147483648 in a Java program, you have to write it as 2147483648L, because it exceeds the range for the int value.

2.10.2 Floating-Point Literals

Floating-point literals are written with a decimal point. By default, a floating-point literal is treated as a double type value. You can make a number a float by appending the letter f or F, and you can make a number a double by appending the letter d or D. A float value has 7-8 numbers of significant digits, and a double value has 15-17 numbers of significant digits.

2.10.3 Scientific Notation

Floating-point literals can be written in scientific notation in the form of aX10^b. A special syntax is used to write scientific notation numbers. For example, 1.23456X10^2 is written as 1.23456E2 or 1.23456E+2 and 1.23456E-2(or e) represents an exponent, and can be in either lowercase or uppercase.

2.11 Evaluating Expressions and Operator Precedence

Java expressions are evaluated in the same way as arithmetic expressions.

Writing a numeric expression in Java involves a straightforward translation of an arithmetic expression using Java operators. Although Java has its own way to evaluate an expression behind the scene, the result of a Java expression and its corresponding arithmetic expression is the same. Therefore, you can safely apply the arithmetic rule for evaluating a Java expression. Operators contained within pairs of parentheses are evaluated first. When more than one operator is used in an expression, the following operator precedence rule is used to determine the order of the evaluation:

* Multiplication, division, and remainder operators are applied first. If an expression contains several multiplication, division, and remainder operators, they are applied from left to right.
* Addition and subtraction operators are applied last. If an expression contains several addition and subtraction operators, they are applied from left to right.

2.13 Augmented Assignment Operators

The operators +, -, \*, /, and % can be combined with the assignment operator to form augmented operators.

Very often, the current value of a variable is used, modified, then reassigned back to the same variable. Java allows you to combine assignment and addition operators using an augmented (or compound) assignment operator. The += is called the addition assignment operator. There are other assignment operators.

Table 2.4 Augmented Assignment Operators

|  |  |  |  |
| --- | --- | --- | --- |
| Operator | Name | Example | Equivalent |
| += | Addition assignment | j += 8 | j = j + 8 |
| -= | Subtraction assignment | j -= 8 | J = j – 8 |
| \*= | Multiplication assignment | J \*= 8 | J = j \* 8 |
| /= | Division assignment | J /= 8 | J = j / 8 |
| %= | Remainder assignment | J%=8 | J = j%8 |

The augmented assignment operator is performed last after all the other operators in the expression are evaluated.

Caution – There are no spaces in the augmented assignment operators.

Note: Like the assignment operator (=), the operators (+=,-=,\*=,/=,%=) can be used to form an assignment statement as well as an expression.

2.14 Increment and Decrement Operators

The increment operator (++) and decrement operator (--) are for incrementing and decrementing a variable by 1.

The ++ and – are two shorthand operators for incrementing and decrementing a variable by 1. These operators are known as postfix (or postincrement) and postfix decrement (or postdecrement), because the operators ++ and – are placed after the variable. These operators can also be placed before the variable. These operators are known as prefix increment ( preincrement) and prefix decrement (or predecrement).The effect of h++ and ++h are the same in the preceding examples. However, their effects are different when they are used in statements that do more than just increment and decrement.

Table 2.5 Increment and Decrement Operators

|  |  |  |  |
| --- | --- | --- | --- |
| Operator | Name | Description | Example (assume i = 1) |
| ++var | Preincrement | Increment var by 1, and use the new var value in the statement | In j = ++i  //j is 2, i is 2 |
| Var++ | Postincrement | Increment var by 1, but use the original var value in the statement | Int j = i++  //j is 1, i is 2 |
| --var | Predecrement | Decrement var by 1, and use the new var value in the statement. | Int j = --1;  //j is 0, I is 0 |
| Var-- | Postdecrement | Decrement var by 1, and use the original var value in the statement | Int j = i--;  //j is 1, i is 0 |

Operands are evaluated from left to right in Java. The left-hand operand of a binary operator is evaluated before any part of the right-hand operand is evaluated. This rule takes precedence over any other rules that govern expressions.

2.15 Numeric Type Conversions

Floating-point numbers can be converted into integers using explicit casting.

Can you perform binary operations with two operands of different types? Yes. If an integer and a floating-point number are involved in a binary operation, Java automatically converts the integer to a floating-point value.

You can always assign a value to a numeric value whose type supports a larger range of values; for instance, you can assign a long value to a float variable. You cannot, however, assign a value to a variable of a type with a smaller range unless use type casting. Casting is an operation that converts a value of one data type into a value of another data type. Casting a type with a small range to a type with a larger range is known as widening a type. Casting a type with a large range to a type with a smaller range is known as narrowing a type. Java will automatically widen a type, but you must narrow a type explicitly. The syntax for casting a type is to specify the target type in parenthesis, followed by the variable’s name or the value to be cast. For example (int) 1.7; Casting does not change the variable being cast. For example, d is not changed after casting in the following code:

double d = 4.5;

int I = (int)d; // i becomes d, but d is still 4.5

2.16 Software Development Process

The software development life cycle is a multistage process that includes requirements specification, analysis, design, implementation, testing, deployment, and maintenance.

Developing a software product is an engineering process. Software products, no matter how large or small, have a life cycle: requirements specification, analysis, design, implementation, testing, deployment, and maintenance.

At any stage of the software development life cycle, it may be necessary to go back to a previous stage to correct errors or deal with other issues that might prevent the software from functioning as expected.

Requirements specification is a formal process that seeks to understand the problem the software will address, and to document in detail what the software system needs to do. This phase involves close interaction between users and developers. Developers need to work closely with their customers and study the problem carefully to identify what the software needs to do.

System analysis seeks to analyze the data flow and to identify the system’s input and output. When you perform analysis, it helps to identify what the output is first, then figure out what input data you need in order to produce the output.

System design is to design a process for obtaining the output form the input. This phaser involves the use of many levels of abstraction to break down the problem into manageable components and design strategies for implementing each component. You can view each component as a subsystem that performs a specific function of the system. The essence of system analysis and design is input, process, and output (IPO).

Implementation involves translating the system design into programs. Separate programs are written for each component then integrated to work together. This phase requires the use of a programming language such as Java. The implementation involves coding, self-testing, and debugging.

Deployment makes the software available for use. Depending on the type of software, it may be installed on each user’s machine or installed on a server accessible on the Internet.

Maintenance is concerned with updating and improving the product. A software product must continue to perform and improve in an ever-evolving environment. This requires periodic upgrades of the product to fix newly discovered bugs and incorporate changes.

2.18 Common Errors and Pitfalls

Common elementary programming errors often involve undeclared variables, uninitialized variables, integer overflow, unintended integer division, and round-off errors.

Common Error 1: Undeclared/uninitialized variables and Unused variables

A variable must be declared with a type and assigned a value before using it. A common error is not declaring a variable or initializing a variable.

Co*mm*on Error 2: Integer Overflow

Numbers are stored with a limited numbers of digits. When a variable is assigned a value that is too large (in size) to be stored, it causes overflow. Java does not report warnings or errors on overflow, so be careful when working with integers close to the maximum or minimum range of a given type. When a floating-point number is too small to be stored, it causes underflow. Java approximates it to zero, so normally you don’t need to be concerned about underflow.

Common Error 3: Round-off Errors

A round-off error, also called a rounding error, is the difference between the calculated approximation of a number and its exact mathematical value. For example, 1/3 is approximately 0.333 if you keep three decimal places. Since the number of digits that can be stored is limited, round-off errors are inevitable. Calculations involving floating-point numbers are approximated because these numbers are not stored with compete accuracy.

Common Error 4: Unintended Integer Division

Java uses the same divide operator, namely /, to perform both integer and floating-point division. When two operands are integers, the / operator performs an integer division. The result of the operation is an integer. The fractional part is truncated. To force two integers to perform a floating-point division, make one of the integers into a floating-point number.

Common Pitfall 1: Redundant Input Objects

New programmers often write the code to create multiple input objects for each input. For example, the following code reads an integer and a double value:

Scanner input = new Scanner(System.in);

System.out.print(“Enter an integer: “);

int v1 = input.nextInt();

Scanner input1 = new Scanner(System.in);

System.out.print(“Enter a double value: “);

double v2 = input.nextDouble();

The code is not good. It creates two input objects unnecessarily and may lead to some subtle errors.

**Chapter 3: Selections**

3.1 Introduction

The program can decide which statements to execute based on a condition.

Like all high-level programming languages, Java provides selection statements: statements that let you choose actions with alternative courses. Selection statements use conditions that are Boolean expressions. A Boolean expression is an expression that evaluates to a Boolean value: true or false.

3.2 boolean Data Type

The Boolean data type declares a variable with the value either true or false.

Java provides six relational operators (also known as comparison operators), which can be used to compare two values

Table 3.1 Relational Operators

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Java Operator | Mathematics Symbol | Name | Example (radius is 5) | Result |
| < | < | Less than | Radius < 0 | False |
| <= | <= | Less Than or Equal to | Radius <= 0 | False |
| > | > | Greater than | Radius > 0 | True |
| >= | >= | Greater than or equal to | Radius >= | True |
| == | = | Equal to | Radius == 0 | False |
| != | Not = | Not equal to | Radius != 0 | True |

A variable that holds a Boolean value is known as a Boolean variable. The Boolean data type is used to declare Boolean variables. A Boolean variable can hold one of the two values: true of false. For example: Boolean lightsOn = true;

True and false are literals, just like the number 10. They are treated as reserved words and cannot be used as identifiers in the program.

3.3 if Statements

An if statement is a construct that enables a program to specify alternative paths of execution.

Java has several types of selection statements: one-way if statements, two-way if-else statements, nested if statements, multi-way if-else statements, switch statements, and conditional operators.

A one-way if statement executes an action if and only if the condition is true. The syntax for a one-way if statement is as follows:

If (Boolean-expression) {

Statement(s);

}

3.4 Two-Way if-else Statements

An if-else statement decides the execution path based on whether the condition is true or false.

A one-way if statement performs an action if the specified condition is true. If the condition is false, nothing is done. But what if you want to take alternative actions when the condition is false? You can use a two-way if-else statement. The actions that a two-way if-else statement specifies differ based on whether the condition is true or false. Here is the syntax for a two-way if-else statement:

If (Boolean-expression) {

Statement(s)-for-the-true-case

}

Else {

Statement(s)-for-the-false-case;

}

If the Boolean-expression evaluates to true, the statement(s) for the true case are executed; otherwise, the statement(s) for the false case are executed.

3.5 Nested If and Multi-Way if-else Statements

An if statement can be inside another if statement to form a nested if statement.

The statement in an if or if-else statement can be any Java be any legal Java statement, including another if or if-else statement. The inner if statement is said to be nested inside the outer if statement. The inner if statement can contain another if statement; in fact, there is no limit to the depth of nesting. In fact, Figure 3.3b is the preferred. coding style for multiple alternative if statements. This style, called multi-way if-else statements, avoids deep indentation and makes the program easy to read.

Figure 3.3b

If (score >= 90)

System.out.print(“A”);

Else if (score >= 80)

System.out.print(“B”);

Else if (score >= 70)

System.out.print(“C”);

Else if (score >= 60)

System.out.print(“D”);

Else

System.out.print(“F”);

3.6 Common Errors and Pitfalls

Forgetting necessary braces, ending an if statement in the wrong place, mistaking == for =, and dangling else clauses are common errors in selection statements. Duplicated statements in if-else statements and testing equality of double values are common pitfalls.

Common Error 1: Forgetting Necessary Braces

The braces can be omitted if the block contains a single statement. However, forgetting the braces when they are needed for grouping multiple statements is a common programming error. If you modify the code by adding new statements in an if statement without braces, you will have to insert the braces.

