Learner Guide

FACULTY OF INFORMATION TECHNOLOGY

Faculty of Information Technology

Year 3 ***Semester 5***

RICHFIELD

richfield.ac.za

FACULTY OF INFORMATION TECHNOLOGY

LEARNER GUIDE

MODULE: PROGRAMMING 631(1ST SEMESTER)

PREPARED ON BEHALF OF
RICHFIELD GRADUATE INSTITUTE OF TECHNOLOGY (PTY) LTD

TABLE OF CONTENTS

SECTION A: PREFACE

1. WELCOME

Welcome to the Faculty of Media Information and Communication Technology at 5*,. We trust you will find the contents and learning outcomes
of this module both interesting and insightful as you begin your academic journey and eventually your carreer in the Information Technology realm.

This section of the study guide is intended to orientate you to the module before the commencement of formal lectures.

The following lectures will focus on the common study units described:

SECTION A: WELCOME & ORIENTATION	
Study unit 1: Orientation Programme Introducing academic staff to the learners by Academic Programme Manager. Introduction of Institution Policies.	**Lecture 1**
Study unit 2: Orientation of Learners to Library and Students Facilities Introducing learners to physical infrastructure	**Lecture 2**
Study unit 3: Distribution and Orientation of JAVA 731 **Learner Guides, Textbooks and Prescribed Materials**	**Lecture 3**
Study unit 4: Discussion on the Objectives and Outcomes of **JAVA** 731	**Lecture 4**
Study unit 5: Orientation and guidelines to completing Assignments Review and Recap of Study units 1-4 and special guideline to late registrations	**Lecture 5**

2. TITLE OF MODULES, COURSE, CODE, NQF LEVEL, CREDITS & MODE OF DELIVERY

1st Semester	Bachelor of Science in Information Technology
Title Of Module: **Code:** **NQF Level:** **Credits:** **Mode of Delivery:**	Java 731 Java 731 7 15 Contact

3. PURPOSE OF MODULE

3.1 JAVA

Learners will be guided through the process of being able to solve real life problems sequentially as the dynamics of this programming language will be put into play. Each and every chapter has a correlation with the next chapter.

3.2 JAVA 631 (1st Semester)

This module provides learners with an informed understanding of JAVA programming concepts.

4. LEARNING OUTCOMES

On completion of these modules, learners should have a fundamental practical and theoretical knowledge of:
JAVA Programming Language

- Language Basics
- Objects
- Variables, Arrays, Operators and Control Statements.
- Classes and Inheritance
- Error Handling
- Files

5. METHOD OF STUDY

Only the key sections that have to be studied are indicated under each topic in this study guide are expected to have a thorough working knowledge of the prescribed text book. These form the basis for tests, assignments and examinations. To be able to do the activities and assignments for this module, and to achieve the learning outcomes and ultimately to be successful in the tests and examination, you will need an in-depth understanding of the the content of these sections in the learning guide and the prescribed books. In order to master the learning material, you must accept responsibility for your own studies. Learning is not the same as memorising. You are expected to show that you understand and are able to apply the information. Use will also be made of lectures, tutorials, case studies and group discussions to present this module.

6. LECTURES AND TUTORIALS

Learners must refer to the notice boards on their respective campuses for details of the lecture and tutorial time tables. The lecturer assigned to the module will also inform you of the number of lecture periods and tutorials allocated to a particular module. Prior preparation is required for each lecture and tutorial. Learners are encouraged to actively participate in lectures and tutorials in order to ensure success in tests, assignments and examinations.

7. NOTICES

All information pertaining to these modules such as tests dates, lecture and tutorial time tables, assignments, examinations etc will be displayed on the notice board located on your campus. Learners must check the notice board on a daily basis. Should you require any clarity, please consult your lecturer, or programme manager, or administrator on your respective campus.

8. PRESCRIBED & RECOMMENDED MATERIAL

8.1 .Prescribed Material:

8.1.1 JAVA How to Program by Deitel, 6th Edition, 2005, ISBN 0-13-129014-2

The purchasing of prescribed books is for the learners own account and is compulsory for all learners. This guide will have limited value if not accompanied by the prescribed text books.

8.2. Recommended Materials

8.2.1. Cay S. Horstmann, Gary Conell- Core Java 2 Volume 1- Fundamentals, 5th Edn. PHI, 2000.

8.2.2. P.Naughton and H.Schildt-Java 2 (The Complete Reference)-Third Edition, TMH, 1999.

8.2.3. K.Arnold and J.Gosling-The Java Programming Language- Second edition, Addison Wesley, 1996.

NB: Learners please note that there will be a limited number of copies of the recommended texts and reference material that will be made available at your campus library. Learners are advised to make copies or take notes of the relevant information, as the content matter is examinable.

8.3. Independent Research:

The student is encouraged to undertake independent research.

8.4. Library Infrastructure

The following services are available to you:

8.4.1. Each campus keeps a limited quantity of the recommended reading titles and a larger variety of similar titles which you may borrow. Please note that learners are required to purchase the prescribed materials.

8.4.2. Arrangements have been made with municipal, state and other libraries to stock our recommended reading and similar titles. You may use these on their premises or borrow them if available. It is your responsibility to safe keep all library books.

8.4.3. PCT&BC has also allocated one library period per week as to assist you with your formal research under professional supervision.

8.4.4. The computers laboratories, when not in use for academic purposes, may also be used for research purposes. Booking is essential for all electronic library usage.

9. ASSESSMENT

Final Assessment for this module will comprise two Continuous Assessment tests, an assignment and an examination. Your lecturer will inform you of the dates, times and the venues for each of these. You may also refer to the notice board on your campus or the Academic Calendar which is displayed in all lecture rooms.

9.1. Continuous Assessment Tests

There are two compulsory tests for each module (in each semester).

9.2. Assignment

There is one compulsory assignment for each module in each semester. Your lecturer will inform you of the Assessment questions at the commencement of this module.

9.3. Examination

There is one two hour examination for each module. Make sure that you diarize the correct date, time and venue. The examinations department will notify you of your results once all administrative matters are cleared and fees are paid up.

The examination may consist of multiple choice questions, short questions and essay type questions. This requires you to be thoroughly prepared as all the content matter of lectures, tutorials, all references to the prescribed text and any other additional documentation/reference materials is examinable in both your tests and the examinations.

The examination department will make available to you the details of the examination (date, time and venue) in due course. You must be seated in the examination room 15 minutes before the commencement of the examination. If you arrive late, you will not be allowed any extra time. Your learner registration card must be in your possession at all times.

9.4. Final Assessment

The final assessment for this module will be weighted as follows:

Continuous Assessment Test 1	
Continuous Assessment Test 2	40 %
Assignment 1	
Total Continuous Assessment	40%
Semester Examinations	60%
Total	100%

9.5. Key Concepts in Assignments and Examinations

In assignment and examination questions you will notice certain key concepts (i.e. words/verbs) which tell you what is expected of you. For example, you may be asked in a question to list, describe, illustrate, demonstrate, compare, construct, relate, criticize, recommend or design particular information / aspects / factors /situations. To help you to know exactly what these key concepts or verbs mean so that you will know exactly what is expected of you, we present the following taxonomy by Bloom, explaining the concepts and stating the level of cognitive thinking that theses refer to.

COMPETENCE	SKILLS DEMONSTRATED
Knowledge	Observation and recall of information Knowledge of dates, events, places Knowledge of major ideas Mastery of subject matter ***Question*** ***Cues*** list, define, tell, describe, identify, show, label, collect, examine, tabulate, quote, name, who, when, where, etc.

Comprehension	Understanding information Grasp meaning Translate knowledge into new context Interpret facts, compare, contrast Order, group, infer causes predict consequences ***Question*** ***Cues*** summarize, describe, interpret, contrast, predict, associate, distinguish, estimate, differentiate, discuss, extend
Application	Use information Use methods, concepts, theories in new situations Solve problems using required skills or knowledge ***Questions*** ***Cues*** apply, demonstrate, calculate, complete, illustrate, show, solve, examine, modify, relate, change, classify, experiment, discover
Synthesis	Use old ideas to create new ones Generalize from given facts Relate knowledge from several areas Predict, draw conclusions ***Question*** ***Cues*** combine, integrate, modify, rearrange, substitute, plan, create, design, invent, what if?, compose, formulate, prepare, generalize, rewrite
Evaluation	Compare and discriminate between ideas Assess value of theories, presentations Make choices based on reasoned argument Verify value of evidence recognize subjectivity ***Question*** ***Cues*** assess, decide, rank, grade, test, measure, recommend, convince, select, judge, explain, discriminate, support, conclude, compare, summarize

10. SPECIMEN ASSIGNMENT COVER SHEET

JAVA 731 (1st Semester)

Assignment Cover Sheet(To be attached to all Assignments – hand written or typed)
Name of Learner:………………………… Student No: …………………………..
Module:…………………………………… Date: ………………………………….
ICAS Number ………………………… ID Number …………………………..

The purpose of an assignment is to ensure that one is able to:

- Interpret, convert and evaluate text.
- Have sound understanding of key fields viz principles and theories, rules, concepts and awareness of how to cognate areas.
- Solve unfamiliar problems using correct procedures and corrective actions.
- Investigate and critically analyze information and report thereof.
- Present information using Information Technology.
- Present and communicate information reliably and coherently.
- Develop information retrieval skills.
- Use methods of enquiry and research in a disciplined field.

ASSESSMENT CRITERIA
(NB: The allocation of marks below may not apply to certain modules like EUC and Accounting) **A. Content- Relevance.**

Question Number	**Mark Allocation**	**Examiner's Mark**	**Moderator's Marks**	**Remarks**
1				
2				
3				
4				
5				
6				
7				
8				
9				
10				
Sub Total	**70 Marks**			
B. Research (A minimum of "**TEN SOURCES**" is recommended)				
Library, EBSCO, Emerald Journals, Internet, Newspapers, Journals, Text Books, Harvard method of referencing				
Sub Total	**15 Marks**			
C. Presentation				
Introduction, Body, Conclusion, Paragraphs, Neatness, Integration, Grammar / Spelling, Margins on every page, Page Numbering, Diagrams, Tables, Graphs, Bibliography				
Sub Total	**15 Marks**			
Grand Total	**100Marks**			

NB: All Assignments are compulsory as it forms part of continuous assessment that goes towards the final mark.