Common Error 2: Wrong semicolon at the if Line

Adding a semicolon at the end of an if line is a common mistake. This mistake is hard to find, because it is neither a compile error nor a runtime error; it is a logic error. This error often occurs when you use the next-line block style. Using the end-of-line block style can help prevent this error.

Common Error 3: Redundant Testing of Boolean Values

To test whether a Boolean variable is true or false in a test condition, it is redundant to use the equality testing operator. Instead, it is better to test the Boolean variable directly. Another reason for doing this is to avoid errors that are difficult to detect. Using the = operator instead of the == operator to compare the equality of two items in a test condition is a common error. This does not have compile errors. It assigns true to even, so even is always true.

Common Error 4: Dangling else Ambiguity

The code in (a) below has two if clauses and one else clause. Which if clause is matched by the else clause? The indentation indicates that the else clause matches the first if clause. However, the else clause actually matches the second if clause. That situation is known as the dangling else ambiguity. The else clause always matches the most recent unmatched if clause in the same block.

(a)

Int I = 1, j = 2, k = 3;

If (I > j) {

If (I > k) {

System.out.println(“A”);

}

} Else {//Should be indented

System.out.println(“B”);

}

Common Error 5: Equality Test of Two Floating-Point Values

Floating-point numbers have a limited precision and calculations; involving floating-point numbers can introduce round-off errors. Therefore, equality test of two floating-point values is not reliable. You cannot reliably test equality of two-floating point values. However, you can compare whether they are close enough by testing whether the difference of the two numbers is less than some threshold.

Common Pitfall 1: Simplifying Boolean Variable Assignment

3.10 Logical Operators

The Logical operators !, &&, ||, and ^ can be used to create a compound Boolean expression.

Sometimes, whether a statement is executed is determined by a combination of several conditions. You can use logical operators to combine these conditions to form a compound Boolean expression. Logical operators, also known as Boolean operators, operate on Boolean values to create a new Boolean value. The not (!) operator which negates true to false and false to true. The and (&&) operator of two Boolean operands is true if and only if both the operands are true. The or (||) operator of two Boolean operands is true if at least one of the operands is true. The exclusive or (^) operator of two Boolean operands is true if and only if the two operands have different Boolean values.

If one of the operands of an && operator is false, the expression is false; if one of the operands of an || operator is true, the expression is true. Java uses these properties to improve performance of these operators. When evaluating p1 && p2, Java first evaluates p1 then, if p1 ids true evaluates p2; if p2 is false, it does not evaluate p2. When evaluating p1 || p2, Java first evaluates p1, then if p1 is false, evaluates p2; if p1 is true, it does not evaluate p2. In programming terminology, && and|| are known as the short-circuit or lazy operators.

3.13 switch Statements

A switch statement executes statements based on a value of a variable or an expression.

The if statement in Listing 3.5, ComputeTax.java, makes selections based on a single true or false condition. There are four cases for computing taxes, which depend on the value of status. To fully account for all the cases, nested if statements were used. Overuse of nested if statements makes a program difficult to read. Java provides a switch statement to simplify coding for multiple conditions. You can write the following switch statement to replace the nested if statement in listing 3.5:  
 switch (status) {

Case 0: compute tax for single filers;

Break;

Case 1: compute tax for married jointly or qualifying widow(er);

Break;

Case 2: compute tax for married filing separately;

Break;

Case 3: compute tax for head of household;

Break;

Default: System.out.println(“Error: invalid status”);

System.exit(1);

}

This statement checks to see whether the status matches the value 0, 1, 2, 3, in that order. If matched, the corresponding tax is computed; if not matched, a message is displayed. Here is the full syntax for the switch statement:

Switch (switch-expression) {

Case value1: statement(s)1;

Break;

Case value2: statement(s)2:

Break;

…

Case valueN: statement(s)N:

Break;

Default: statement(s)-for-default

The switch statement observers the following rules:

* The switch-expression must yield a value of char, byte, short, int, or String type and must always be enclosed in parenthesis.
* The value1,…, and valueN must have the same data type as the value of the switch-expression. Note that value1,…, and valueN are constant expressions, meaning they cannot contain variables, such as 1 + x.
* When the value in a case statement matches the value of the switch-expression, the statements starting from this case are executed until either a break statement or the end of the switch statement is reached.
* The default case, which is optional, can be used to perform actions when none of the specified cases matches the switch-expression.
* The keyword break is optional. The break statement immediately ends the switch statement.

Caution: Do not forget to use a break statement when one is needed. Once a case is matched, the statements starting from the matched case are executed until a break statement or the end of the switch statement is reached. This is referred to as fall-through behavior.

3.14 Conditional Operators

A conditional operator evaluates an expression based on a condition.

You might want to assign a value to a variable that is restricted by certain conditions. From example, the following statement assigns 1 to y if x is greater than 0 and -1 if x is less than or equal to 0:

If (x > 0)

Y = 1;

Else

Y = -1

Alternatively, as in the following example, you can use a conditional operator to achieve the same result:

Y = (x > 0) ? 1 : -1;

The symbols ? and : appearing together is called a conditional operator (also known as a ternary operator because it uses three operands. It is the only ternary operator in Java. The conditional operator is in a completely different style, with no explicit if in the statement. The syntax to use the operator is as follows:

Boolean-expression ? expression1 : expression2

The result of this expression is expression1 if Boolean-expression is true; otherwise the result is expression2. Suppose you want to assign the larger number of variable num1 and num2 to max. You can simply write a statement using the conditional operator:

Max = (num1 > num2) ? num1 : num2;

For example, the following statement displays the message “num is even” if num is even, and otherwise displays “num is odd.”

System.out.println((num % 2 ==0) ? “num is even” : “num is odd”);

3.15 Operator Precedence and Associativity

Operator precedence and associativity determine the order in which operators are evaluated.

Suppose you have this expression:

3 + 4 \* 4 > 5 \* (4 + 3) – 1 && (4-3>5)

What is the value? What is the execution order of the operators?

The expression within parenthesis is evaluated first. When evaluating an expression without parentheses, the operators are applied according to the precedence rule and the associativity rule.

The precedence rule defines precedence for operators, which contains the operators you have learned so far. Operators are listed in decreasing order of precendence from top to bottom. The logical operators have lower precedence than the relational operators, and the relational operators haver lower precedence than the arithmetic operators. Operators with the same precedence appear in the same group.

Table 3.8 Operator Precedence Chart

|  |  |
| --- | --- |
| Precedence | Operator |
| top | Var++ and var—(Postfix) |
| | | +,- (Unary plus and minus), ++var and –var(Prefix) |
| | | (type)(Casting) |
| | | !(Not) |
| | | \*,/,% (Multiplication, division, and remainder) |
| | | +,-(binary addition and subtraction) |
| to | <,<=,>,>=(Relational) |
| | | ==,!= (Equality) |
| | | ^ (Exclusive OR) |
| | | && (AND) |
| | | || (OR) |
| Bottom | =,+=,-=,\*=,/=,%= (Assignment operators) |

If operators with the same precedence are next to each other, their associativity determines the order of evaluation. All binary operators except assignment operators are left associative.

3.16 Debugging

Debugging is the process of finding and fixing errors in a program.

Logic errors are called bugs. The process of finding and correcting errors is called debugging. A common approach to debugging is to use a combination of methods to help pinpoint the part of the program where the bug is located. You can hand-trace the program (catch errors by reading the program), or you can insert print statements in order to show the value of the variable or the execution flow of the program. These approaches might work for debugging short, simple program, but for a large, complex program, the most effective approach is to use a debugger utility.

JDK includes a command-line debugger, jdb, which is invoked with a class name. jdb is itself a Java program, running its own copy of Java interpreter. All the Java IDE tools, such as Eclipse and NetBeans, include integrated debuggers. The debugger utilities let you follow the execution of a program. They vary from one system to another, but they all support most of the following helpful features.

* Executing a single statement at a time: The debugger allows you to execute one statement at a time so that you can see the effect of each statement.
* Tracing into or stepping over a method: If a method is being executed, you can ask the debugger to enter the method and execute one statement at a time in the method, or you can ask it to step over the entire method. You should step over the entire method if you know that the method works,
* Setting breakpoints: You can also set a breakpoint at a specific statement. Your program pauses when it reaches a breakpoint. You can set as many breakpoints as you want. Breakpoints are particularly useful when you know where your programming errors starts. You can set a breakpoint at that statement, and have the program execute until it reaches the breakpoint.
* Displaying variables: The debugger lets you select several variables and display their values. As you trace through a program, the content of a variable is continuously updated.
* Displaying call stacks: The debugger lets you trace all of the method calls. This feature is helpful when you need to see a large picture of the program-execution flow.
* Modifying variables: Some debuggers enable you to modify the value of a variable when debugging. This is convenient when you want to test a program with different samples, but do not want to leave the debugger.

**Chapter 4: Mathematical Functions, Characters, and Strings**

4.2 Common Mathematical Functions

Java provides many useful methods in the Math class for performing common mathematical functions.

A method is a group of statements that performs a specific task. This section introduces other useful methods in the Math class. Service methods include rounding, min, max, absolute, and random methods. In addition to methods, the Math class provides two useful double constants PI and E (the base of natural logarithms). You can use these constants as Math.Pi and Math.E in any program.

4.2.1 Trigonometric Methods

The Math class contains the following methods listed in Table 4.1for performing trigonometric functions:

Table 4.1 Trigonometric Methods in the Math Class

|  |  |
| --- | --- |
| Method | Description |
| Sin(radians) | Returns the trigonometric sine of an angle in radians. |
| Cos(radians) | Returns the trigonometric cosine of an angle in radians. |
| Tan(radians) | Returns the trigonometric tangent of an angle in radians. |
| toRadians(degree) | Returns the angle in radians for the angle in degrees. |
| toDegrees(radians) | Returns the angle in degrees for the angle in radians. |
| Asin(a) | Returns the angle in radians for the inverse of sine. |
| Acos(a) | Returns the angle in radians for the inverse of cosine. |
| Atan(a) | Returns the angle in radians for the inverse of tangent. |

4.2.2 Exponent Methods

There are five methods related to exponents in the Math class as listed in Table 4.2.

Table 4.2 Exponent Methods in the Math Class

|  |  |
| --- | --- |
| Method | Description |
| Exp(x) | Returns e raised to power of x. |
| Log(x) | Returns the natural logarithm of x |
| Log10(x) | Returns the base 10 logarithm of x. |
| Pow(a,b) | Returns a raised to the power of b. |
| Sqrt(x) | Returns the square root of x. |

4.2.3 The Rounding Methods

The Math class contains four rounding methods.

Table 4.3 Rounding Methods in the Math class

|  |  |
| --- | --- |
| Method | Description |
| Cell(x) | X is rounded up to its nearest integer. This integer is returned as a double value. |
| Floor(x) | X is rounded down to its nearest integer. This integer is returned as a double value. |
| Rint(x) | X is rounded to its nearest integer. If x is equally close to two integers, the even one is returned as a double value. |
| Round(x) | Returns (int)Math.floor(x + 0.5) if x is a float and returns (long)Math.floor(x+0.5) if x is a double. |

4.2.4 The min, max, and abs Methods

The min and max methods return the minimum and maximum numbers of two numbers (int, long, float, or double). The abs method returns the absolute value of the number (int, long, float, or double).

4.2.5 The random Method

This method generates a random double value greater than or equal to 0.0 and less than 1.0. You can use it to write a simple expression to generate random numbers in any range. For example,

(int)(Math.random()\*10) -🡪 Returns a random integer between 0 and 9.