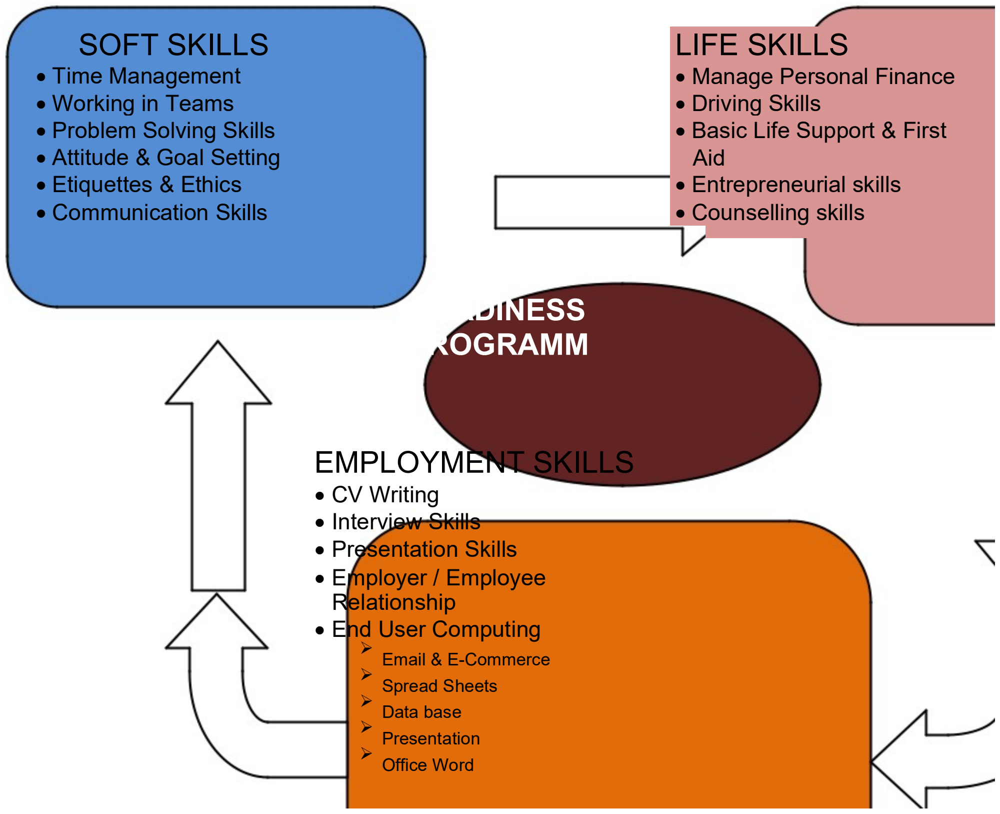
11. WORK READINESS PROGRAMME (WRP)

In order to prepare learners for the world of work, a series of interventions over and above the formal curriculum, are concurrently implemented to prepare learners.

These include:

- Soft skills
- Employment skills
- Life skills
- End –User Computing (if not included in your curriculum)

The illustration below outlines some of the key concepts for Work Readiness that will be included in your timetable.



It is in your interest to attend these workshops, complete the Work Readiness Log Book and prepare for the Working World.

12. WORK INTEGRATED LEARNING (WIL)

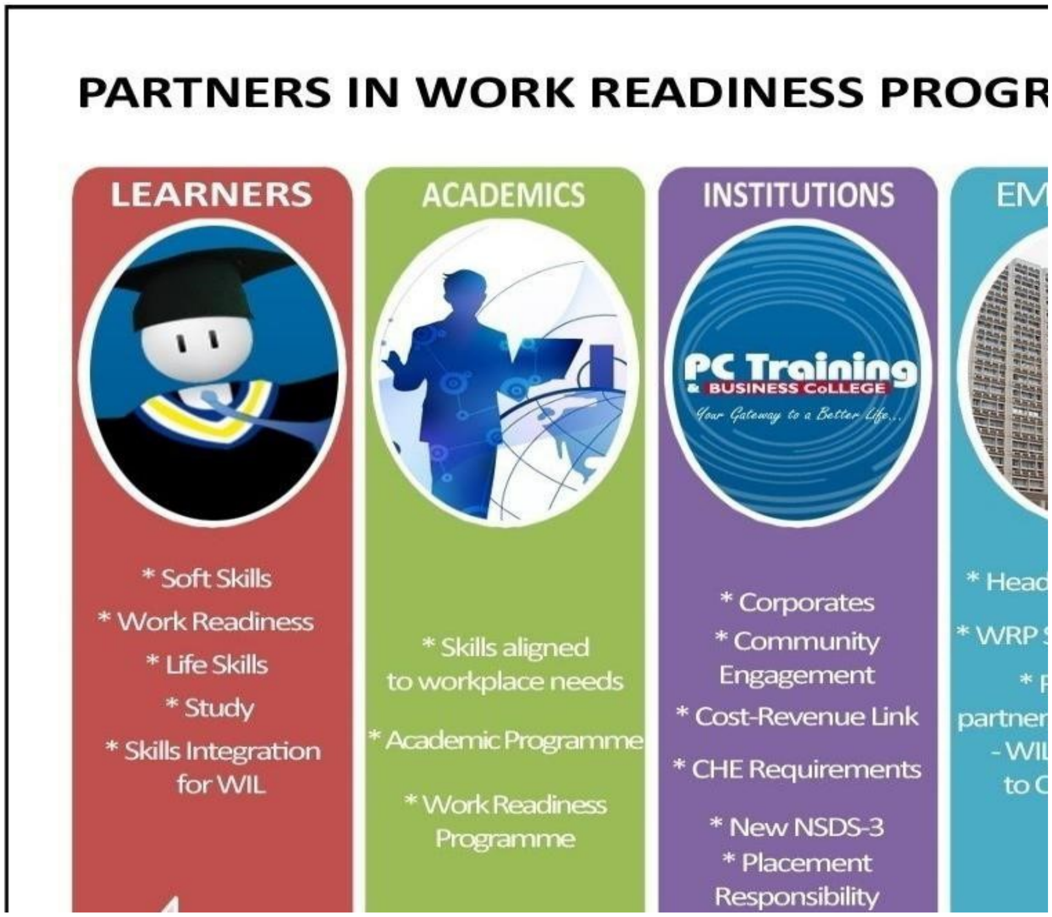
Work Integrated Learning forms a core component of the curriculum for the completion of this programme. All modules making of the Diploma/BSc in Information Technology will be assessed in an integrated manner towards the end of the programme or after completion of all other modules.

Prerequisites for placement with employers will include:

- Completion of all tests & assignment
- Success in examination
- Payment of all arrear fees
- Return of library books, etc.
- Completion of the Work Readiness Programme.

Learners will be fully inducted on the Work Integrated Learning Module, the Workbooks & assessment requirements before placement with employers.

The partners in Work Readiness Programme (WRP) include:



Good luck with your studies…

LEARNER GUIDE

MODULE: Programming 631 (Java) (1ST SEMESTER)

TOPIC 1 : THE INTRODUCTION OF JAVA PROGRAMMING ,DATA TYPES AND VARIABLES	
1.1 The Creation of Java	
1.2 Java Applications	
1.3 JVM Importance	**Lectures**
1.4 Object Oriented Programming	**6 -10**
1.5 First simple Java Program	
1.6 Second Java Simple	
1.7 The Simple Types	
1.8 Primitive Data Types	
1.9 Variables	
TOPIC 2:OPERATORS AND CONTROL STRUCTURES	
2.1 Arithmetic Operators	
2.2 The basic Arithmetic Operators	
2.3 Increment and Decrement Operators	
2.4 Relational Operators	**Lecture**
2.5 The Assignment Operators	**11-17**
2.6 Control Statements	
2.7 Iteration Statements	
Review Questions	
TOPIC 3: ARRAYS AND ARRAYLISTS	.
3.1 Dynamic Initialization	**Lecture**
3.2 Arrays	**18 - 22**
3.3 Strings	
3.4 Array lists	
Review Questions	
TOPIC 4: INTRODUCING CLASSES AND METHODS	
4.1 Introducing Classes.	
4.2 Declaring Objects.	
4.3 Introducing Methods.	
4.4 Constructors.	**Lecture**
4.5 The This Keyword	**23- 29**
4.6 Garbage Collection	
4.7 Understanding the static	
TOPIC 5: INHERITANCE AND POLYMORPHISM	
5.1 Inheritance	
5.2 Inheritance Basics	
5.3 Polymorphism	
5.4 Method Overloading	
5.5 Method Overriding	**Lecture**
5.6 Details about Overridden methods	**30-33**
5.7 Exception Handling	
5.8 Exception Types	
5.9 Java's Built-in Exception	
TOPIC 6: EXCEPTION HANDLING AND FILES	
6.1 Exception Handling Fundamentals	
6.2 Exception Types	
6.3 Using Mulltiple Exception	**Lecture**
6.4 Streams	**34 - 40**
6.5 Byte Streams Classes	
6.6 Character Stream Classes	
Review Questions	

TOPIC 7: MULTITHREAD PROGRAMMING	
7.1 Multi-Threaded Programming	**Lecture 40- 44**
7.2 Synchronization	
7.3 Collection Framework, Java.Util Package	
Review Questions	

TOPIC 1

1. THE INTRODUCTION OF JAVA PROGRAMMING, DATA TYPES AND VARIABLES

LEARNING OUTCOMES

After studying this topic you should be able to:

- Describe features of Java Programming.
- Difference between The Java Applets and Applications.
- Understand Importance of JVM.
- Explain Object Oriented Programming Concepts.
- Create a simple Java Program.
- Describe features of Data types, Variables and Arrays.
- Understand Primitive data types syntaxes and its importance in programming.
- Declare Variables
- Understand Dynamic Initialization.
-

1.1. THE CREATION OF JAVA

Java was conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems, Inc. in 1991. It took 18 months to develop the first working version. This language was initially called "Oak" but was renamed "Java" in 1995. Between the initial implementation of Oak in the fall of 1992 and the public announcement of Java in the spring of 1995, many more people contributed to the design and evolution of the language. Bill Joy, Arthur van Hoff, Jonathan Payne, Frank Yellin, and Tim Lindholm were key contributors to the maturing of the original prototype. Somewhat surprisingly, the original impetus for Java was not the Internet! Instead,the primary motivation was the need for a platform-independent (that is, architectureneutral) language that could be used to create software to be embedded in various consumer electronic devices, such as microwave ovens and remote controls. As you can probably guess, many different types of CPUs are used as controllers. The trouble with C and C++ (and most other languages) is that they are designed to be compiled for a specific target. Although it is possible to compile a C++ program for just about any type of CPU, to do so requires a full C++ compiler targeted for that CPU. The problem is that compilers are expensive and time-consuming to create. An easier— and more cost-efficient—solution was needed. In an attempt to find such a solution, Gosling and others began work on a portable, platform-independent language that could be used to produce code that would run on a variety of CPUs under differing environments. This effort ultimately led to the creation of Java.

Java's Lineage

Java is related to C++, which is a direct descendent of C. Much of the character of Java is inherited from these two languages. From C, Java derives its syntax. Many of Java's object-oriented features were influenced by C++. In fact, several of Java's defining characteristics come from—or are responses to—its predecessors. Moreover, the creation of Java was deeply rooted in the process of refinement and adaptation that has been occurring in computer programming languages for the past three decades. For these reasons, this section reviews the sequence of events and forces that led up to Java. As you will see, each innovation in language design was driven by the need to solve a fundamental problem that the preceding languages could not solve. Java is no exception.

Why Java Is Important to the Internet

The Internet helped catapult Java to the forefront of programming, and Java, in turn,has had a profound effect on the Internet. The reason for this is quite simple: Java expands the universe of objects that can move about freely in cyberspace. In a network, two very broad categories of objects are transmitted between the server and your personal computer: passive information and dynamic, active programs. For example, when you read your e-mail, you are viewing passive data. Even when you download a program, the program's code is still only passive data until you execute it. However, a second type of object can be transmitted to your computer: a dynamic, self-executing program. Such a program is an active agent on the client computer, yet is initiated by the server. For example, a program might be provided by the server to display properly the data that the server is sending.

As desirable as dynamic, networked programs are, they also present serious problems in the areas of security and portability. Prior to Java, cyberspace was effectively closed to half the entities that now live there. As you will see, Java addresses those concerns and, by doing so, has opened the door to an exciting new form of program: the applet.

1.2. JAVA APPLICATIONS

Java can be used to create two types of programs: applications and applets. But Java applets are deprecated. An *application* is a program that runs on your computer, under the operating system of that computer. That is, an application created by Java is more or less like one created using C or C++. When used to create applications, Java is not much different from any other computer language.

Java's Magic: The Bytecode

The key that allows Java to solve both the security and the portability problems just described is that the output of a Java compiler is not executable code. Rather, it is bytecode. *Bytecode* is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called the *Java Virtual Machine* (*JVM*). That is, in its standard form, the JVM is an *interpreter for bytecode.* This may come as a bit of a surprise. As you know, C++ is compiled to executable code. In fact, most modern languages are designed to be compiled, not interpreted—mostly because of performance concerns. However, the fact that a Java program is executed by the JVM helps solve the major problems associated with downloading programs over the Internet. Here is why.

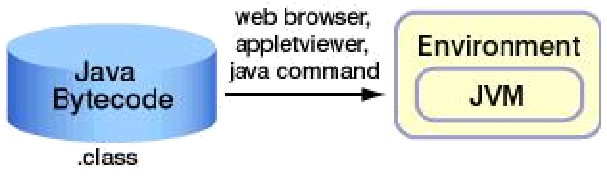
Translating a Java program into bytecode helps makes it much easier to run a program in a wide variety of environments. The reason is straightforward: only the JVM needs to be implemented for each platform. Once the run-time package exists for a given system, any Java program can run on it. Remember, although the details of the JVM will differ from platform to platform, all interpret the same Java bytecode. If a Java program were compiled to native code, then different versions of the same program would have to exist for each type of CPU connected to the Internet. This is, of course, not a feasible solution. Thus, the interpretation of bytecode is the easiest way to create truly portable programs. The fact that a Java program is interpreted also helps to make it secure. Because the execution of every Java program is under the control of the JVM, the JVM can contain the program and prevent it from generating side effects outside of the system. As you will see, safety is also enhanced by certain restrictions that exist in the Java language. When a program is interpreted, it generally runs substantially slower than it would run if compiled to executable code. However, with Java, the differential between the two is not so great.

The use of bytecode enables the Java run-time system to execute programs much faster than you might expect.

1.3 <u>JVM IMPORTANCE</u>

JVM is the environment in which java programs execute.It is a software that is implemented on top of real hardware and of standard operating system. A JVM can also implement programming languages other than Java. For example, Ada source code can be compiled to Java bytecode, which may then be executed by a JVM. JVMs can also be released by other companies besides Sun (the developer of Java) — JVMs using the "Java" trademark may be developed by other companies as long as they adhere to the JVM specification published by Sun (and related contractual obligations). Java was conceived keeping in mind the concept of WORA: "write once and run anywhere". This is possible using the Java Virtual Machine. The JVM is the environment in which java programs execute. It is software that is implemented on top of real hardware and of standard operating-systems.

JVM is a crucial component of the Java Platform, and because JVMs are available for many hardware and software platforms, Java can be both middleware and a platform in its own right[clarification needed] — hence the trademark write once, run anywhere. The use of the same bytecode for all platforms allows Java to be described as "compile once, run anywhere", as opposed to "write once, compile anywhere", which describes cross-platform compiled languages. A JVM also enables such features as automated exception handling,



which provides "root-cause" debugging information for every software error (exception), independent of the source code.

A JVM is distributed along with a set of standard class libraries that implement the Java API (Application Programming Interface). Appropriate APIs bundled together form the Java Runtime Environment (JRE).

Java development process
The javac command compiles Java source code (.java) into *bytecode* (.class).

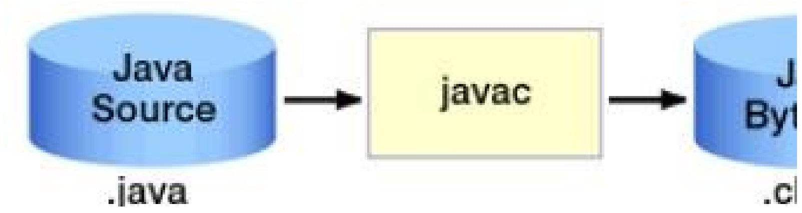


Figure 1.1

These bytecodes are loaded and executed in the Java virtual machine (JVM), which is embeddable within other environments, such as Web browsers and operating systems.

Figure 1.2

The Java Buzzwords
No discussion of the genesis of Java is complete without a look at the Java buzzwords. Although the fundamental forces that necessitated the invention of Java are portability and security, other factors also played an important role in molding the final form of the language. The key considerations were summed up by the Java team in the following list of buzzwords:

- Simple
- Secure

- Portable
- Object-oriented
- Robust
- Multithreaded
- Architecture-neutral
- Interpreted
- High performance
- Distributed
- Dynamic

Simple

Java was designed to be easy for the professional programmer to learn and use effectively. Assuming that you have some programming experience, you will not find Java hard to master. If you already understand the basic concepts of object-oriented programming, learning Java will be even easier. Best of all, if you are an experienced C++ programmer, moving to Java will require very little effort. Because Java inherits the C/C++ syntax and many of the object-oriented features of C++, most programmers have little trouble learning Java. Also, some of the more confusing concepts from C++ are either left out of Java or implemented in a cleaner, more approachable manner. Beyond its similarities with C/C++, Java has another attribute that makes it easy to learn: it makes an effort not to have *surprising* features. In Java, there are a small number of clearly defined ways to accomplish a given task.

Object-Oriented

Although influenced by its predecessors, Java was not designed to be source-code compatible with any other language. This allowed the Java team the freedom to design with a blank slate. One outcome of this was a clean, usable, pragmatic approach to objects. Borrowing liberally from many seminal object-software environments of the last few decades, Java manages to strike a balance between the purist's "everything is an object" paradigm and the pragmatist's "stay out of my way" model. The object model in Java is simple and easy to extend, while simple types, such as integers, are kept as high-performance nonobjects.

Robust

The multiplatformed environment of the Web places extraordinary demands on a program, because the program must execute reliably in a variety of systems. Thus, the ability to create robust programs was given a high priority in the design of Java. To gain reliability, Java restricts you in a few key areas, to force you to find your mistakes early in program development. At the same time, Java frees you from having to worry about many of the most common causes of programming errors. Because Java is a strictly typed language, it checks your code at compile time. However, it also checks your code at run time. In fact, many hard-to-track-down bugs that often turn up in hard-to-reproduce run-time situations are simply impossible to create in Java. Knowing that what you have written will behave in a predictable way under diverse conditions is a key feature of Java.

Multithreaded

Java was designed to meet the real-world requirement of creating interactive,networked programs. To accomplish this, Java supports multithreaded programming, which allows you to write programs that do many things simultaneously. The Java run-time system comes with an elegant yet sophisticated solution for multiprocess synchronization that enables you to construct smoothly running interactive systems. Java's easy-to-use approach to multithreading allows you to think about the specific behavior of your program, not the multitasking subsystem.

Architecture-Neutral

A central issue for the Java designers was that of code longevity and portability. One of the main problems facing programmers is that no guarantee exists that if you write a program today, it will run tomorrow—even on the same machine. Operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction. The Java designers made several hard decisions in the Java language and the Java Virtual Machine in an attempt to alter this situation. Their goal was "write once; run anywhere, any time, forever." To a great extent, this goal was accomplished.

Interpreted and High Performance

As described earlier, Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode. This code can be interpreted on any system that provides a Java Virtual Machine. Most previous attempts at crossplatform solutions have done so at the expense of performance. Other interpreted systems, such as BASIC, Tcl, and PERL, suffer from almost insurmountable performance deficits. Java, however, was designed to perform well on very low-power CPUs. As explained earlier, while it is true that Java was engineered for interpretation, the Java bytecode was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler. Java run-time systems that provide this feature lose none of the benefits of the platform-independent code.

Distributed
Java is designed for the distributed environment of the Internet, because it handles TCP/IP protocols. In fact, accessing a resource using a URL is not much different from accessing a file. The original version of Java (Oak) included features for intraaddress - space messaging. This allowed objects on two different computers to execute procedures remotely. Java revived these interfaces in a package called *Remote Method Invocation* (*RMI*). This feature brings an unparalleled level of abstraction to client/server programming.

Dynamic
Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time. This makes it possible to dynamically link code in a safe and expedient manner. This is crucial to the robustness of the applet environment, in which small fragments of bytecode may be dynamically updated on a running system.

1.4 OBJECT ORIENTED PROGRAMMING

Object-oriented programming is at the core of Java. In fact, all Java programs are objectoriented— this isn't an option the way that it is in C++, for example. OOP is so integral to Java that you must understand its basic principles before you can write even simple Java programs. Therefore, this chapter begins with a discussion of the theoretical aspects of OOP.

Two Paradigms
As you know, all computer programs consist of two elements: code and data. Furthermore, a program can be conceptually organized around its code or around its data. That is,some programs are written around "what is happening" and others are written around "who is being affected." These are the two paradigms that govern how a program is constructed. The first way is called the *process-oriented model.* This approach characterizes a program as a series of linear steps (that is, code). The process-oriented model can be thought of as *code acting on data.* Procedural languages such as C employ this model to considerable success. However, as mentioned in Chapter 1, problems with this approach appear as programs grow larger and more complex. To manage increasing complexity, the second approach, called *object-oriented programming,* was conceived. Object-oriented programming organizes a program around its data (that is, objects) and a set of well-defined interfaces to that data. An object-oriented program can be characterized as *data*

controlling access to code. As you will see, by switching the controlling entity to data, you can achieve several organizational benefits.

The Three OOP Principles

All object-oriented programming languages provide mechanisms that help you implement the object-oriented model. They are encapsulation, inheritance, and polymorphism. Let's take a look at these concepts now.