4.3 Character Data Type and Operations

A character data type represents a single character.

In addition to processing numeric values, you can process characters in Java. The character data type, char, is used to represent a single character. A character literal is enclosed in single quotation marks. Consider the following code:

char letter = ‘A’;

char numChar = ‘4’;

4.3.1 Unicode and ASCII code

Computers use binary numbers internally. A character is stored in a computer as a sequence of 0’s and 1’s. Mapping a character to its binary representation is called encoding. There are different ways to encode a character. How characters are encoded is defined by an encoding scheme.

Java supports Unicode, and encoding scheme established by the Unicode Consortium to support the interchange, processing, and display of written texts in the world’s diverse languages. Unicode was originally designed as a 16-bit character encoding. The primitive data type was intended to take advantage of this design by providing a simple data type that could hold an y character. However, it turned out that the 65,536 characters possible in a 16-bit encoding are not sufiecient to represent all the characters in the world. The Unicode standard therefore has been extended to allow up to 1,112,064 characters. Those characters are supplementary characters.

A 16-bit Unicode takes two bytes, preceded by \u, expressed in four hexadecimal digits that run from \u0000 to \uFFFF.

Most computers use ASCII (American Standard Code for Information Interchange), an 8-bit encoding scheme, for representing all uppercase and lowercase letters, digits, punctuation marks, and control characters. Unicode includes ASCII code, with \u0000 to \u007F corresponding to the 128 ASCII characters.

Table 4.4. ASCII Code for Commonly Used Characters

|  |  |  |
| --- | --- | --- |
| Characters | Code Value in Decimal | Unicode Value |
| ‘0’ to ‘9’ | 48 to 57 | \u0030 to \u0039 |
| ‘A’ to ‘Z’ | 65 to 90 | \u0041 to \u005A |
| ‘a’ to ‘z’ | 97 to 122 | \u0061 to \u007A |

You can use ASCII characters such as ‘X’, ‘1’, and ‘$’ in a Java program as well as Unicode’s.

Char letter = ‘A’;

Char letter = ‘\u0041’; // Character A’s Unicode is 0041

Both assign character A to the char variable letter.

4.3.2 Escape Sequences for Special Characters

Java uses a special notation to represent special characters. This special notation, called an escape sequence, consists of a backslash (\) followed by a character or a combination of digits.

Table 4.5 Escape Sequences

|  |  |  |  |
| --- | --- | --- | --- |
| Escape Sequence | Name | Unicode Code | Decimal Value |
| \b | Backspace | \u0008 | 8 |
| \t | Tab | \u0009 | 9 |
| \n | Linefeed | \u000A | 10 |
| \f | Formfeed | \u000C | 12 |
| \r | Carriage Return | \u000D | 13 |
| \\ | Backslash | \u005C | 92 |
| \” | Double Quote | \u0022 | 34 |

The backslash \ is called an escape character. It is a special character.

4.3.3 Casting between char and Numeric Types

A char can be cast into any numeric type, and vice versa. When an integer is cast into a char, only its lower 16-bits of data are used; the other part is ignored. When a floating-point value is cast into a char, the floating-point value is first cast into an int, which is then cast into a char.

When a char is cast into a numeric type, the character’s Unicode is cast into the specified numeric type.

Implicit casting can be used if the result of a casting fits into the target variable. Otherwise, explicit casting must be used. For example, since the Unicode of ‘a’ is 97, which is within the range of a byte, these implicit casting are fine:

byte b = ‘a’;

int I = ‘a’;

But the following statement is incorrect, because the Unicode \uFFF4 cannot fit into a byte:

byte b = ‘\uFF4’;

To force this assignment, use explicit casting, as follows:

byte b = (byte)’\uFFF4’;

Any positive integer between 0 and FFFF in hexadecimal can be cast into a character implicitly. Any number not in this range must be cast into a char explicitly. All numeric operators can be applied to char operands. A char operand is automatically cast into a number if the other operand is a number or a character. If the other operand is a string, the character is concatenated with the string. For example, the following statements

Int i = ‘2’ + ‘3’; //(int)’2’ is 50 and (int)’3’ is 51

System.out.println(“I is “ + i); //I is 101

Int j = 2 + ‘a’; //(int)’a’ is 97

System.out.println(“j is “ + j); // j is 99

System.out.println(j + : is the Unicode for character “) + (char)j); // 99 is the Unicode for character c

4.3.4 Comparing and Testing Characters

Two characters can be compared using the relational operators just like comparing two numbers. This is done by comparing the Unicodes of the two characters. For example,

‘a’ < ‘b’ is true because the Unicdoe for ‘a’ (97) is less than the Unicode for ‘b’ (98).

‘a’ < ‘A’ is false because the Unicode for ‘a’ (97) is greater than the Unicode for ‘A’ (65).

‘1’ < ‘8’ is true because the Unicode for ‘1’ (49) is less than the Unicode for ‘8’ (56).

Often in the program, you need to test whether a character is a number, a letter, an uppercase letter, or a lowercase letter. As given in Appendix B, the ASCII character set, that the Unicodes for lowercase letters are consecutive integers starting from the Unicode for ‘a’, then for ‘b’, ‘c’, …, and ‘z’. The same is true for the uppercase letters and for the numeric characters. This property can be used to write the code to test characters. For example, the following code tests whether a character ch is an uppercase letter, a lowercase letter, or a digital character:

If (ch >= ‘A’ && ch <= ‘Z’)

System.ouit.println(ch + “ is an uppercase letter”);

Else if (ch >= ‘a’ && ch <= ‘z’)

System.out.println(ch + “ is a lowercase letter”);

Else if (ch >= ‘0’ && ch <= ‘9’)

System.out.println(ch + “ is a numeric character”);

For convience, java provides the following methods in the Character class for testing characters as listed in Table 4.6. The Character class is defined in the java.lang package.

Table 4.6 Methods in the Character Class

|  |  |
| --- | --- |
| Method | Description |
| isDigit(ch) | Returns true if the specified character is a digit. |
| IsLetter(ch) | Returns true if the specified character is a letter. |
| isLetterOrDigit(ch) | Returns true if the specified character is a letter or a digit. |
| isLowerCase(ch) | Returns true if the specified character is a lowercase letter. |
| isUpperCase(ch) | Returns true if the specified character is a uppercase letter. |
| toLowerCase(ch) | Returns the lowercase of the specified character. |
| toUpperCase(ch) | Returns the uppercase of the specified character. |

4.4 The String Type

A string is a sequence of characters.

The char type represents only one character. To represent a string of characters, use the data type called String. String is a predefined class in the java library, just like the classes System and Scanner. The String type is not a primitive type. It is known as a reference type. Any java class can be used as a reference variable that references an object. Table 4.7 lists the String methods for obtaining string length, for accessing characters in the string, for concatenating string, for converting string to uppercases or lowercases, and for trimming a string.

Table 4.7 Simple Methods for String Objects

|  |  |
| --- | --- |
| Method | Description |
| Length() | Returns the number of characters in this string. |
| charAt(index) | Returns the character at the specified index from this string. |
| Concat(s1) | Returns a new string that concatenates this string with string s1 |
| toUpperCase() | Returns a new string with all letters in uppercase. |
| toLowerCase() | Returns a new string with all letters in lowercase. |
| Trim() | Returns a new string with whitespace characters trimmed on both sides. |

Strings are objects in Java. The methods listed above can only be invoked from a specific string instance. For this reason, these methods are called instance methods. A non-instance method is called a static method. A static method can be invoked without using an object. All the methods defined in the Math class are static methods. They are not tied to a specific object instance. The syntax to invoke an instance method is referenceVariable.methodName(arguments). A method may have many arguments or no arguments. Recall that the syntax to invoke a static method is ClassName.methodName(arguments).

4.4.1 Getting String Length

You can use the length() method to return the number of characters in a string. For example, the following code

String message = “Welcome to Java”;

System.out.println(“The length of “ + message + “ is “ + message.length());

Displays:

The length of Welcome to Java is 15

4.4.2 Getting Characters from a String

The s.charAt(index) method can be used to retrieve a specific character in a string s, where the index is between 0 and s.length()-1.

4.4.3 Concatenating Strings

You can use the concat method to concatenate two strings. The statement given below, for example3, concatenates strings s1 and s2 into s3:

String s3 = s1.concat(s2);

Because string concatenation is heavily used in programming, Java provides a convenient way to accomplish it. You can use the plus (+) operator to concatenate two strings, so the previous statement is equivalent to

String s3 = s1 + s2;

The following code combines the strings message, “ and “, and “HTML into one string:

String myString = message + “ and “ + “HTML”;

Recall that the + operator can also concatenate a number with a string. In this case, the number is converted into a string then concatenated. Note at least one of the operands must be a string in order for concatenation to take place. If one of the operands is a non-string, the non-string value is converted into a string and concatenated with the other string. If neither of the operands is a string, the plus sign (+) is the addition operator that adds two adds two numbers.

The augmented += operator can also be used for string concatenation. For example, the following code appends the string “ and Java is fun” with the string “Welcome to Java” in message.

Message += “ and Java is fun”;

So the new message is “Welcome to Java and Java is fun.”

4.4.4 Converting Strings

The toLowerCase() method returns a new string with all lowercase letters, and toUppercase() method returns a new string with all uppercase letters. For example,

“Welcome”.toLowerCase() returns a new string welcome.

“Welcome”.toUpperCase() returns a new string WELCOME.

The trim() method returns a new string by eliminating whitespace characters from both ends of the string. The characters “ “, \t, \f, \r, or \n are known as whitespace characters. For example,

“\t Good Night \n”.trim() returns a new string Good Night.

4.4.5 Reading a String from the Console

To read a string from the console, invoke the next() method on a Scanner object. For example, the following code reads three strings from the keyboard:

Scanner input = new Scanner(System.in);

System.out.print(“Enter three words separated by spaces: “);

String s1 = input.next();

String s2 = input.next();

String s3 = input.next();

System.out.println(“s1 is “ + s1);

System.out.println(“s2 is “ + s2);

System.out.println(“s3 is “ + s3);

The next() method reads a string that ends with a whitespace character. You can use the nextLine() method to read and entire line of text. The nextLine() method reads a string that ends with the Enter key pressed. For example, the following statements read a line of text:

Scanner input = new Scanner(System.in);

System.out.println(“Enter a line: “);

String s = input.nextLine();

Char ch = s.charAt(0);

System.out.println(“The character entered is “ + ch);

4.4.7 Comparing Strings

The String class contains the methods for comparing two strings.

Table 4.8 Comparison Methods for String Objects

|  |  |
| --- | --- |
| Method | Description |
| Equals(s1) | Returns true if this string is equal to string s1. |
| equalsIgnoreCase(s1) | Returns if this string is equal string s1; it is case insensitive. |
| compareTo(s1) | Returns an integer greater than 0, equal to 0, or less than 0 to indicate whether this string is greater than, equal to, or less than s1. |
| compareToIgnoreCase(s1) | Same as compareTo except that the comparision is case insensitive. |
| startsWith(prefix) | Returns true if this string starts with the specified prefix. |
| endsWith(suffix) | Returns true if this string ends with the specified suffix. |
| Contains(s1) | Returns true if s1 is a substring in this string. |

How do you compare the contents of two strings? You might attempt to use the == operator. However, the == operator checks only whether string1 and string2 refer to the same object; it does not tell you whether they have the same contents. Therefore, you cannot use the == operator to find out whether two string variables have the same contents. Instead, you should use the **equals** method. The following code, for instance, can be used to compare two strings:

If (string1.equals(string2))

System.out.println(“string1 and string2 have the same contents”);

Else

System.out.println(“string1 and string2 are not equal”);

The **compareTo** method can also be used to compare two strings. For example:

S1.comparesTo(s2)

The method returns the value 0 if s1 is equal to s2, a value less than 0 if s1 is lexicographically less than s2, and a value greater than if s1 is lexicographically greater than s2. The actual value returned from the compareTo method depends on the offset of the first two distinct characters in s1 and s2 from left to right. For example, suppose s1 is abc and s2 is abg, and s1.compareTo(s2) returns -4. The first two characters (a vs. a) from s1 and s2 are compared. Because they are equal, the second two characters (b vs. b) are compared. Because they are also equal, the third two characters (c vs g) are compared. Since the character c is 4 less than g, the comparison returns -4.