Encapsulation

Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse. One way to think about encapsulation is as a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper. Access to the code and data inside the wrapper is tightly controlled through a well-defined interface. To relate this to the real world, consider the automatic transmission on an automobile. It encapsulates hundreds of bits of information about your engine, such as how much you are accelerating, the pitch of the surface you are on, and the position of the shift lever. You, as the user, have only one method of affecting this complex encapsulation: by moving the gear-shift lever. You can't affect the transmission by using the turn signal or windshield wipers, for example. Thus, the gear-shift lever is a well-defined (indeed,unique) interface to the transmission. Further, what occurs inside the transmission does not affect objects outside the transmission. For example, shifting gears does not turn on the headlights! Because an automatic transmission is encapsulated, dozens of car manufacturers can implement one in any way they please. However, from the driver's point of view, they all work the same. This same idea can be applied to programming. The power of encapsulated code is that everyone knows how to access it and thus can use it regardless of the implementation details—and without fear of unexpected side effects. In Java the basis of encapsulation is the class. Although the class will be examined in great detail later in this book, the following brief discussion will be helpful now. A *class* defines the structure and behavior (data and code) that will be shared by a set of objects. Each object of a given class contains the structure and behavior defined by the class, as if it were stamped out by a mold in the shape of the class. For this reason, objects are sometimes referred to as *instances of a class.* Thus, a class is a logical construct; an object has physical reality. When you create a class, you will specify the code and data that constitute that class. Collectively, these elements are called *members* of the class. Specifically, the data defined by the class are referred to as *member variables* or *instance variables.* The code that operates on that data is referred to as *member methods* or just *methods.* (If you are familiar with C/C++, it may help to know that what a Java programmer calls a *method,* a C/C++ programmer calls a *function.*) In properly written Java programs, the methods define how the member variables can be used. This means that the behavior and interface of a class are defined by the methods that operate on its instance data. Since the purpose of a class is to encapsulate complexity, there are mechanisms for hiding the complexity of the implementation inside the class. Each method or variable in a class may be marked private or public. The *public* interface of a class represents everything that external users of the class need to know, or may know. The *private* methods and data can only be accessed by code that is a member of the class. Therefore, any other code that is not a member of the class cannot access a private method or variable. Since the private members of a class may only be accessed by other parts of your program through the class' public methods, you can ensure that no improper actions take place. Of course, this means that the public interface should be carefully designed not to expose too much of the inner workings of a class.

Inheritance

Inheritance is the process by which one object acquires the properties of another object. This is important because it supports the concept of hierarchical classification. As mentioned earlier, most knowledge is made manageable by hierarchical (that is, top-down) classifications. For example, a Golden Retriever is part of the classification *dog,* which in turn is part of the *mammal* class, which is under the larger class *animal.* Without the use of hierarchies, each object would need to define all of its characteristics explicitly. However, by use of inheritance, an object need only define those qualities that make it unique within its class. It can inherit its general attributes from its parent. Thus, it is the inheritance mechanism that makes it possible for one object to be a specific instance of a more general case. Let's take a closer look at this process.

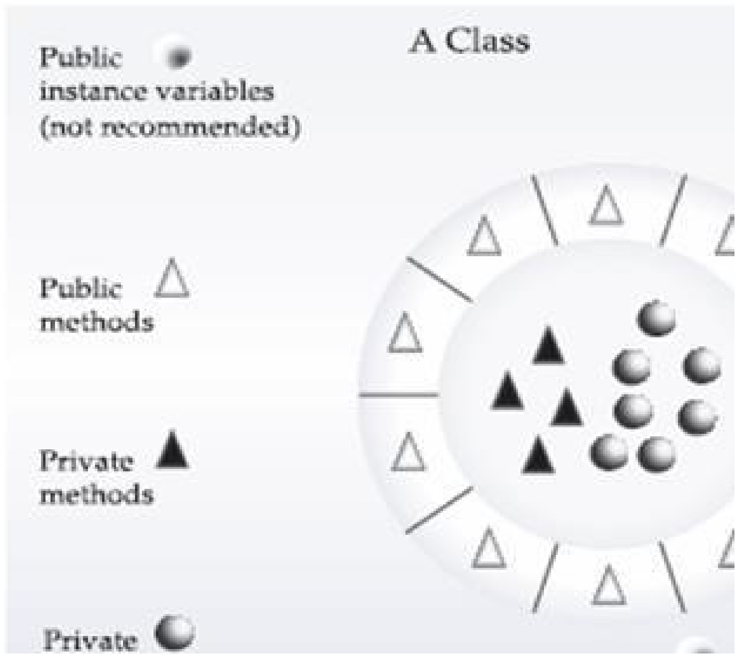


Figure 1.3

Most people naturally view the world as made up of objects that are related to each other in a hierarchical way, such as animals, mammals, and dogs. If you wanted to describe animals in an abstract way, you would say they have some attributes, such as size, intelligence, and type of skeletal system. Animals also have certain behavioral aspects; they eat, breathe, and sleep. This description of attributes and behavior is the *class* definition for animals.

If you wanted to describe a more specific class of animals, such as mammals, they would have more specific attributes, such as type of teeth, and mammary glands. This is known as a *subclass* of animals, where animals are referred to as mammals' *superclass.* Since mammals are simply more precisely specified animals, they *inherit* all of the attributes from animals. A deeply inherited subclass inherits all of the attributes from each of its ancestors in the *class hierarchy.*

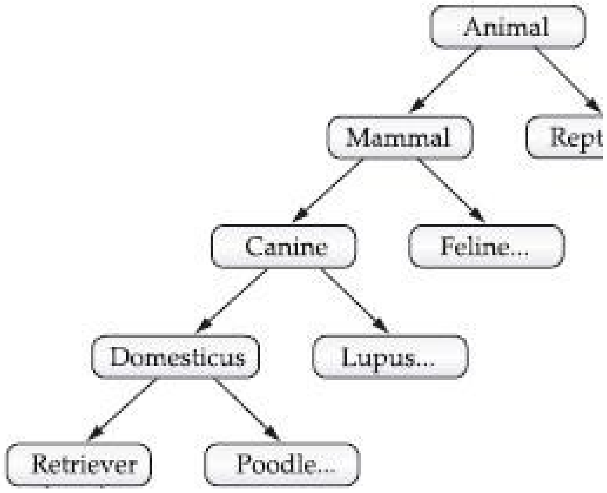


Figure 1.4

Inheritance interacts with encapsulation as well. If a given class encapsulates some attributes, then any subclass will have the same attributes *plus* any that it adds as part of its specialization (see Figure 2-2). This is a key concept which lets object-oriented programs grow in complexity linearly rather than geometrically. A new subclass inherits all of the attributes of all of its ancestors. It does not have unpredictable interactions with the majority of the rest of the code in the system.

Polymorphism

Polymorphism (from the Greek, meaning "many forms") is a feature that allows one interface to be used for a general class of actions. The specific action is determined by the exact nature of the situation. Consider a stack (which is a last-in, first-out list). You might have a program that requires three types of stacks. One stack is used for integer values, one for floating-point values, and one for characters. The algorithm that implements each stack is the same, even though the data being stored differs. In a non– object-oriented language, you would be required to create three different sets of stack routines, with each set using different names. However, because of polymorphism, in Java you can specify a general set of stack routines that all share the same names. More generally, the concept of polymorphism is often expressed by the phrase "one interface, multiple methods." This means that it is possible to design a generic interface to a group of related activities. This helps reduce complexity by allowing the same interface to be used to specify a *general class of action.* It is the compiler's job to select the *specific action* (that is, method) as it applies to each situation. You, the programmer, do not need to make this selection manually. You need only remember and utilize the general interface.

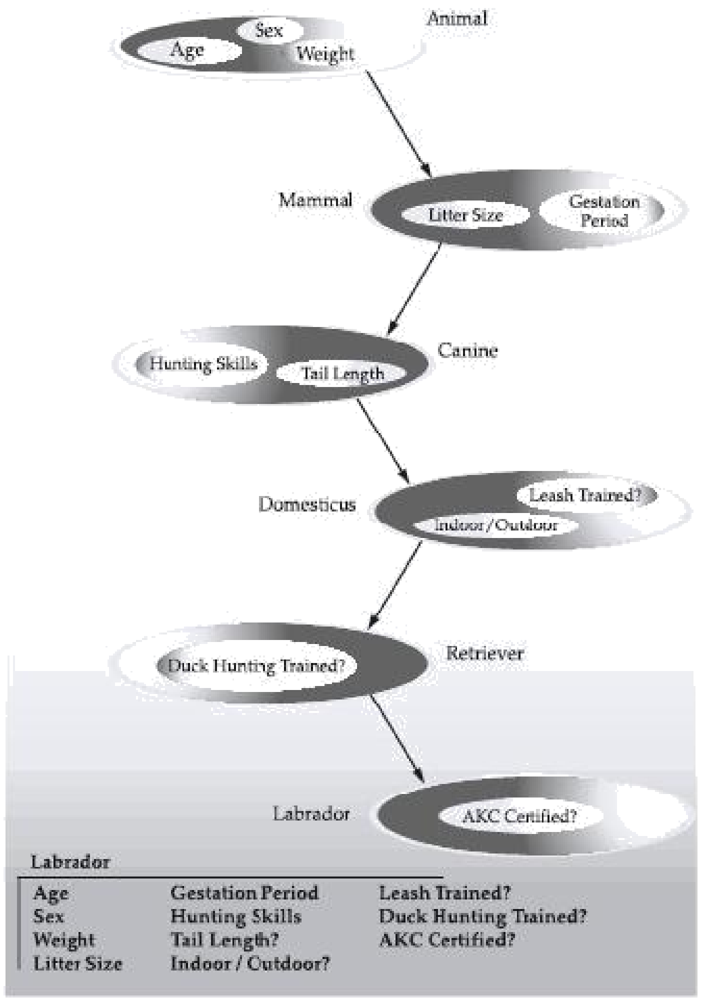


Figure 1.5 **Labrador inherites the incapsulation of all of ita superclasses**

Extending the dog analogy, a dog's sense of smell is polymorphic. If the dog smells a cat, it will bark and run after it. If the dog smells its food, it will salivate and run to its bowl. The same sense of smell is at work in both situations. The difference is what is being smelled, that is, the type of data being operated upon by the dog's nose! This same general concept can be implemented in Java as it applies to methods within a Java program.

1.5 A FIRST SIMPLE PROGRAM

Now that the basic object-oriented underpinning of Java has been discussed, let's look at some actual Java programs. Let's start by compiling and running the short sample program shown here. As you will see, this involves a little more work than you might imagine.

```
/*
This is a simple Java program.
Call this file "Example.java".
*/
class Example
```

```
{
// Your program begins with a call to
main(). public static void main(String args[])
 {
 System.out.println("This is a simple Java program.");
 }
}
```

Note:
The descriptions that follow use the standard Java 2 SDK (Software DevelopmentKit), which is available from Sun Microsystems. If you are using a different Javadevelopment environment, then you may need to follow a different procedurefor compiling and executing Java programs. In this case, consult your compiler'sdocumentation for details.

Entering the Program
For most computer languages, the name of the file that holds the source code to a program is arbitrary. However, this is not the case with Java. The first thing that you must learn about Java is that the name you give to a source file is very important. For this example, the name of the source file should be **Example.java**. Let's see why. In Java, a source file is officially called a *compilation unit.* It is a text file that contains one or more class definitions. The Java compiler requires that a source file use the **.java** filename extension. Notice that the file extension is four characters long. As you might guess, your operating system must be capable of supporting long filenames. This means that DOS and Windows 3.1 are not capable of supporting Java. However, Windows 95/98 and Windows NT/2000/XP work just fine.

As you can see by looking at the program, the name of the class defined by the program is also **Example**. This is not a coincidence. In Java, all code must reside inside a class. By convention, the name of that class should match the name of the file that holds the program. You should also make sure that the capitalization of the filename matches the class name. The reason for this is that Java is case-sensitive. At this point, the convention that filenames correspond to class names may seem arbitrary. However, this convention makes it easier to maintain and organize your programs.

Compiling the Program
To compile the **Example** program, execute the compiler, **javac**, specifying the name of the source file on the command line, as shown here:

```
C:\>javac Example.java
```

The **javac** compiler creates a file called **Example.class** that contains the bytecode version of the program. As discussed earlier, the Java bytecode is the intermediate representation of your program that contains instructions the Java interpreter will execute. Thus, the output of **javac** is not code that can be directly executed.To actually run the program, you must use the Java interpreter, called **java**. To do so, pass the class name **Example** as a command-line argument, as shown here:

```
C:\>java Example
```

When the program is run, the following output is displayed:

```
This is a simple Java program.
```

When Java source code is compiled, each individual class is put into its own output file named after the class and using the **.class** extension. This is why it is a good idea to give your Java source files the same name as the class they contain—the name of the source file will match the name of the **.class** file. When you execute the Java interpreter as just shown, you are actually specifying the name of the class that you want the interpreter to execute. It will automatically search for a file by that name that has the **.class** extension. If it finds the file, it will execute the code contained in the specified class.

A Closer Look at the First Sample Program

Although **Example.java** is quite short, it includes several key features which are common to all Java programs. Let's closely examine each part of the program. The program begins with the following lines:

```
/*
This is a simple Java program.
Call this file "Example.java".
*/
```

This is a *comment.* Like most other programming languages, Java lets you enter a remark into a program's source file. The contents of a comment are ignored by the compiler. Instead, a comment describes or explains the operation of the program to anyone who is reading its source code. In this case, the comment describes the program and reminds you that the source file should be called **Example.java**. Of course, in real applications, comments generally explain how some part of the program works or what a specific feature does. Java supports three styles of comments. The one shown at the top of the program is called a *multiline comment.* This type of comment must begin with /* and end with */. Anything between these two comment symbols is ignored by the compiler. As the name suggests, a multiline comment may be several lines long. The next line of code in the program is shown here:

```
class Example {
```

This line uses the keyword **class** to declare that a new class is being defined. **Example** is an *identifier* that is the name of the class. The entire class definition, including all of its members, will be between the opening curly brace ({) and the closing curly brace (}). The use of the curly braces in Java is identical to the way they are used in C, C++, and C#. For the moment, don't worry too much about the details of a class except to note that in Java, all program activity occurs within one. This is one reason why all Java programs are (at least a little bit) object-oriented.

The next line in the program is the *single-line comment,* shown here:

```
// Your program begins with a call to main().
```

This is the second type of comment supported by Java. A *single-line comment* begins with a // and ends at the end of the line. As a general rule, programmers use multiline comments for longer remarks and single-line comments for brief, line-by-line descriptions. The next line of code is shown here:

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

This line begins the **main( )** method. As the comment preceding it suggests, this is the line at which the program will begin executing. All Java applications begin execution by calling **main( )**. (This is just like C/C++.) The exact meaning of each part of this line cannot be given now, since it involves a detailed understanding of Java's approach to encapsulation. However, since most of the examples in the first part of this book will use this line of code, let's take a brief look at each part now.

The **public** keyword is an *access specifier,* which allows the programmer to control the visibility of class members. When a class member is preceded by **public**, then that member may be accessed by code outside the class in which it is declared. (The opposite of **public** is **private**, which prevents a member from being used by code defined outside of its class.) In this case, **main( )** must be declared as **public**, since it must be called by code outside of its class when the program is started. The keyword **static** allows **main( )** to be called without having to instantiate a particular instance of the class. This is necessary since **main( )** is called by the Java interpreter before any objects are made The keyword **void** simply tells the compiler that **main( )** does not return a value. As you will see, methods may also return values. If all this seems a bit confusing, don't worry. All of these concepts will be discussed in detail in subsequent chapters. As stated, **main( )** is the method called when a Java application begins. Keep in mind that Java is case-sensitive. Thus, **Main** is different from **main**. It is important to understand that the Java compiler will compile classes that do not contain a **main( )** method. But the Java interpreter has no way to run these classes. So, if you had typed **Main** instead of **main**, the compiler would still compile your program. However, the Java interpreter would report an error because it would be unable to find the **main( )** method.Any information that you need to pass to a method is received by variables specified within the set of parentheses that follow the name of the method. These variables are called *parameters.* If there are no parameters required for a given method, you still need to include the empty parentheses. In **main( )**, there is only one parameter, albeit a complicated one. **String args[ ]** declares a parameter named **args**, which is an array of instances of the class **String**. (*Arrays* are collections of similar objects.) Objects of type **String** store character strings. In this case, **args** receives any command-line arguments present when the program is executed. This program does not make use of this information, but other programs shown later in this book will. The last character on the line is the **{**. This signals the start of **main( )**'s body. All of the code that comprises a method will occur between the method's opening curly brace and its closing curly brace.

One other point: **main( )** is simply a starting place for your program. A complex program will have dozens of classes, only one of which will need to have a **main( )** method to get

things started. When you begin creating applets—Java programs that are embedded in Web browsers—you won't use **main( )** at all, since the Web browser uses a different means of starting the execution of applets.
The next line of code is shown here. Notice that it occurs inside **main( )**.

```
System.out.println("This is a simple Java program.");
```

This line outputs the string "This is a simple Java program." followed by a new line on the screen. Output is actually accomplished by the built-in **println( )** method. In this case, **println( )** displays the string which is passed to it. As you will see, **println( )** can be used to display other types of information, too. The line begins with **System.out**. While too complicated to explain in detail at this time, briefly, **System** is a predefined class that provides access to the system, and **out** is the output stream that is connected to the console. As you have probably guessed, console output (and input) is not used frequently in real Java programs and applets. Since most modern computing environments are windowed and graphical in nature, console I/O is used mostly for simple, utility programs and for demonstration programs. Later in this book, you will learn other ways to generate output using Java. But for now, we will continue to use the console I/O methods. Notice that the **println( )** statement ends with a semicolon. All statements in Java end with a semicolon. The reason that the other lines in the program do not end in a semicolon is that they are not, technically, statements. The first **}**

1.6 A SECOND SHORT PROGRAM

Perhaps no other concept is more fundamental to a programming language than that of a variable. As you probably know, a *variable* is a named memory location that may be assigned a value by your program. The value of a variable may be changed during the execution of the program. The next program shows how a variable is declared and how it is assigned a value. In addition, the program also illustrates some new aspects of console output. As the comments at the top of the program state, you should call this file **Example2.java**.

```
/*
Here is another short example.
Call this file "Example2.java".
*/
class Example2 {
public static void main(String args[]) {
int num; // this declares a variable called num
num = 100; // this assigns num the value 100
System.out.println("This is num: " + num);
num = num * 2;
System.out.print("The value of num * 2 is ");
System.out.println(num);
}
}
```

When you run this program, you will see the following output:

```
This is num: 100
The value of num * 2 is 200
```

Let's take a close look at why this output is generated. The first new line in the program is shown here:

```
int num; // this declares a variable called num
```

This line declares an integer variable called **num**. Java (like most other languages) requires that variables be declared before they are used.

Following is the general form of a variable declaration:

```
type var-name;
```

Here, *type* specifies the type of variable being declared, and *var-name* is the name of the variable. If you want to declare more than one variable of the specified type, you may use a comma-separated list of variable names. Java defines several data types, including integer, character, and floating-point. The keyword **int** specifies an integer type. In the program, the line:

```
num = 100; // this assigns num the value 100
```

assigns to **num** the value 100. In Java, the assignment operator is a single equal sign. The next line of code outputs the value of **num** preceded by the string "This is num:".

```
System.out.println("This is num: " + num);
```

In this statement, the plus sign causes the value of **num** to be appended to the string that precedes it, and then the resulting string is output. (Actually, **num** is first converted from an integer into its string equivalent and then concatenated with the string that precedes it. This process is described in detail later in this book.) This approach can be generalized. Using the + operator, you can string together as many items as you want within a single **println( )** statement.

The next line of code assigns **num** the value of **num** times 2. Like most other languages, Java uses the * operator to indicate multiplication. After this line executes, **num** will contain the value 200.Here are the next two lines in the program:

```
System.out.print("The value of num * 2 is ");
System.out.println(num);
```

Several new things are occurring here. First, the built-in method **print( )** is used to display the string "The value of num * 2 is ". This string is *not* followed by a newline. This means that when the next output is generated, it will start on the same line. The **print( )** method is just like **println( )**, except that it does not output a newline character after each call. Now look at the call to **println( )**. Notice that **num** is used by itself. Both **print( )** and **println( )** can be used to output values of any of Java's built-in types.

DATA TYPES

1.7. THE SIMPLE TYPES

Java defines eight simple (or elemental) types of data: **byte**, **short**, **int**, **long**, **char**, **float**, **double**, and **boolean**. These can be put in four groups:

- Integers: This group includes **byte**, **short**, **int**, and **long**, which are for whole valued signed numbers.
- Floating-point numbers: This group includes **float** and **double**, which represent numbers with fractional precision.
- Characters: This group includes **char**, which represents symbols in a character set, like letters and numbers.
- Boolean: This group includes **boolean**, which is a special type for representing

  true/false values.

You can use these types as-is, or to construct arrays or your own class types. Thus, they form the basis for all other types of data that you can create.