The String class also provides the equalsIgnoreCase and compareToIgnoreCase methods for comparing strings. The equalsIgnoreCase and compareToIgnoreCase methods ignore the case of the letters when comparing two strings. You can also use str.startsWith(prefix) to check whether string str starts with a specified prefix, str.endsWith(suffix) to check whether string str ends with a specified suffix, and str.contains(s1) to check whether string str contains string s1.

4.4.8 Obtaining Substrings

You can obtain a single character from a string using the charAt method. You can also obtain a substring from a string using the substring method in the String class.

Table 4.9 The String class contains the Methods for Obtaining Substrings

|  |  |
| --- | --- |
| Method | Description |
| Substring(beginIndex) | Returns this string’s substring that begins with the character at the specified beginIndex and extends to the end of the string. |
| substring(begingIndex, endIndex) | Returns this string’s substring that begins at the specified beginIndex and extends to the character at index endIndex – 1. Note the character at endIndex is not part of the substring. |

4.4.9 Finding a Character or a Substring in a String

The String class provides several versions of indexOf and lastIndexOf methods to find a character or a substring in a string.

Table 4.10 The String Class Contains the Methods for Finding Substrings

|  |  |
| --- | --- |
| Method | Description |
| Index of (ch) | Returns the index of the first occurrence of ch in the string. Returns -1 if not matched. |
| indexOf(ch, fromIndex) | Returns the index of the first occruence of ch after fromIndex in the string. Returns – 1 if not matched. |
| indexOf(s) | Returns the index of the first occurrence of String S in this string. Returns -1 if not matched. |
| indexOf(s, fromIndex) | Returns the index of the first occurrence of string S in this string after fromIndex. Returnds -1 if not matched. |
| lastIndex(ch) | Returns the index of the last occurrence of ch in the string. Returns -1 if not matche. |
| lastIndexOf(ch, fromIndex) | Returns the index of the last occurrence of ch before fromIndex in this string. Returns -1 if not matched. |
| lastIndexOf(s) | Returns the index of the last occurrence of string s. Returns -1 if not matched. |
| lastIndex(s, fromindex) | Returns the index of the last occurrence of string S before fromIndex. Returns -1 if not matched. |

4.4.10 Conversion between Strings and Numbers

You can convert numeric string into a number. To convert a string into an int value, use the Integer.parseInt method, as follows:

Int intValue = Integer.parseInt(intString);

Where intString is a numeric string such as “123”.

To convert a string into a double value, use the Double.parseDouble method, as follows:

Double doubleValue = Double.parseDouble(doubleString);

Where doubleString is a numeric string such as “123.45”.

If the string is not a numeric string, the conversion would cause a runtime error. The integer and double classes are both included in the java.lang package, and thus they are automatically imported. You can convert a number into a string; use the string concatenating operator as follows:

String s = number + “”;

4.6 Formatting Console Output

You can use the System.out.printf method to display formatted output on the console.

Often, it is desirable to display numbers in a certain format. The f in the printf stands for formatted, implying that the method prints an item in some format. The syntax to invoke this method is

System.out.printf(format, item1, item2, … , itemk);

Where format is a string that may consist of substrings and format specifiers.

A format specifier specifies how an item should be formatted. An item may be a numeric value, a character, a Boolean value, or a string. A simple format specifier consists of a percent sign (%) followed by a conversion code.

Table 4.11 Frequently Used Format Specifiers

|  |  |  |
| --- | --- | --- |
| Format Specifier | Output | Example |
| %b | A Boolean value | True or False |
| %c | A character | ‘a’ |
| %d | A decimal integer | 200 |
| %f | A floating-point number | 45.46000 |
| %e | A number in standard scientific notation | 4.556000e+01 |
| %s | A string | “java is Cool” |

Items must match the format specifiers in order, in number, and in exact type. By default, a floating-point value is displayed with six digits after the decimal point. You can specify the width and precision in format specifier.

Table 4.12 Examples of Specifying Width and Precision

|  |  |
| --- | --- |
| Example | Output |
| %5c | Output the character and add four spaces before the character item, because the width is 5. |
| %6b | Output Boolean value and add one space before the false value and two spaces before the true value. |
| %5d | Output the integer item with width 5. If the number of digits in the item is < 5, add spaces before the number. If the number of digits is > 5, the width is automatically increased. |
| %10.2f |  |
| %10.2e |  |
| %12s |  |

If an item requires more spaces than the specified width, the width is automatically increased. For example, the following code

System.out.printf(“%3d#%2s#%4.2f\n”, 1234, “Java”, 51.6653);

Displays

1234#Java#51.67

The specified width for int item 1234 is 3, which is smaller than its actual size 4. The width is automatically increased to 4. The specified width for string item Java is 2, which is smaller than its actual size 4. The width is automatically increased to 4. The specified width for double item 51.6653 is 4, but it needs width 5 to display 51.67, so the width is automatically increased to 5. You can display a number with comma separators by adding a comma in front of a number specifier. You can pad a number with leading zeros rather than spaces by adding a 0 in front if a number specifier. You can put the minus sign (-) in the format specifier to specify that the item is left justified in the output within the specified field.

Chapter 5: Loops

5.1 Introduction

A loop can be used to tell a program to execute statements repeatedly.

Java provides a powerful construct called a loop that controls how many times an operation or a sequence of operations is performed in succession. Using a loop statement, you can simply tell the computer to display a string a hundred times with having to code the print statement a hundred times.

Loops are constructs that control repeated executions of a block of statements. The concept of looping is fundamental to programming, Java provides three types of loop statements: while lops, do-while loops, and for loops.

5.2 The while Loop

A while loop executes statements repeatedly while the condition is true.

The syntax for the while loop is as follows:

While (loop-continuation-condition) {

//Loop body

Statement(s);

}

The part of the loop that contains the statements to be repeated is called the loop body. A one-time execution of a loop body is referred to as an iteration (or repetition) of the loop. Each loop contains a loop-continuation-condition, a Boolean expression that controls the executions of the body. It is evaluated each time to determine if the loop body is executed. If its evaluation is true, the loop body is executed; if its evaluation is false, the entire loop terminates and the program control turns to the statement that follows the while loop.

int count = 0; 🡨loop-continuation-condition

While (count < 100) {

System.out.println(“Welcome to Java!”);

Count++;

}

The loop for displaying Welcome to Java! A hundred times introduced in the preceding section is an example of a while loop.

The loop-continuation-condition is count < 100 and the loop body contains two statements. In this example, you know exactly how many times the loop body needs to be executed because the control variable count is used to count the number of iterations. This type of loop is known as a counter-controlled loop.

Note: The loop-continuation-condition must always appear inside the parentheses. The braces enclosing the loop body can be omitted only if the loop body contains one or no statement.

Here is another example to help understand how to a loop works.

Int sum = 0, I = 1;

While (I < 10) {

Sum = sum + I;

I++;

}

System.out.println(“sum is “ + sum); //sum is 45

If I < 10 is true, the program adds I to sum. Variable I is initially set to 1, then is incremented to 2, 3, and up to 10. When I is to, I < 10 is false, so the loop exists. Therefore, the sum is 1 + 2 + 3 + … + 9 = 45.

What happens if the loop is mistakenly written as follows?

int sum = sum, I = 1;

While (I < 10) {

Sum = sum + 1;

}

This loop is infinite, because I is always 1 and I < 10 will always be true.

Note: Make sure that the loop-continuation-condition eventually becomes false so that the loop will terminate. A common programming error involves infinite loops (i.e. the loops runs forever). If your program takes an unusually long time to run and does not stop, it may be an infinite loop. If you are running the program from the command window, press CTRL+C to stop it.

Caution: Programmers often make the mistake of executing a loop one more or less time This is commonly known as the off-by-one error.

5.4 Loop Design Strategies

The key to designing a loop is to identify the code that needs to be repeated and write a condition for terminating the loop.

Write a correct loop is not an easy task for novice programmers. Consider three steps when writing a loop.

Step 1: Identify the statements that need to be repeated.

Step 2: Wrap these statements in a loop as follows:

While (true) {

Statements;

}

Step 3: Code the loop-continuation-condition and add appropriate statements for controlling the loop.

While (loop-continuation-condition) {

Additional statements for controlling the loop;

}

5.5 Controlling a Loop with User Confirmation or a Sentinel Value

It is a common practice to use a sentinel value to terminate the input.

The preceding example executes the loop five times. If you want a user to decide whether to continue, you can offer a user confirmation. The template of the program can be coded as follows:

char continueLoop = ‘Y’;

while (continueLoop == ‘Y’) {

`//Execute the loop body once

…

//Prompt the use for confirmation

System.out.println(“Enter Y to continue and N to quit: “);

continueLoop = input.getLine().charAt();

}

Another common technique for controlling a lopp[ is to designate a special value when reading and processing a set of values. This special input value, known as a sentinel value, signifies the end of the input. A loop that uses a sentinel value to control its execution is called a sentinel-controlled loop.

Listing 5.5 gives a program that reads and calculates the sum of an unspecified number of integers.

Caution: Don’t use floating-point values for equality checking in a loop control. Because floating-point values are approximations for some values, using them could result in imprecise counter values and inaccurate results. Consider the following code for computing 1 + 0.9 + 0.8 + … + 0.1;

double item = 1; double sum = 0;

while (item !=0) {// No guarantee item will be 0

sum += item;

item -= 0.1;

}

System.out.println(sum);

Variable item starts with 1 and is reduced by 0.1 every time the loop body is executed. The loop should be terminate when item becomes 0. However, there is no guarantee that item will be exactly 0, because the floating-point arithmetic is approximated. This loop seems okay on the surface, but it is actually an infinite loop.

In the preceding example, if you have a large number of data to enter, it would be cumbersome to type from the keyboard. You can store the data separated by whitespaces in a text file, say input.txt, and run the program using the following command:

Java SentinelValue < input.txt

This command is called input redirection. The program takes the input from the file input.txt rather than having the user type the data from the keyboard at runtime.

5.6 The do-while Loop

A do-while loop is the same as a while loop except that it executes the loop body first then checks the loop continuation condition.

The do-while loop is a variation of the while loop. Its syntax is as follows:

Do {

// Loop body

Statement(s);

} while (loop-continuation-condition);

The loop body is executed first, then the loop-continuation-condition is evaluated. If the evaluation is true, the loop body is executed again; if it is false, the do-while loop terminates. For example, the following while loop statement

Int count = 0;

While (count < 100) {

System.out.println(“Welcome to Java!”);

Count++;

}

Can be written using a do-while loop as follows:

Int count = 0;

Do {

System.out.println(“Welcome to Java!”);

} while (count < 100);

The difference between a while loop and a do-while loop is the order in which the loop-continuation-condition is evaluated and the loop body is executed. In the case of a do-while loop, the loop body is executed at least once. You can write a loop using either the while loop or do-while loop. Sometimes is a more convenient choice than the other. For example, you can rewrite the while loop in Listing 5.5 using a do-while loop, as given in Listing 5.6.