1.8. PRIMITIVE DATA TYPES

Integer Numbers

Java defines four integer types: **byte**, **short**, **int**, and **long**. All of these are signed, positive and negative values. Java does not support unsigned, positive-only integers. Many other computer languages, including C/C++, support both signed and unsigned integers. However, Java's designers felt that unsigned integers were unnecessary. Specifically, they felt that the concept of *unsigned* was used mostly to specify the behavior of the *high-order bit,* which defined the *sign* of an **int** when expressed as a number. As you will see in Chapter 4, Java manages the meaning of the high-order bit differently, by adding a special "unsigned right shift" operator. Thus, the need for an unsigned integer type was eliminated.The *width* of an integer type should not be thought of as the amount of storage itconsumes, but rather as the *behavior* it defines for variables and expressions of that type. The Java run-time environment is free to use whatever size it wants, as long as the types behave as you declared them. In fact, at least one implementation stores **byte**s and **short**s as 32-bit (rather than 8- and 16-bit) values to improve performance, because that is the word size of most computers currently in use. The width and ranges of these integer types vary widely, as shown in this table:

Name	Width	Range
long	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32	–2,147,483,648 to 2,147,483,647
short	16	–32,768 to 32,767
byte	8	–128 to 127

byte

The smallest integer type is **byte**. This is a signed 8-bit type that has a range from –128 to 127. Variables of type **byte** are especially useful when you're working with a stream of data from a network or file. They are also useful when you're working with raw binary data that may not be directly compatible with Java's other built-in types. Byte variables are declared by use of the **byte** keyword. For example, the following declares two **byte** variables called **b** and **c**:

```
byte b, c;
```

short

is a signed 16-bit type. It has a range from –32,768 to 32,767. It is probably the least-used Java type, since it is defined as having its high byte first (called *big-endian*format). This type is mostly applicable to 16-bit computers, which are becoming increasingly scarce.Here are some examples of **short** variable declarations:

```
short s;
short t;
```

int

The most commonly used integer type is **int**. It is a signed 32-bit type that has a range from –2,147,483,648 to 2,147,483,647. In addition to other uses, variables of type **int** are commonly employed to control loops and to index arrays. Any time you have an integer expression involving **byte**s, **short**s, **int**s, and literal numbers, the entire expression is *promoted* to **int** before the calculation is done.

long

is a signed 64-bit type and is useful for those occasions where an **int** type is not large enough to hold the desired value. The range of a **long** is quite large. This makes it useful when big, whole numbers are needed. For example, here is a program that computes the number of miles that light will travel in a specified number of days.

```
// Compute distance light travels using long
variables. class Light {
public static void main(String args[])
{ int lightspeed;
long days;
long seconds;
long distance;
// approximate speed of light in miles per
second lightspeed = 186000;
days = 1000; // specify number of days here seconds
= days * 24 * 60 * 60; // convert to seconds distance
= lightspeed * seconds; // compute distance
System.out.print("In " + days); System.out.print("
days light will travel about ");
System.out.println(distance + " miles.");
}
}
This program generates the following output:
In 1000 days light will travel about 16070400000000 miles.
```

Float-Point Numbers

float

The type **float** specifies a *single-precision* value that uses 32 bits of storage. Single precision is faster on some processors and takes half as much space as double precision, but will become imprecise when the values are either very large or very small. Variables of type **float** are useful when you need a fractional component, but don't require a large degree of precision. For example, **float** can be useful when representing dollars and cents.Here are some example **float** variable declarations:

```
float hightemp, lowtemp;
```

double
Double precision, as denoted by the **double** keyword, uses 64 bits to store a value. Double precision is actually faster than single precision on some modern processors that have been optimized for high-speed mathematical calculations. All transcendental math functions, such as **sin( )**, **cos( )**, and **sqrt( )**, return **double** values. When you need to maintain accuracy over many iterative calculations, or are manipulating large-valued numbers, **double** is the best choice.Here is a short program that uses **double** variables to compute the area of a circle:

```
// Compute the area of a
circle. class Area {
public static void main(String args[])
{ double pi, r, a;
r = 10.8; // radius of circle
pi = 3.1416; // pi, approximately
    a = pi * r * r; // compute area
System.out.println("Area of circle is " + a);
}
}
```

Characters
In Java, the data type used to store characters is **char**. However, C/C++ programmers beware: **char** in Java is not the same as **char** in C or C++. In C/C++, **char** is an integer type that is 8 bits wide. This is *not* the case in Java. Instead, Java uses Unicode to represent characters. *Unicode* defines a fully international character set that can represent all of the characters found in all human languages. It is a unification of dozens of character sets, such as Latin, Greek, Arabic, Cyrillic, Hebrew, Katakana, Hangul, and many more. For this purpose, it requires 16 bits. Thus, in Java **char** is a 16-bit type. The range of a **char** is 0 to 65,536. There are no negative **char**s. The standard set of characters known as ASCII still ranges from 0 to 127 as always, and the extended 8-bit character set, ISO-Latin-1, ranges from 0 to 255.
Here is a program that demonstrates **char** variables:

```
// Demonstrate char data
type. class CharDemo {
public static void main(String args[])
{ char ch1, ch2;
ch1 = 88; // code for X ch2 = 'Y';
System.out.print("ch1 and ch2: ");
System.out.println(ch1 + " " +
ch2);
}
}
```

This program displays the following output:

```
ch1 and ch2: X Y
```

Booleans

Java has a simple type, called **boolean**, for logical values. It can have only one of two possible values, **true** or **false**. This is the type returned by all relational operators, such as **a < b**. **boolean** is also the type *required* by the conditional expressions that govern the control statements such as **if** and **for**.
Here is a program that demonstrates the **boolean** type:

```
// Demonstrate boolean
values. class BoolTest {
public static void main(String args[])
{ boolean b;
b = false; System.out.println("b
is " + b); b = true;
System.out.println("b is " + b);

// a boolean value can control the if statement
if(b) System.out.println("This is executed.");
b = false;
if(b) System.out.println("This is not executed.");
// outcome of a relational operator is a boolean
value System.out.println("10 > 9 is " + (10 > 9));
}
}
```

The output generated by this program is shown here:

```
b is false
b is true
```

This is executed.
10 > 9 is true

1.9 VARIABLES

The variable is the basic unit of storage in a Java program. A variable is defined by the combination of an identifier, a type, and an optional initializer. In addition, all variables have a scope, which defines their visibility, and a lifetime. These elements are examined next.

Declaring a Variable

In Java, all variables must be declared before they can be used. The basic form of a variable declaration is shown here:

```
type identifier [ = value][, identifier [= value] ...] ;
```

The *type* is one of Java's atomic types, or the name of a class or interface. (Class and interface types are discussed later in Part I of this book.) The *identifier* is the name of the variable. You can initialize the variable by specifying an equal sign and a value. Keep in mind that the initialization expression must result in a value of the same (or compatible) type as that specified for the variable. To declare more than one variable of the specified type, use a comma-separated list.Here are several examples of variable declarations of various types. Note that some include an initialization.

```
int a, b, c; // declares three ints, a, b, and c.
int d = 3, e, f = 5; // declares three more ints,
initializing // d and f.
byte z = 22; // initializes z.
double pi = 3.14159; // declares an approximation of pi.
char x = 'x'; // the variable x has the value 'x'.
```

1.10. DYNAMIC INITIALIZATION

Although the preceding examples have used only constants as initializers, Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared.For example, here is a short program that computes the length of the hypotenuse of a right triangle given the lengths of its two opposing sides:

```
// Demonstrate dynamic
initialization. class DynInit {
public static void main(String args[])
{ double a = 3.0, b = 4.0;
// c is dynamically initialized
double c = Math.sqrt(a * a + b * b);
System.out.println("Hypotenuse is " + c);
}
}
```

Here, three local variables—**a**, **b**,and **c**—are declared. The first two, **a** and **b**, are initialized by constants. However, **c** is initialized dynamically to the length of the hypotenuse (using the Pythagorean theorem). The program uses another of Java's built-in methods, **sqrt( )**, which is a member of the **Math** class, to compute the square root of its argument. The key point here is that the initialization expression may use any element valid at the time of the initialization, including calls to methods, other variables, or literals.

The Scope and Lifetime of Variables

So far, all of the variables used have been declared at the start of the **main( )** method.However, Java allows variables to be declared within any block. As explained in lesson 2, a block is begun with an opening curly brace and ended by a closing curly brace. A block defines a *scope.* Thus, each time you start a new block, you are creating a new scope. As you probably know from your previous programming experience, a scope determines what objects are visible to other parts of your program. It also determinesthe lifetime of those objects.

LEARNER ACTIVITY 1

1. What are Features of Java Programming?
2. Difference between The Java Applets and Applications.
3. Explain the Importance of JVM.
4. Detail Explanation of Object Oriented Programming Concepts.
5. What is the role of Java in Internet?

TOPIC 2

2. OPERATORS AND CONTROL STATEMENTS

LEARNING OUTCOMES

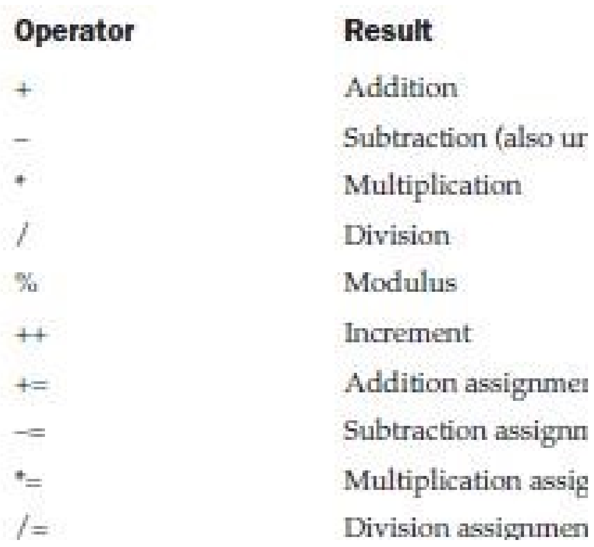
After studying this topic, you should be able to:

- Understand Operators and their importance in java programming.
- Understand The Basic Arithmetic Operators.
  - Increment and Decrement Operators.
  - Relational Operators.
  - The Assignment Operators.
- Understand Control Statements.
- Work with Iteration Statements.

Java provides a rich operator environment. Most of its operators can be divided into the following four groups: arithmetic, bitwise, relational, and logical. Java also defines some additional operators that handle certain special situations.

2.1 ARITHMETIC OPERATORS

Arithmetic operators are used in mathematical expressions in the same way that
they are used in algebra. The following table lists the arithmetic operators:



Operator	Result
+	Addition
–	Subtraction (also ur
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignmei
–=	Subtraction assignn
*=	Multiplication assig
/=	Division assignmen

2.2 THE BASIC ARITHMETIC OPERATORS

The basic arithmetic operations—addition, subtraction, multiplication, and division all behave as you would expect for all numeric types. The minus operator also has a unary form which negates its single operand. Remember that when the division operator is applied to an integer type, there will be no fractional component attached to the result. The following simple example program demonstrates the arithmetic operators. It also illustrates the difference between floating-point division and integer division.

```
// Demonstrate the basic arithmetic
operators. class BasicMath {
public static void main(String args[ ]) {
// arithmetic using integers
System.out.println("Integer
Arithmetic"); int a = 1 + 1;
int b = a * 3; int c = b / 4; int d
= c - a; int e = -d;
System.out.println("a = " + a);
System.out.println("b = " +
b); System.out.println("c = " +
c); System.out.println("d = " +
d); System.out.println("e = " +
e);

// arithmetic using doubles
System.out.println("\nFloating Point
Arithmetic"); double da = 1 + 1;
double db = da * 3; double dc =
db / 4; double dd = dc - a;
double de = -dd;
System.out.println("da = " + da);
System.out.println("db = " +
db); System.out.println("dc = " +
dc); System.out.println("dd = " +
dd); System.out.println("de = "
+ de);
}
}
```

When you run this program, you will see the following output:

```
Integer Arithmetic
    a = 2
    b = 6
    c = 1
    d = -1
    e = 1
```

Floating Point Arithmetic

```
da = 2.0
db = 6.0
dc = 1.5
dd = -0.5
de = 0.5
```

The Modulus Operator

The modulus operator, **%**, returns the remainder of a division operation. It can be applied to floating-point types as well as integer types. (This differs from C/C++, in which the **%** can only be applied to integer types.) The following example program demonstrates the **%**:

```
// Demonstrate the %
operator. class Modulus {
public static void main(String args[ ])
{ int x = 42;
double y = 42.25;
System.out.println("x mod 10 = " + x % 10);
System.out.println("y mod 10 = " + y % 10);
}
}
```

When you run this program you will get the following output:

```
x mod 10 = 2
y mod 10 = 2.25
```

Arithmetic Assignment Operators

Java provides special operators that can be used to combine an arithmetic operation with an assignment. As you probably know, statements like the following are quite common in programming:

```
a = a + 4;
```

In Java, you can rewrite this statement as shown here:

```
a += 4;
```

This version uses the += assignment operator. Both statements perform the same action: they increase the value of **a** by 4.