Tip: Use a do-while loop if you have statements inside the loop that must be exectuted at least once.

5.7 The for Loop

A for loop has a concise syntax for writing loops.

Often you write a loop in the following common form:

i = initialValue; //Initialize loop control variable

while (I < endValue) {

//Loop body

…

i++; //Adjust loop control variable

}

This loop is intuitive and easy for beginners to grasp. However, programmers often forget to adjust the control variable, which leads to an infinite loop. A for loop can be used to simplify the preceding loop as shown in (a), which is equivalent to (b)

In general, the syntax of a for loop is as follows:

For (initial-action; loop-continuation-condition; action-after-each-iteration) {

// Loop body

Statement(s);

}

The for loop statement starts with the keyword for, followed by a pair of parentheses enclosing the control structure of the loop. This structure consists of initial-action, loop-continuation-condition, and action-after-each-iteration. The control structure is followed by the loop body enclosed inside braces. The initial-action, loop-continuation-condition, and action-after-each-iteration are separated by semicolons. A for loop generally uses a variable to control how many time the loop body is executed and when the loop terminates. This variable is referred to as a control variable. The initial-action often initializes a control variable, the action-after-each-iteration usually increments or decrements the control variable, and the loop-continuation-condition tests whether the control variable has reached a termination value. For example, the following for loop prints Welcome to Java! A hundred times:

int i;

for (i = 0; I < 100; i++) {

System.out.println(“Welcome to Java!”);

}

The initial-action, i = 0, initializes the control variable, i. The loop-continuation-condition, i < 100, is a Boolean expression. The expression is evaluated right after the initialization and at the beginning of each iteration. If this condition is true, the loop body is executed. If it is false, the loop terminates and the program control turns to the line following the loop. The action-after-each-iteration, i++, is a statement that adjusts the control variable. This statement is executed after each iteration and increments the control variable. Eventually, the value of the control variable should force the loop-continuation-condition to become false; otherwise, the loop is infinite.

The loop control variable can be declared and initialized in the for loop. Here is an example:

for (int i = 0; i < 100; i++) {

System.out.println(“Welcome to Java!”);

}

If there is only one statement in the loop body, as in this example, the braces can be omitted.

A for loop performs an initial action once, then repeatedly executes the statements in the loop body, and performs an action after an iteration when the loop-continuation-condition evaluates to true.

Tip: The control variable must be declared inside the control structure of the loop or before the loop. If the loop control variable is used only in the loop, and not elsewhere, it is a good programming practice to declare it in the initial-action of the for loop. If the variable is declared inside the loop control structure, it cannot be reference outside the loop.

Note: The initial-action in a for loop can be a list of zero or more comma-separated variable declaration statements or assignment expression. For example,

for (int i = 0, j = 0; i + j < 10; i++, j++) {

//Do something

}

The action-after-each-iteration in a for loop can be a list of zero or more comma-separated statements. For example:

for (int i = 1; i < 100; System.out.println(i), i++) ;

This example is correct, but it is a bad example, because it makes the code difficult to read. Normally, you declare and initialize a control variable as an initial action, and increment or decrement the control variable as an. Action after each iteration.

5.8 Which Loop to Use?

You can use a for loop, a while loop, or a do-while loop, whichever is convenient.

The while loop and do-while loop are easier to learn tan the for lop. However, you will learn the for loop quickly after some practice. A for loop places control variable initialization, loop continuation condition, and adjustment after each iteration all together. It is more concise and enables you to write the code with less errors than the other two loops.

The while loop and for loop are called pretest loops because the continuation condition is checked before the loop body is executed. The do-while loop is called a posttest loop because the condition is checked after the loop is executed. The three forms of loop statements--- while, do-while, and for---are expressly equivalent; that is, you can write a loop in any of these three forms.

Use the loop statement that is the most intuitive and comfortable for you. In general, a for loop may be used if the number of repetitions is known in advance, as, for example, when you need to display a message a hundred times. A while loop may be used if the number of repetitions is not fixed, as in the case of reading the numbers until the input is 0. A do-while loop can be used to replace a while loop is the loop body has to be executed before the continuation condition is tested.

Caution: Adding a semicolon at the end of the for clause before the loop body is a common mistake.

5.9 Nested Loops

A loop can be nested inside another loop.

Nested loops consist of an outer loop and one or more inner loops. Each time the outer loop is repeated, the inner loops are reentered, and started anew.

5.20 Minimizing Numeric Errors

Using floating-point numbers in the loop continuation condition may cause numeric errors.

Numeric errors involving floating-point numbers are inevitable, because floating-point numbers are represented in approximation in computers by nature. This section discusses how to minimize such errors through an example.

------Refer to Listing 5.8 TestSum.java

The for loop repeatedly adds the control variable I to sum. This variable, which begins with 0.01, is incremented by 0.01 after each iteration. The loop terminates when i exceeds 1.0. The for loop initial action can be any statement, but it is often used to initialize a control variable. From this example, you can see a control variable can be a float type. In fact, it can be any data type. The exact sum should be 50.50, but the answer is 50.499985. The result is imprecise because computers use a fixed number of bits to represent floating-point numbers, and thus they cannot represent some floating-point numbers exactly. If you change float in the program to double, as follows, you should see a slight improvement in precision, because a double variable holds 64 bits, whereas a float variable holds 32 bits.

// Initialize sum

Double sum = 0;

For (double i = 0.01; i <= 1.0; i = i + 0.01)

Sum+= i;

However, you will be stunned to see the result is actually 49.50000000003. What went rong? If you display i for each iteration in the loop, you will see that the last I is slightly larger than 1 (not exactly 1). This causes the last i not to be added into sum. The fundamental problem is the floating-point are represented by approximation. To fix the problem, use an integer count to ensure all the numbers are added to sum. Here is the new loop:

Double currentValue = 0.01;

For (int count = 0; count < 100; count++) {

Sum += currentValue;

currentValue += 0.01;

}

After this loop, sum is 50.5000000003. This loop adds the numbers from smallest to biggest. What happens if you add from biggest to smallest:

Double currentValue = 1.0;

For (int count = 0; count < 100; count++) {

Sum += currentValue;

Currentvalue -= 0.01;

}

After this loop, sum is 50.49999999995. Adding from biggest to smallest is less accurate than adding from smallest to biggest. This phenomenon is an artifact of the finite-precision arithmetic. Adding a very small number to a very big number can have no effect if the result requires more precision than the variable can store. For example, the inaccurate result of 10000000.0 + 0.000000001 is 10000000.0. To obtain more accurate results, carefully select the order of computation. Adding smaller numbers before bigger numbers to sum is one way to minimize errors.

5.11 Case Studies

Loops are fundamental in programming. The ability to write loops is essential in learning Java programming.

\*\*\*\*\*\*\*\*\*Refer to Case Study 5.11.1 Find the Greatest Common Divisor

Translating a logical solution to Java code is not unique. For example, you could use a for loop to rewrite the code as follows:

for int k = 2; k<= n1 && k <= n2; k++) {

if (n1 % k == 0 && n2 % k == 0) {

gcd = k;

}

}

A problem often has multiple solution, and the gcd problem can be solved in many ways. A more efficient solution is to use the classic Euclidean algorithm.

\*\*\*\*\*\*\*\*\*Refer to Case Study 5.11.2 Predicting the Future Tuition

5.12 Keywords break and continue

The break and continue keywords provide additional controls in a loop.

Pedagogical Note: Two keywords, break and continue, can be used in loop statements to provide additional controls. Using break and continue can simplify programming in some cases. Overusing or improperly using them, however, can make programs difficult to read and debug.

You have used the keyword break in a switch statement. You can also use break in a loop to immediately terminate the loop.

\*\*\*\*\*\*\*\*\*Refer to Listing 5.12 TesBreak.java

Note: The continue statement is always inside a loop. In the while and do-while loops, the loop-continuation-condition is evaluated immediately after the continue statement. In the for loop, the action-after-each-iteration is performed, then the loop-continuation-condition is evaluated immediately after the continue statement.

Note: Some programming languages have a goto statement. The goto statement indiscriminately transfers control to any statement in the program and executes it. This makes your program vulnerable to errors. The break and continue statements in Java are different from goto statements. They operate only in a loop or a switch statement. The break statement breaks out of the loop, and the continue statement breaks out of the current iteration in the loop.

You can always write a program without using break or continue in a loop. In general, though, using break and continue is appropriate if it simplifies coding and makes programs easier to read.

Chapter 6 – Methods

Methods can be used to define reusable code and organize and simplify coding.

Suppose you need to find the sum of integers from 1 to 10, 20 to 37, and 35 to 49, respectively. You may write the code as follows:

int sum = 0;

For (int i = 1; I <= 10; i++) {

Sum += 1;

}

System.out.println(“Sum from 1 to 10 is “ + sum);

int sum = 0;

For (int i = 20; I <= 37; i++) {

Sum += 1;

}

System.out.println(“Sum from 20 to 37 is “ + sum);

int sum = 0;

For (int i = 35; I <= 49; i++) {

Sum += 1;

}

System.out.println(“Sum from 35 to 49 is “ + sum);

You may have observed that computing these sums from 1 to 10, 20 to 37, and 35 to 49 are very similar, except that the starting and ending integers are different. Wouldn’t it be nice if we could write the common code once and reuse it? We can do so by defining a method and invoking it. The preceding code can be simplified as follows:

public static int sum( int i1, int i2) {

int result = 0;

for (int i = i1; I <= i2; i++) {

result +=i;

}

return result;

}

Public static void main(String[] args) {

System.out.println(“Sum from 1 to 10 is “ + sum(1,10));

System.out.println(“Sum from 20 to 37 is “ + sum(20,27));

System.out.println(“Sum from 35 to 49 is “ + sum(35,49));

}

Lines 1-7 define the method named sum with two parameters i1 and i2. The statements in the main method invoke sum(1,10) to compute the sum from 1 to 10, sum(20,37) to compute the sum from 20 to 37, and sum(35, 49) to compute the sum from 35 to 49.

A method is a collection of statements grouped together to perform an operation. In earlier chapters you have used predefined methods such as System.out.println. These methods are defined in the Java library. In this chapter, you will learn how to define your own methods and apply method abstraction to solve complex problems.

6.2 Defining a Method

A method definition consists of method name, parameters, return value type, and body.

The syntax for defining a method is as follows:

Modifier returnValueType methodName(list of parameters) {

//Method body;

}

The method header specifies the modifiers, return value type, method name, and parameters of the method. The static modifier is used for all the methods in this chapter. The reason for using it will be discussed in Chapter 9, Object and Classes.

A method may return a value. The returnValueType is the data type of the value the method returns. Some methods perform desired operations without returning a value. In this case, the returnValueType is the keyword void. If a method returns a value, it is called a value-returning method; otherwise, it is called a void method.

The variables defined in the method header are known as formal parameters or simply parameters. A parameter is like a placeholder: when a method is invoked, you pass a value to the parameter. This value is referred to as an actual parameter or argument. The parameter list refers to the method’s type, order, and number of parameters. The method name and the parameter list together constitute the method signature. Parameters are optional; that is, a method may contain no parameters.

The method body contains a collection of statements that implement the method. In order for a value-returning method to return a result, a return statement using the keyword return is required. The method terminates when a return statement is executed.

Note: Some programming languages refer to methods as procedures and functions. In those languages, a value-returning method is called a function and a void method is called a procedure.

Caution: In the method header, you need to declare each parameter separately. For instance, max(int num1, int num2) is correct, but max(int num1, num2) is wrong.