Here is another example,

```
a = a % 2;
```

which can be expressed as

```
a %= 2;
```

2.3. INCREMENT AND DECREMENT

The ++ and the – – are Java's increment and decrement operators. They were introduced in Chapter 2. Here they will be discussed in detail. As you will see, they have some special properties that make them quite interesting. Let's begin by reviewing precisely what the increment and decrement operators do. The increment operator increases its operand by one. The decrement operator decreases its operand by one. For example, this statement:

```
x = x + 1;
```

can be rewritten like this by use of the increment operator:

```
x++;
```

Similarly, this statement:

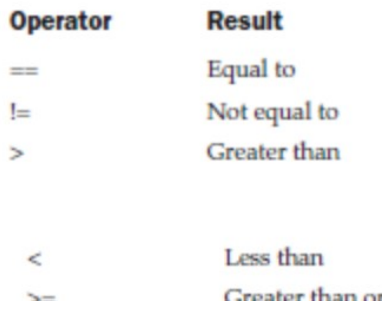
```
x = x - 1;
```

is equivalent to

```
x--;
```

2.4. RELATIONAL OPERATORS

The *relational operators* determine the relationship that one operand has to the other. Specifically, they determine equality and ordering. The relational operators are shown here:



Operator	Result
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or

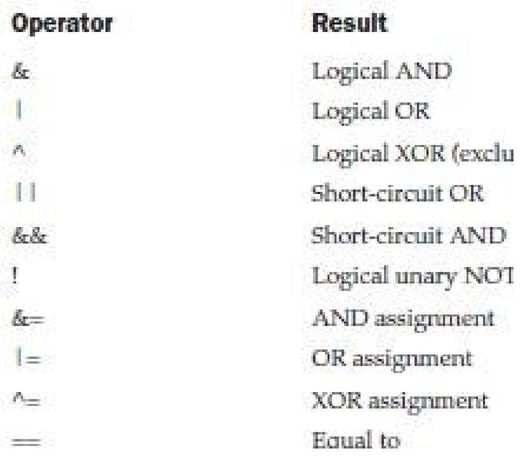
The outcome of these operations is a **boolean** value. The relational operators are most frequently used in the expressions that control the **if** statement and the various loop statements. Any type in Java, including integers, floating-point numbers, characters, and Booleans can be compared using the equality test, **==**, and the inequality test, **!=**. Notice that in Java (as in C/C++/C#) equality is denoted with two equal signs, not one. (Remember: a single equal sign is the assignment operator.) Only numeric types can be compared using the ordering operators. That is, only integer, floating-point, and character operands may be compared to see which is greater or less than the other. As stated, the result produced by a relational operator is a **boolean** value. For example, the following code fragment is perfectly valid:

```
int a = 4;
int b = 1;
boolean c = a < b;
```

In this case, the result of **a<b** (which is **false**) is stored in **c**.

Boolean Logical Operators

The Boolean logical operators shown here operate only on **boolean** operands. All of the binary logical operators combine two **boolean** values to form a resultant **boolean** value.



Operator	Result
&	Logical AND
|	Logical OR
^	Logical XOR (exclu
||	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT
&=	AND assignment
|=	OR assignment
^=	XOR assignment
==	Equal to

The logical Boolean operators, **&**, |, and ^, operate on **boolean** values in the same way that they operate on the bits of an integer. The logical **!** operator inverts the Boolean state: **!true == false** and **!false == true**. The following table shows the effect of each logical operation:

```
// Demonstrate the boolean logical
operators. class BoolLogic {
public static void main(String args[])
{ boolean a = true;
boolean  b  =  false;
boolean  c  =  a  |  b;
boolean d = a & b;
boolean e = a ^ b;
boolean f = (!a & b) | (a & !b); boolean
g = !a; System.out.println(" a = " + a);
System.out.println(" b = " + b);
System.out.println(" a|b = " + c);
System.out.println(" a&b = " + d);
System.out.println(" a^b = " + e);
System.out.println("!a&b|a&!b = " + f);
System.out.println(" !a = " + g);

}
}
```

After running this program, you will see that the same logical rules apply to **boolean** values as they did to bits. As you can see from the following output, the string representation of a Java **boolean** value is one of the literal values **true** or **false**:

```
a = true
b = false
a|b = true
a&b = false
a^b = true
a&b|a&!b = true
!a = false
```

2.5. THE ASSIGNMENT OPERATOR

You have been using the assignment operator since Topic 2. Now it is time to take a formal look at it. The *assignment operator* is the single equal sign, =. The assignment operator works in Java much as it does in any other computer language. It has this general form:

```
var = expression;
```

Here, the type of *var* must be compatible with the type of *expression.*
The assignment operator does have one interesting attribute that you may not be familiar with: it allows you to create a chain of assignments. For example, consider this fragment:

```
int x, y, z;
x = y = z = 100; // set x, y, and z to 100
```

This fragment sets the variables **x**, **y**, and **z** to 100 using a single statement. This works because the = is an operator that yields the value of the right-hand expression. Thus, the value of **z = 100** is 100, which is then assigned to **y**, which in turn is assigned to **x**. Using a "chain of assignment" is an easy way to set a group of variables to a common value.

2.6. CONTROL STATEMENTS

A programming language uses *control* statements to cause the flow of execution to advance and branch based on changes to the state of a program. Java's program control statements can be put into the following categories: selection, iteration,and jump. *Selection* statements allow your program to choose different paths of execution based upon the outcome of an expression or the state of a variable. *Iteration* statements enable program execution to repeat one or more statements (that is, iteration statementsform loops). *Jump* statements allow your program to execute in a nonlinear fashion. All of Java's control statements are examined here.

Java's Selection Statements

Java supports two selection statements: **if** and **switch**. These statements allow you to control the flow of your program's execution based upon conditions known only during run time. You will be pleasantly surprised by the power and flexibility contained in these two statements.

The If Statement

The **if** statement was introduced in Chapter 2. It is examined in detail here. The **if** statement is Java's conditional branch statement. It can be used to route program execution through two different paths. Here is the general form of the **if** statement:

```
if (condition) statement1;
else statement2;
```

Here, each *statement* may be a single statement or a compound statement enclosed in curly braces (that is, a *block*). The *condition* is any expression that returns a **boolean** value.The **else** clause is optional.The **if** works like this: If the *condition* is true, then *statement1* is executed. Otherwise, *statement2* (if it exists) is executed. In no case will both statements be executed. For example, consider the following:

```
int a, b;
// ...
if(a < b) a = 0;
else b = 0;
```

Here, if **a** is less than **b**, then **a** is set to zero. Otherwise, **b** is set to zero. In no case are they both set to zero.Most often, the expression used to control the **if** will involve the relational operators.However, this is not technically necessary. It is possible to control the **if** using a single **boolean** variable, as shown in this code fragment: boolean dataAvailable;

```
// ...
if (dataAvailable)
ProcessData();
else
waitForMoreData();
```

Remember, only one statement can appear directly after the **if** or the **else**. If you want to include more statements, you'll need to create a block, as in this fragment:
int bytesAvailable;

```
// ...
if (bytesAvailable > 0) {
ProcessData();
bytesAvailable -= n;
} else
waitForMoreData();
```

Here, both statements within the **if** block will execute if **bytesAvailable** is greater than zero. Some programmers find it convenient to include the curly braces when using the **if**,even when there is only one statement in each clause. This makes it easy to add another statement at a later date, and you don't have to worry about forgetting the braces. In fact, forgetting to define a block when one is needed is a common cause of errors. For example, consider the following code fragment:

```
int bytesAvailable;
// ...
if (bytesAvailable > 0) {
```

```
ProcessData();
bytesAvailable -= n;
} else
waitForMoreData();
bytesAvailable = n;
```

It seems clear that the statement **bytesAvailable = n;** was intended to be executed inside the **else** clause, because of the indentation level. However, as you recall, whitespace is insignificant to Java, and there is no way for the compiler to know what was intended. This code will compile without complaint, but it will behave incorrectly when run. The preceding example is fixed in the code that follows: int bytesAvailable;

```
// ...
if (bytesAvailable > 0) {
ProcessData();
bytesAvailable -= n;
} else {
waitForMoreData();
bytesAvailable = n;
}
```

Nested ifs

A *nested* **if** is an **if** statement that is the target of another **if** or **else**. Nested **if**s are very common in programming. When you nest **if**s, the main thing to remember is that an **else** statement always refers to the nearest **if** statement that is within the same block as the **else** and that is not already associated with an **else**. Here is an example:

```
if(i == 10) {
if(j < 20) a = b;
if(k > 100) c = d; // this if is
else a = c; // associated with this else
}
else a = d; // this else refers to if(i == 10)
```

As the comments indicate, the final **else** is not associated with **if(j<20)**, because it is not in the same block (even though it is the nearest **if** without an **else**). Rather, the final **else** is associated with **if(i==10)**. The inner **else** refers to **if(k>100)**, because it is the closest **if** within the same block.

The if-else-if Ladder

A common programming construct that is based upon a sequence of nested **if**s is the ***if-else-if*** *ladder*. It looks like this:

```
if(condition)
statement;
else if(condition)
statement;
```

```
else if(condition)
statement;
...
else
statement;
```

The **if** statements are executed from the top down. As soon as one of the conditions controlling the **if** is **true**, the statement associated with that **if** is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final **else** statement will be executed. The final **else** acts as a default condition; that is, if all other conditional tests fail, then the last **else** statement is performed. If there is no final **else** and all other conditions are **false**, then no action will take place.
Here is a program that uses an **if-else-if** ladder to determine which season a particular month is in.

```
// Demonstrate if-else-if
statements. class IfElse {
public static void main(String args[])
{ int month = 4; // April
String season;
if(month == 12 || month == 1 || month ==
2) season = "Winter";
else if(month == 3 || month == 4 || month ==
5) season = "Spring";
else if(month == 6 || month == 7 || month ==
8) season = "Summer";
else if(month == 9 || month == 10 || month ==
11) season = "Autumn";
else
season = "Bogus Month"; System.out.println("April
is in the " + season + ".");
}
}
```

Here is the output produced by the program:

```
April is in the Spring.
```

You might want to experiment with this program before moving on. As you will find, no matter what value you give **month**, one and only one assignment statement within the ladder will be executed.