Note: We say “define a method” and “declare a variable.” We are making a subtle distinction here. A definition defines what the defined item is, but a declaration usually involves allocating memory to store data for the declared item.

6.3 Calling a Method

Calling a method executes the code in the method.

In a method definition, you define what to method is to do. To execute the method, you have to call or invoke it. The program that calls the function is called a caller. There are two ways to call a method, depending on whether the method returns a value or not. If a method returns a value, a call to the method is usually treated as a value. For example,

int larger = max(3,4);

calls max(3,4) and assigns the result of the method to the variable larger. Another example of a call that is treated as a value is

System.out.println(max(3,4));

Which prints the return value of the method call max(3,4). If a method returns void, a call to the mthod must be a statement. For example, the method println returns void. The following call is a statement:

System.out.println(“Welcome to Java!”);

When a program calls a method, program control is transferred to the called method. A called method returns control to the called when its return statement is executed or when its method-ending closing brace is reached.

\*\*\*\*\*\*Refer to Listing6\_1\*\*\*\*\*\*\*

Listing 6.1 contains the main method and the max method. The main method is just like any other method, except that it is invoked by the JVM to start the program.

The main method’s header is always the same. Like the one in this example, it includes the modifiers public and static, return value type void, method name main, and a parameter of the String[] type. String[] indicates the parameter is an array of String, a subject discussed in Chapter 7.

The statements in main may invoke other methods that are defined in the class that contains the main method or in other classes. In this example, the main method invokes max(1, j), which is defined in the same class with the main method. When the max method is invoked, variable I’s value 5 is passed to num1 and variable j’s value is passed to num2 in the max method. The flow control transfers to the max method and the max method is executed. When the return statement in the max method is executed, the max method returns the control to its caller (. In this case, the caller is the main method).

Caution: A return statement is required for a value-returning method.

Each time a method is invoked, the system creates an activation record (also called an activation frame) that store parameters and variables for the method and places the activation record in an area of memory known as a call stack. A call stack is also known as execution stack, runtime stack, or machine stack and it is often shortened to just “the stack.” When a method calls another method, the caller’s activation record is kept intact and a new activation record is created for the new method called. When a method finishes its work and returns to its caller, its activation record is removed from the call stack.

A call stack stores the activation records in a last-in, first-out fashion: The activation record for the for the method that is invoked last is removed first from the stack. Understanding call stacks helps you to comprehend how methods are invoked.

6.4 void vs. Value-Returning Methods

A void method does not return a value.

\*\*\*Please Refer to Listing 6.2\*\*\*\*\*\*

The printGrade method is a void method because it does not return any value. A call to a void method must be a statement. Therefore, it is invoked as a statement in the main method. Like any Java statement, it is terminated with a semicolon. To see the difference between a void and value-retuning method, let’s redesign the printGrade method to return a value. The new method, which we call getGrade, returns the grade as given in Lisiting 6.3.

\*\*\*Please refer to Listing 6.3\*\*\*

The get grade method returns a character grade based on the numeric score value. The caller invokes this method in lines 3 and 4.

The getGrade method can be invoked by a caller whenever a character may appear. The printGrade method does not return any value, so it must be invoked as a statement.

**6.5 Passing Parameters by Values**

The arguments are passed by value to parameters when invoking a method.

The power of a method is its ability to work with parameters. You can use println to print any string, and max to find the maximum of any two int values. When calling a method, you need to provide arguments, which must be given in the same order as their respective parameters in the method signature. This is know as parameter order association. For example, the following method prints a message n times:

Public static void nPrintln(String message, int n) {

For (int I = 0; I < n; i++) {

System.out.println(message);

}

}

You can use nPrintln(“Hello”, 3) to print Hello three times. The nPrinln(“Hello”, 3) statement passes the actual string parameter Hello to the parameter message, passes 3 to n, and prints Hello three times. However, the statement nPrintln(3, “Hello”) would be wrong. The data type of 3 does not match the data type for the first parameter, message, nor the second argument, Hello, match the second parameter, n.

Caution: The arguments must match the parameters in order, number, and compatible type, as defined in the method signature. Compatible type means you can pass an argument to a parameter without explicit casting, such as passing an int value argument to a double value parameter.

When you invoke a method with an argument, the value of the argument is passed to the parameter. This is referred to a pass-by-value. If the argument is a variable rather than a literal value, the value of the variable is passed to the parameter. The variable is not affected, regardless of the changes made to the parameter inside the method.

\*\*\*Please Refer to Listing 6.4\*\*\*

\*\*\*Please refer to Listing 6.5\*\*\*

Before the swap method is invoked, num1 is 1 and num2 is 2. After the swap method is invoked, num1 is still 1 and num2 is still 2. Their values have not been swapped. Another twist is to change the parameter name n1 in swap to num1. What effect does this have? No change occurs, because it makes no difference whether the parameter and the argument have the same name. The parameter is a variable in the method with its own memory space. The variable is allocated when the methods is invoked, and it disappears when the method is returned to its caller.

6.6 Modularizing Code

Modularizing makes the code easy to maintain and debug and enables the code to be reused.

Methods can be used to reduce redundant code and enable code reuse. Methods can also be used to modularize code and improve the quality of the program.

\*\*\*Refer to Listing 6.6 \*\*\*

By encapsulating the code for obtaining the gcd in a method, this program has several advantages:

1. It isolates the problem for computing the gcd from the rest of the code in the main method. Thus, the logic becomes clear, and the program is easier to read.
2. The erros on computing the gcd are confined in the gcd method, which narrows the scope of debugging.
3. The gcd method now can be reused by other programs.

\*\*\*Please refer to Listing 6.7 \*\*\*

\*\*\*Please Refer to Case Study 6.7\*\*\*

**6.8 Overloading Methods**

Overloading methods enable you to define the methods with the same name as long as their parameter lists are different.

The max method used earlier works only with the int data type. But what is you need to determine which of the two floating-point numbers has the maximum value? The solution is to create another method with the same name but different parameters, as shown in the following code:

public static double max(double num1, double num2) {

if (num1 > num2) {

return num1;

} else {

return num2;

}

}

If you call max with int parameters, the max method that expects int parameters will be invoked; and if you call max with double parameters, the max method that expects double parameters will be invoked. This is referred to as method overloading; that is, two methods have the same name but different parameter lists within one class. The java compiler determines which method to use based on the method signature.

\*\*\*Please refer to Listing 6.9\*\*\*

Tip: Overloading methods can make programs clearer and more readable. Methods that perform the same function with different types of parameters should be given the same name.

Note: Overloading methods must have different parameter lists. You cannot overload methods based on different modifiers or return types.

Note: Sometimes there are two or more possible matches for the invocation of a method, but the compiler cannot determine the most specific match. This is referred to as ambiguous invocation. Ambiguous invocation causes a compile error.

The following code:

Public static double max(int num1, double num2) {

}

Public static double max(double num1, int num2) {

}

**6.9 The Scope of Variables**

The scope of a variable is the part of the program where the variable can be reference.

A variable defined inside a method is referred to as a local variable. The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable. A local variable must be declared and assigned a value before it can be used.

A parameter is actually a local variable. The scope of a method parameter covers the entire method. A variable declared in the initial-action part of a for-loop header has its scope in the entire loop. However, a variable declared inside a for-loop body has its scope limited in the loop body from its declaration to the end of the block that contains the variable. A variable declared in the initial-action part of a for-loop header has its scope in the entire loop. You can declare a local variable with the same name in different blocks in a method, but you cannot declare a local variable twice in the same block or in nested blocks.

\*\*\*Please Refer to Case Study 6.10\*\*\*

\*\*\*Please Refer to Listing 6.11 after Case Study 6.10\*\*\*

6.11 Method Abstraction and Stepwise Refinement

The key to developing software is to apply the concept of abstraction.

Method abstraction is achieved by separating the use of a method from its implementation. The client can use a method without knowing how it is implemented. The details of the implementation are encapsulated in the method and hidden from the client who invokes the method. This is also known as information hiding or encapsulation. If you decide to change the implementation, the client program will not be affected, provided that you do not change the method signature. The implementation of the method is hidden from the client in a “Black box.”’

The method body can be thought of as a black box that contains the detailed implementation for the method. You have already used the System.out.print method to display a string and the mx method to find the maximum number. You know how to write code to invoke these methods in your program, but as a user of these methods, you are not required to know how they are implemented. The concept of method abstraction can be applied to the process of developing programs. When writing a large program, you can use the divide-and-conquer strategy, also known as stepwise refinement, to decompose it into subproblems. The subproblems can be further decomposed into smaller, more manageable problems.

Suppose that you write a program that displays the calendar for a given month of the year. The program prompts the user to enter the year and the month, and then display the entire calendar for the month. Let us use this example to demonstrate the divide-and-conquer approach.

6.11.1 Top-Down Design

How would you get started on such a program? Would you immediately start coding? Beginning programmers often start by trying to work out the solution to every detail. Although details are important in the final program, concern for detail in the early stages may block the problem-solving process. To make problem solving flow as smoothly as possible, this example begins by using method abstraction to isolate details from design and only later implements the details.

For example, the problem is first broken into two subproblems: get input from the user, and print the calendar for the month. At this stage, you should be concerned with what the subproblems will achieve, not with how to get input and print the calendar for the month.

You can use Scanner to read input for the year and the month. The problem of printing the calendar for a given month can be broken into two subproblems: print the month title, and print the month body. The month title, consists of three lines: month and year, a dashed line, and the names of the seven days of the week. You need to get the month name from the numeric month. This is accomplished in getMonthName.

In order to print the month body, you need to know which day of the week is the first day of the month (getStartDay) and how many days the month has (getNumberOfDaysInMonth). For example, December 2013 has 31 days, and December 1, 2013 is a Sunday. How would you get the start day for the first date in a month? There are several ways to do so. For now, we’ll use an alternative approach. Assume you know that the start day for January 1, 1800 was a Wednesday (START\_DAY\_FOR\_JAN\_1\_1800 = 3). You could compute the total number of days (totalNumberOFDays) between January 1, 1800 and the first date of the calendar month. The start day for the calendar month is (totalNumberOfDays + START\_DAY\_FOR\_JAN\_1\_1800) % 7, since every week has seven days. Thus, the getStartDay problem can be further refined as getTotalNumberOfDays. To get the total number of days, you need to know whether the year is a leap year and the number of days in each month. Thus, getTotalNumberOfDays can be futher refined into two subproblems: isLeapYear and getNumberOfDaysInMonth.

6.11.2 Top-Down and/or Bottom-Up Implementation

Now we turn our attention to implantation. In general, a subproblem corresponds to a method in the implementation, although some are so simple that this is unnecessary. You would need to decide which modules to implement as methods and which to combine with other methods. Decisions of this kind should be based on whether the overall program will be easier to read as a result of your choice. In this example, the subproblem readInput can be simply implemented in the main method. You can use either a “top-down” or a “bottom-up” approach. The top-down approach implements one method in the structure chart at a time from the top to the bottom. Stubs –a simple but incomplete version of a method –can be used for the mthods waiting to be implemented. The use of stubs enables you to quickly build the framework of the program. Implement the main method first then use a stud for the printMonth method. For example, let printMonth display the year and the month in the stub. Thus, your program may begin as follows:

Public class PrintCalendar {

/\*\* Main Method \*/

Public static void main(String[] args) {

Scanner input = new Scanner(System.in);

//Prompt the user to enter year

System.out.print(“Enter full year (e.g., 2012): “);

int year = input.nextInt();

//Prompt the use to enter month

Int month = input.nextInt();

//Print Calendar for the month of the year

printMonth(year, month);

}

/\*\* A Stub for printMonth may look like this \*/

Public static void printMonth (int year, int month) {

System.out.print(month + “ “ + year);

}

Compile and test the program, and fix any errors. You can now implement the printMonth method. For methods invoked from the printMonth methods, you can again use stubs. The bottom-up approach implements one method in the structure chart at a time from the bottom to the top. For each method implemented, write a test program, known as the driver to test it. The top-down and bottom-up approaches are equally good: Both approaches implement methods incrementally, help to isolate programing errors, and make debugging easy. They can be used together.

6.11.3 Implementation Details

The isLeapYear(int year) method can be implemented using the following code from Section 3.11:

Return year % 00 == 0 || (year % 4 ==0 && year % 100 !=0);

Use the following facts to implement getTotalNumberOFDaysInMonth(int year, int month):

* January, March, May, July, August, October, and December have 31 days.
* April, June, September, and November have 30 days.
* February has 28 days during a regular year, and 29 days during a leap year. A regular year, therefore has 365 days, and a leap year has 366 days.

To implement getTotalNumberOfDays(int year, int month), you need to compute the total number of days (totalNumberOfDays) between January 1, 1800 and the first day of the calendar month. You could find the total number of days between the year 1800 and the calendar year then figure out the total number of days prior to the calendar month in the calendar year. The sum of these two total is totalNumberOfdays. To print a body, first pad some space before the start say then print the lines for every week. The complete program is given in Listing 6.12

\*\*\*Please Refer to Listing 6.12\*\*\*

The program does not validate user input. For instance, if the user enters either a month not in the range between 1 and 12 or a year before 1800, the program displays an erroneous calendar. To avoid this error, add an if statement to check the input before printing the calendar. This program prints calendars for a month, but could easily be modified to print calendars for a whole year. Although it can print month only after January 1800, it could be modified to print months before 1800.

6.11.4 Benefits of Stepwise Refinement

Stepwise refinement breaks a large problem into smaller manageable subproblems. Each sub-problem can be implemented using a method. This approach males the program easier to write, reuse, debug, test, modify, and maintain.

Simpler program

The print calendar program is long. Rather than writing a long sequence of statements in one method, stepwise refinement breaks it into smaller methods. This simplifies the program and makes the while program easier to read and understand.

Reusing Methods

Stepwise refinement promotes code reusue within a program. The isLeapYear method is defined once and invoked from the getTotalNumberOfDays and getNumberOfDaysInMonth methods. This reduces redundant code.

Easier Developing, Debugging, and testing

Since each subproblem is solved in a method, a method can be developed, debugged, and tested individually. This isolates the errors and makes developing, debugging, and testing easier. When implementing. A large program, use the top-down and/or bottom-up approach. Do not write the entire program at once. Using these approaches seems to take more development time (because you repeatedly compile and run the program), but it actually saves time and makes debugging easier.

Better Facilitating Teamwork

When a large problem is divided into subproblems, subproblems can be assigned to different programmers. This makes it easier for programmers to work in teams.

**Chapter 7: Single-Dimensional Arrays**

**7.1 Introduction**

A single array variable can reference a large collection of data.

Often you will have to store a large number of values during the execution of a program. An efficient, organized approach is needed. Java and most other high-level languages provide a data structure, the array, which stores a fixed-size sequential collection of elements of the same type.

**7.2 Array Basics**

Once an array is created, its size is fixed. AN array reference variable is used to access the elements in an array using an index.

An array is used to store a collection of data, but often we find it more useful to think of an array as a collection of variables of the same type. Instead of declaring individual variables such as number 0, number1, …, number99, you can declare one array variable such as numbers and use numbers[0], numbers[1], …, and numbers [99] to represent individual variables. This section introduces how to declare array variables, create arrays, and process arrays using indexes.

**7.2.1 Declaring Array Variables**

To use an array in a program, you must first declare a variable to reference the array and specify the array’s element type. Here is the syntax for declaring an array variable:

elementType[] arrayRefVar;

or

elementType arrayRefVar[]; //Allowed, but not preferred

The elementType can be any data type, and all elements in the array will have the same data type. For example, the following code declares a variable myList that reference an array of double elements.

double[] myList;

Or

double myList[]; //Allowed, but not preferred

Note: You can use elementType arrayRefVar[] to declare an array variable. This style comes from the C/C++ language and was adopted in Java to accommodate C/C++ programmers.

**7.2.2 Creating Arrays**

Unlike declarations for primitive data type variables, the declaration of an array variable does not allocate any space in memory for the array. It creates only a storage location for the reference to an array. If a variable does not contain a reference to an array, the value of the variable is null. You cannot assign elements to an array unless it has already been created. After an array variable is declared, you can create an array by using the new operator an assign its reference to the variable with the following syntax:

arrayRefVar = new element type[arraySize];

This statement does two things: (1) it creates an array using new elementType[arraySize] and (2) it assigns the reference of the newly created array to the variable arrayRefVar. Declaring an array variable, creating an array, and assigningnthe reference of the array to the variable can be combined in one statement as

elementType[] arrayRefVar = new elementType[arraySiz];

or

elementType arrayRefVar[] = new elementType[arraySize];

Here is an example of such a statement:

double[] myList = newdouble[10];

This statement declares an array variable, myList, creates an array of 10 elements of double type, and assigns its reference to myList. To assign values to the elements, use the syntax

arrayRefVar[index] = value;

For example, the following code initializes the array:

myList[0] = 5.6;

myList[1] = 4.5;

…  
 myList[8] = 99.993;

myList[9] = 11123;

**7.2.3 Array Size and Default Values**

When space for an array is allocated, the array size must be given, specifying the number of elements that can be stoe4d in it. The size of an array cannot be changed after the array is created. Size can be obtained using arrayRefVar.length. For example, myList.length is 10. Wen an array is created, its elements are assigned the default value of 0 for the numeric primitive data types, \u0000 for char types, and false for Boolean types.

**7.2.4 Accessing Array Elements**

The array elements are accessed through the index. Array indices are 0 based; that is, they range from 0 to arrayRefVar.length -1. In the example myList holds 10 double valuesm and the indices are from 0 to 9. Each element in the array is represented using the following syntax, known as an indexed variable:

arrayRefVar[index];

For example, myList[9] represents the last element in the array myList.

An indexed variable can be used in the same way as a regular variable. For example, the following code adds thevalues in myList[0] and myList[1] to myList[2]:

myList[2] = myList[0] + myList[1];

The following loop assigns 0 to myList[0], 1 to myList[1], …, and 9 to myList[9]:

For (int I = 0; I < myList.length; i++) {

myList[i] = I;

}

**7.2.5 Array Initializers**

Java has a shorthand notation, known as the array initializer, which combines the declaration, creation, and initialization of an array in one statement using the following syntax:

elementType[] arrayRefVar = {value0, value1, …, valueK};

For example, the statement

double[] myList = [1.9,2.9,3.4,3.5};

declares, creates, and initializes the array myList with four elements, which is equivalent to the following statements:

double[] myList = newDouble[4];

myList[0] = 1.9;

myList[1] = 2.9;

myList[2] = 3.4;

myList[3] = 3.5;

7.2.6 Processing Arrays

When processing array elements, you will often use a for loop for one of two reasons:

1. All of the elements in an array are of the same type. They are evenly processed in the same fashion repeatedly using a loop.
2. Since the size of the array is known, it is natural to use a for loop.

Assume that the array is created as follows:

double[] myList = new double[10];

The following are some examples of processing arrays:

1. Initializing arrays with input values: The following loop initializes the array myList with user input values:

Java.util.Scanner input = new java.util.Scanner(System.in);

System.out.print(“Enter” + myList.length + “ values: “);

For (int I = 0; I < myList.length; i++)

myList[i] = input.nextDouble();

1. Initializing arrays with random values: The following loop initializes the array myList with random values between 0.0 and 100.0, but less than 100.0;

For (int I = 0; I < myList.length; i++) {

myList[i] = Math.random() \*100;

}

1. Displaying arrays: To print an array. You have to print each element in the array using a loop such as the following:

For (int I = 0; I < myList.length; i++) {

System.out.print(myList[i] + “ “);

}

1. Summing all elements: Use a variable named total to store the sum. Initially total is 0. Add each element in the array to total using a loop such as the following:

Double total = 0;

For (int I = 0; I < myList.length; i++) {

Total += myList[i];

}

1. Finding the largest element: Use a variable named max to store the largest element. Initially max is myList[0]. To find the largest element in the array myList, compare each element with max, and update max if the element is greater than max.

Double max = myList[0];

For (int I = 1; I < myList.length; i++) {

If (myList[i] > max) {

Max = myList[i];

}

}

1. Finding the smallest index of the largest element: Often you need to locate the largest element in an array. If an array has multiple elements with the same largest value, find the smallest index of such an element. Suppose that the array myList is {1,5,3,4,5,5}. The largest element is 5, and the smallest index for 5 is 1. Use a variable named max to store the largest element, and a variable named indexOfMax to denote the index of the largest element. Initialy max is myList[0] and indexOfMax is 0. Compare each element in myList with max and update max and indexOfMax if the element is greater than max.

Double max = myList[0];

Int indexOfMax = 0;

For (int I = 1; I < myList.length; i++) {

If (myList[i] > max) {

Max = myList{i};

indexOfMax = I;

}

}

1. Random shuffling: In many application, you need to randomly reorder the elements in an array. This is called shuffling. To accomplish this, for each element myList[i], randomly generate an index j and swap myList[i] with myList[j], as follows:

For (int I = 0; I < myList.length – 1; I ++) {

// Generate an index j randomly

Int j = (int)(Math.random() \* myList.length);

Double temp = myList[i];

myList[i] = myList[j];

myList[j] = temp:

}

1. Shifting elements: Sometimes you need to shift the elements left or right. Here is an example of shifting the elements one position to the left and filling the last element with the first element:

Double temp = myList[0]; //Retain the first element

// Shift elements left

For (int I = 1; I < myList.length; i++) {

myList[I – 1] = myList[i];

}

// Move the first element to fill in the last position

myList[mylist.length -1] = temp;

1. Simplifying coding: Arrays can be used to greatly simplify coding for certain tasks. For example, suppose you wish to obtain the English name of a given month by its number. If the month names are stored in an array, the month name for a given month can be accessed via the index. The following code prompts the user to enter a month number and displays its month name:

String[] months = {“January”, “February”, …, “December”};

System.out.print(“enter a month number (1 to 12): “);

Int monthNumber = input.nextInt();

System.out.println(“The month is “ + months[monthNumber – 1];

If you didn’t use the months array, you would have to determine the onth name using a lengthy multieay if-else statement as follows:

If (monthNumber == 1)

System.out.println(“The month is January”);

Else if (monthNumber == 2)

System.out.println(“The month is February’);

…

Else

System.out.println(“The month is December”);

7.2.7 Foreach Loops

Java supports a convenient for loop, known as a foreach loop, which enables you to traverse the array sequentially without using an index variable. For example, the following code displays all the elements in the array myList:

For (double e: myList) {

System.out.println(e);

}

You can read the code as “for each element e in myList, do the following.” Note that the variable, e, must be ddeclared as the same type as the elemetns in myList.

In general, the syntax for a foreach loop is

For (elementType element: arrayRefVar) {

//Process the element

}

You still have to use an index variable if you wish to traverse the array in a different order or change the elements in the array.

7.5 Copying Arrays

To copy the contents of one array into another, you have to copy the array’s individual elements into the other array.

Often, in a program, you need to duplicate an array or a part of an array. In such cases you could attempt to use the assignment statement (=), as follows:

List2 = list1;

However, this statement does not copy the contents of the array referenced by list 1 to list 2, but instead merely copies the reference value from list 1 to list 2. After this statement, list 1 and list2 reference the same array. The array previously referenced by list 2 is referenced; it becomes garbage, which will be automatically collected by the Java Virtual Machine. This process is called garbage collection.

In java, you can use assignment statement to copy primitive data type variables, but not arrays. Assigning one array variable to another array variable actually copies one reference to another and makes both variables point to the same memory location. There are three wasy to copy arrays:

1. Use a loop to copy individual elements one by one.
2. Use the static arraycopy method in the System class.
3. Use the clone method to copy arrays; this will be introduced in Chapter 13.

You can write a loop to copy every element from the source array to the corresponding element in the target array. The following code, for instance, copies sourceArray to targetArray using a for loop:

Int[] sourceArray = {2,3,1,5,10};

Int[] targetArray = new int[sourceArray.length];

For (int I = 0; I < sourceArray.length; i++) {

targetArray[i] = sourceArray[i];

}

Another approach is to use the arraycopy method in the java.lang.System class to copy arrays instead of using a loop.

Arraycopy(sourceArray, srcPos, targetArray, tarPos, sourceArray.length);

The parameters srcPos and tarPos indicate the starting positions in sourceArray and targetArray, respectively. The number of elements copied from sourceArray to targetArray is indicated by Length. For example, you can rewrite the loop using the following statement:

System.arraycopy(sourceArray, 0, targetArray, 0, sourceArray.length);

The arraycopy method does not allocate memory space for the target array. The target array must already been created with its memory space allocated. After the copying takes place, targetArray and sourceArray have the same content but independent memory locations.

7.6 Passing Arrays to Methods

When passing an array to a method, the reference of the array is passed to the method.

Just as you can pass primitive type values to methods you can also pass arrays to methods. For example, the following method displays the elements in an int array:

Public static void printArray(int[] array) {

For (int I = 0; I < array.length; i++) {

}

}

You can invoke it by passing an array. For example, the following statement invokes the printArray method to display 3,1,2,6,4, and 2.

printArray(new int[]{3,1,2,,6,4,2});

Note: The preceding statement creates an array using the following syntax:

New elementType[]{value0, value1, … , valuek};

There is not explicit reference variable for the array. Such array is called an anonymous array.

Java uses pass-by-value to pass arguments to a method. There are important differences between passing the values of variables of primitive data types and passing arrays.

* For an argument of a primitive type, the argument’s value is passed.
* For an argument of an array type, the value of the argument is a reference to an array; this reference value is passed to the method. Semantically, it can be best described as pass-by-sharing, that is, the array in the method is the same as the array being passed. Thus, if you change the array in the method, you will see the change outside the method.

Take the following code, for example:

Public class TestArrayArguments {

Int x = 1; // x represents an int value

Int[] y = new int[10]; // y represents an array of int values

M(x,y); //Invoke m with arguments x and y

System.out.println(“x is “ + x);

System.out.println(“y[0] is “ + y[0]);

}

Public static void m(int number, int[] numbers) {

Number = 1001; // Assign a new value to number

Numbers[0] = 555; //Assign a new value to numbers[0]

}

}

You may wonder why after m is invoked, x remains 1, but y[0] becomes 555. This is because y and numbers, although they are independent variables, reference the same array. When m(x,y) is invoked, the values of x and y are passed to number and numbers. Since y contains the reference value to the array, numbers now contains the same reference value to the same array.

Note: Arrays are objects in Java (objects are introduced in Chapter 9). The JVM stores the objects in an area of memory called the heap, which is used for dynamic memory allocation.

\*\*\*Refer to Listing 7.3\*\*\*

7.7 Returning an Array from a method

When a method returns an array, the reference of the array is returned.

You can pass arrays when invoking a method. A method may also return an array. For example, the following method returns an array that is the reversal of another array.

Public static int[] reverse(int[] list) {

Int[] result = new int[list.length];

For (int I = 0; j = result.length -1; I < list.length; i++, j--) {

Result[j] = list[i];

}

Return result;

}

\*\*\* Refer to Case Study 7.8 \*\*\*

**7.9 Variable-Length Argument Lists**

A variable number of arguments of the same type can be passed to a method and treated as an array.

You can pass a variable number of arguments of the same type to a method. The parameter in the method is declared as follows:

typeName… parameterName

In the method declaration, you specify the type by an ellipsis (…). Only one variable-length parameter may be specified in a method, and this parameter must be the last parameter. Any regular parameters must precede it. Java treats a variable-length parameter as an array. You can pass an array or a variable number of arguments to a variable-length parameter. When invoking a method with a variable number of arguments, Java creates an array and passes the arguments to it.

\*\*\*Refer to Listing 7.5 \*\*\*

7.10 Searching Arrays

If an array is sorted, binary search is more efficient than linear search for finding an element in the array.

Searching is the process of looking for specific element in an array. Searching is a common task in computer programming. Many algorithms and data structures are devoted to searching. This section discusses two commonly used approaches, linear search and binary search.

7.10.1 The Linear Search Approach

The linear search approach compares the key element key sequentially with each element in the array. It continues to do so until the key matches an element in the array, or the array is exhausted without a match being found. If a match is made, the linear search returns the index in the array that matches the key. If no match is found, the search returns -1.

\*\*\*Refer to Listing 7.6 \*\*\*

The linear search method compares the key with each element in the array. The elements can n any order. On average, the algorithm will have to examine half of the elements in an array before finding the key, if it exists. Since the execution time of a linear search increases linearly as the number of array elements increases, linear search is inefficient for a large array.

7.10.2 The Binary Search Approach

Binary search is the other common search approach for a list of values. For binary search to work, the elements in the array must already be order. Assume that the array is in ascending order. The binary search first compares the key element with the element in the middle of the array. Consider the following three cases”

1. If the key is less than the middle elements, you need to continue to search for the key in the first half of the array.
2. If the key is equal to the middle element, the search end with a match.
3. If the key is greater than the middle element, you need to continue to search for the key only in the second half of the array.

Clearly, the binary search method eliminates at least half of the array after each comparison. Sometimes you eliminate half of the elements, and sometimes you eliminate half plus one. You know how the binary search works. The next task is to implement it incrementally, one step at a time. You may start with the first iteration of the search. It compares the key with the middle element in the list whose low index is 0 and high index list.length – 1. If key < list[mid], set the high index to mid -1; if key == list[mid], a match is found and return mid; if key > list[mid], set the low index to mid + 1. Next, consider implementing the method to perform the search repeatedly by adding a loop. The search ends if the key is found, or if the key is not found when low > high.

When the key is not found, low is the insertion point where a key would be inserted to maintain the order of the list. It is more useful to return the insertion point than -1. The method must return a negative value to indicate that the key is not in the list. Can it simply return -low? No. If the key is less than list[0], low would be 0. -0 is 0. This would indicate the key matches list[0]. A good choice is to let the method return -low-1 if the key is not in the list. Returning -low-1 indicates not only that the key is not in the list, but also where the key would be inserted.

\*\*\* Refer to Listing 7.7 \*\*\*

The precondition for the binary search method is that the list must be sorted in increasing order. The post condition is that the method returns the index of the element that matches the key if the key is in the list or a negative integer k such that -k – 1 is the position for inserting the key. Precondition and postcondition are the terms often used to describe the properties of a method. Preconditions are the things that are true before the method is invoked, and postconditions are the things that are true after the method is returned.

7.11 Sorting Arrays

Sorting, like searching, is a common task in computer programming. Many different algorithms have been developed for sorting. This section introduces an intuitive sorting algorithm: selection sort.

Suppose you want to sort a list in ascending order. Selection sort finds the smallest number in the list and swaps it with the first element. It then finds the smallest number remaining and swaps it with the first element. It then finds the smallest number remaining and swaps it with the second element, and so on, until only a single number remains.

\*\*\*Refer to Listing 7.8\*\*\*

7.12 The Arrays Class

The java.util.Arrays class contains useful methods for common array operations such as sorting and searching.

The java.util.Arrays class contains various static methods for sorting and searching arrays, comparing arrays, filling array elements, and returning a string representation of the array. These methods are overloaded for all primitive types. You can use the sort or parallelSort method to sort a whole array or a partial array. For example, the following code sorts an array of numbers and an array of characters:

double[] numbers = {6.0,4.4,1.9,2.9,3.4,3.5};

java.util.Arrays.sprt(numbers); //Sort the whole array

java.util.Arrays.parallelSort(numbers); // Sort the whole array

char[] chars = {‘a’, ‘A’, ‘4’, ‘F’, ‘D’, ‘P’};

java.util.Arrays.sort(chars, 1, 3) //Sort part of the array

java.util.Arrays.parallelSort(chars, 1, 3) //Sort part of the array

Invoking sort(numbers) sorts the whole array numbers/ Invoking sot(chars, 1,3) sorts a partial array from chars[1] to chars[3-1].parallelSort is more efficient if your computer has multiple processors. You can use the binarySearch method to search for a key in an array. The array must be presoted in increasing order. If the key is not in the array, the method returns –(insertionIndex + 1). For example, the following code searches the keys in an array of integers and an array of characters:

int[] list = {2,4,7,10,11,45,50,59,60,66,70,79};

System.out.println(“1. Index is “ + java.util.Arrays.binarySearch(list, 11));

System.out.println(“2. Index is “ + java.util.Arrays.binarySearch(list, 12));

Char[] chars = { ‘a’,’c’,’g’,’x’,’y’,’z’};

System.out.println(“3. Index is “ + java.util.Arrays.binarySearch(chars, ‘a’));

System.out.println(“4. Index is “ + java.util.Arrays.binarySearch(chars, ‘t’));

The out of the preceding code is as follows:

Index is 4

Index is -6

Index is 0

Index is -4

You can use the equals method to check whether two arrays are strictly equal. Two arrays are strictly equal if their corresponding elements are the same. In the following code, list1 and list2 are equal, but list2 and list3 are not.

7.13 Command-Line Arguments

The main method can receive string arguments from the command line.

Perhaps you have already noticed the unusual header for the main method, which has the parameter args of the String[] type. It is clear that args is an array of strings. The main method id just like a regular method with a parameter. You can call a regular method by passing actual parameters. Can you pass arguments to main? Yes, of course you can. In the following examples, the main method in class TestMain is invoked by a method in A:

public class A {

Public static void main(String[] args) {

String[] strings = {“New York”, “Boston”, “Atlanta”};

TestMain.main(strings);

}

}

public class TestMain {

public static void main(String[] args) {

For (int I = 0; I < args.length; i++) {

System.out.println(args[i]);

}

}

}

A main method is just like a regular method. Furthermore, you can pass arguments ot a main method from the command line.

7.13.1 Passing String to the main Method

You can pass strings to a main method from the command line when you run the program. The following command line, for example, starts the program TestMain with three strings: ar0, arg1, and arg2:

java TestMain arg0 arg1 arg2

arg0, arg1, and arg2 are strings, but they don’t have to appear in double quotes on the command line. The strings are separated by a space. A string that contains a space must be enclosed in double quotes. Consider the following command line:

Java TestMain “First num” alpha 53

It starts the program with three strings: Frist num, alpha, and 53. Since First num is a string, it enclosed it is enclosed in double quotes. Note 53 is actually treated as a string. You can use “53” instead of 53 in the command line.

When the main method is invoked, the Java interpreter creates an array to hold command-line arguments and pass the array reference to args. For example, if you invoke a command line with n arguments, the Java interpreter creates an array such as the one that follows:

Args = new String[n];

The java interpreter than passes args to invoke the main method.

\*\*\*Please Refer to Case Study 7.13.2 \*\*\*