Switch

The **switch** statement is Java's multiway branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression.
As such, it often provides a better alternative than a large series of **if-else-if** statements. Here is the general form of a **switch** statement:

```
switch (expression) {
case value1:
```

```
// statement
sequence break;
case value2:
// statement
sequence break;
...
case valueN:
// statement
sequence break;
default:
// default statement sequence
}
```

The *expression* must be of type **byte**, **short**, **int**, or **char**; each of the *values* specified in the **case** statements must be of a type compatible with the expression. Each **case** value must be a unique literal (that is, it must be a constant, not a variable). Duplicate **case** values are not allowed.The **switch** statement works like this: The value of the expression is compared with each of the literal values in the **case** statements. If a match is found, the code sequence following that **case** statement is executed. If none of the constants matches the value of the expression, then the **default** statement is executed. However, the **default** statementis optional. If no **case** matches and no **default** is present, then no further action is taken.The **break** statement is used inside the **switch** to terminate a statement sequence.When a **break** statement is encountered, execution branches to the first line of code that follows the entire **switch** statement. This has the effect of "jumping out" of the **switch**.

Here is a simple example that uses a **switch** statement:

```
// A simple example of the switch.
class SampleSwitch {
public static void main(String args[ ])
{ for(int i=0; i<6; i++)
switch(i) {
case 0:
System.out.println("i is
zero."); break;
case 1:
System.out.println("i is
one."); break;
case 2:
System.out.println("i is
two."); break;
case 3:
System.out.println("i is
three."); break;
default:
System.out.println("i is greater than 3.");
}
}
}
```

The output produced by this program is shown here:

```
i is zero.
i is one.
i is two.
i is three.
i is greater than 3.
i is greater than 3.
```

Iteration Statements

Java's iteration statements are **for**, **while**, and **do-while**. These statements create what we commonly call *loops.* As you probably know, a loop repeatedly executes the same set of instructions until a termination condition is met. As you will see, Java has a loop to fit any programming need.

While

The **while** loop is Java's most fundamental looping statement. It repeats a statement or block while its controlling expression is true. Here is its general form:

```
while(condition) {
// body of loop
}
```

The *condition* can be any Boolean expression. The body of the loop will be executed as long as the conditional expression is true. When *condition* becomes false, control passes to the next line of code immediately following the loop. The curly braces are unnecessary if only a single statement is being repeated.
Here is a **while** loop that counts down from 10, printing exactly ten lines of "tick":

```
// Demonstrate the while
loop. class While {
public static void main(String args[])
{ int n = 10;
while(n > 0) {
System.out.println("tick " +
n); n--;
}
}
}
```

When you run this program, it will "tick" ten times:

```
tick 10
tick 9
tick 8
tick 7
tick 6
tick 5
tick 4
tick 3
tick 2
tick 1
```

Since the **while** loop evaluates its conditional expression at the top of the loop, the body of the loop will not execute even once if the condition is false to begin with. For example, in the following fragment, the call to **println( )** is never executed:

```
int a = 10, b = 20;
while(a > b)
System.out.println("This will not be displayed");
```

The body of the **while** (or any other of Java's loops) can be empty. This is because a *null statement* (one that consists only of a semicolon) is syntactically valid in Java. For example, consider the following program:

```
// The target of a loop can be
empty. class NoBody {
public static void main(String args[ ])
{ int i, j;
i = 100;
j = 200;
// find midpoint between i and j
while(++i < --j) ; // no body in this loop
System.out.println("Midpoint is " + i);
}
}
```

This program finds the midpoint between **i** and **j**. It generates the following output:

```
Midpoint is 150
```

Here is how the **while** loop works. The value of **i** is incremented, and the value of **j** is decremented. These values are then compared with one another. If the new value of **i** is still less than the new value of **j**, then the loop repeats. If **i** is equal to or greater than **j**, the loop stops. Upon exit from the loop, **i** will hold a value that is midway between the original values of **i** and **j**. (Of course, this procedure only works when **i** is less than **j** to begin with.) As you can see, there is no need for a loop body; all of the action occurs within the conditional expression, itself. In professionally written Java code, short loops are frequently coded without bodies when the controlling expression can handle all of the details itself.

do-while

As you just saw, if the conditional expression controlling a **while** loop is initially false, then the body of the loop will not be executed at all. However, sometimes it is desirable to execute the body of a **while** loop at least once, even if the conditional expression is false to begin with. In other words, there are times when you would like to test the termination expression at the end of the loop rather than at the beginning. Fortunately, Java supplies a loop that does just that: the **do-while**. The **do-while** loop always executes its body at least once, because its conditional expression is at the bottom of the loop. Its general form is

```
do {
// body of loop
} while (condition);
```

Each iteration of the **do-while** loop first executes the body of the loop and then evaluates the conditional expression. If this expression is true, the loop will repeat. Otherwise, the loop terminates. As with all of Java's loops, *condition* must be a Boolean expression.
Here is a reworked version of the "tick" program that demonstrates the **do-while** loop. It generates the same output as before.

```
// Demonstrate the do-while
loop. class DoWhile {
public static void main(String args[])
{ int n = 10;
do {
System.out.println("tick " +
n); n--;
} while(n > 0);
}
}
The loop in the preceding program, while technically correct, can be written
more efficiently as follows:
do {
System.out.println("tick " +
n); } while(--n > 0);
```

In this example, the expression **(– –n > 0)** combines the decrement of **n** and the test for zero into one expression. Here is how it works. First, the **– –**n statement executes, decrementing **n** and returning the new value of **n**. This value is then compared with zero. If it is greater than zero, the loop continues; otherwise it terminates.The **do-while** loop is especially useful when you process a menu selection, because you will usually want the body of a menu loop to execute at least once. Consider the following program which implements a very simple help system for Java's selection and iteration statements:

for

You were introduced to a simple form of the **for** loop in Chapter 2. As you will see, it is a powerful and versatile construct. Here is the general form of the **for** statement:

```
for(initialization; condition; iteration) {
// body
}
```

If only one statement is being repeated, there is no need for the curly braces. The **for** loop operates as follows. When the loop first starts, the *initialization* portion of the loop is executed. Generally, this is an expression that sets the value of the *loop control variable,* which acts as a counter that controls the loop. It is important to understand that the initialization expression is only executed once. Next, *condition* is evaluated. This must be a Boolean expression. It usually tests the loop control variable against a target value. If

this expression is true, then the body of the loop is executed. If it is false, the loop terminates. Next, the *iteration* portion of the loop is executed. This is usually an expression that increments or decrements the loop control variable. The loop then iterates, first evaluating the conditional expression, then executing the body of the loop, and then executing the iteration expression with each pass. This process repeats until the controlling expression is false.
Here is a version of the "tick" program that uses a **for** loop:

```
// Demonstrate the for
loop. class ForTick {
public static void main(String args[])
{ int n;
for(n=10; n>0; n--)
System.out.println("tick " + n);
}
}
```

Declaring Loop Control Variables Inside the for Loop

Often the variable that controls a **for** loop is only needed for the purposes of the loop and is not used elsewhere. When this is the case, it is possible to declare the variable inside the initialization portion of the **for**. For example, here is the preceding program recoded so that the loop control variable **n** is declared as an **int** inside the **for**:

```
// Declare a loop control variable inside the
for. class ForTick {
public static void main(String args[]) {
// here, n is declared inside of the for
loop for(int n=10; n>0; n--)
System.out.println("tick " + n);
}
}
```

When you declare a variable inside a **for** loop, there is one important point to remember: the scope of that variable ends when the **for** statement does. (That is, the scope of the variable is limited to the **for** loop.) Outside the **for** loop, the variable will cease to exist. If you need to use the loop control variable elsewhere in your program, you will not be able to declare it inside the **for** loop. When the loop control variable will not be needed elsewhere, most Java programmers declare it inside the **for**. For example, here is a simple program that tests for prime numbers. Notice that the loop control variable, **i**, is declared inside the **for** since it is not needed elsewhere.

```
// Test for primes.
class FindPrime {
public static void main(String args[])
{ int num;
boolean isPrime =
true; num = 14;
for(int i=2; i <= num/2; i++)
{ if((num % i) == 0) { isPrime
= false;
break;
}
}
if(isPrime) System.out.println("Prime");
else System.out.println("Not Prime");
}
}
```

Using the Comma

There will be times when you will want to include more than one statement in the initialization and iteration portions of the **for** loop. For example, consider the loop in the following program:

```
class Sample {
public static void main(String args[]) {
int a, b;
b = 4;
for(a=1; a<b; a++) {
System.out.println("a = " + a);
System.out.println("b = " + b);
b--;
}
}
}
```

As you can see, the loop is controlled by the interaction of two variables. Since the loop is governed by two variables, it would be useful if both could be included in the **for** statement, itself, instead of **b** being handled manually. Fortunately, Java provides a way to accomplish this. To allow two or more variables to control a **for** loop, Java permits you to include multiple statements in both the initialization and iteration portions of the **for**. Each statement is separated from the next by a comma.

Using the comma, the preceding **for** loop can be more efficiently coded as shown here:

```
// Using the comma.
class Comma {
public static void main(String args[])
{ int a, b;
for(a=1, b=4; a<b; a++, b--) {
System.out.println("a = " + a);
System.out.println("b = " + b);
}
}
}
```

In this example, the initialization portion sets the values of both **a** and **b**. The two comma-separated statements in the iteration portion are executed each time the loop repeats. The program generates the following output:

```
a = 1
b = 4
```

```
a = 2
b = 3
```

Nested Loops

Like all other programming languages, Java allows loops to be nested. That is, one loop may be inside another. For example, here is a program that nests **for** loops:

```
// Loops may be
nested. class Nested {
public static void main(String args[])
{ int i, j;
for(i=0; i<10; i++) {
for(j=i; j<10; j++)
System.out.print(".");
System.out.println();
}
}
```

The output produced by this program is shown here:

```
..........
.........
........
.......
......
.....
..
```

LEARNING ACTIVITY 3

---

1. What are Features of Operators and it’s importance in java programming ?
2. Describe the basic Arithmetic Operators.
3. Explain Increment and Decrement Operators.
4. Give the list of Relational Operators.
5. Explain about the Assignment Operators.
6. Describe Control Statements.
7. Explain Iteration Statements.

TOPIC 3

3. ARRAYSAND ARRAYLISTS

LEARNING OUTCOMES

After studying this topic, you should be able to:

- Describe features of Data types, Variables and Arrays.
- Understand Primitive data types syntaxes and its importance in programming.
- Declare Variables
- Understand Dynamic Initialization.
- Declare Array.
- Work with Strings.

3.1. ARRAYS

An *array* is a group of like-typed variables that are referred to by a common name. Arrays of any type can be created and may have one or more dimensions. A specific element in an array is accessed by its index. Arrays offer a convenient means of grouping related information.

One-Dimensional Arrays

A *one-dimensional array* is, essentially, a list of like-typed variables. To create an array, you first must create an array variable of the desired type. The general form of a one-dimensional array declaration is:

```
type var-name[ ];
```

Here, *type* declares the base type of the array. The base type determines the data type of each element that comprises the array. Thus, the base type for the array determines what type of data the array will hold. For example, the following declares an array named **month_days** with the type "array of int":

```
int month_days[ ];
```

Although this declaration establishes the fact that **month_days** is an array variable, no array actually exists. In fact, the value of **month_days** is set to **null**, which represents an array with no value. To link **month_days** with an actual, physical array of integers, you must allocate one using **new** and assign it to **month_days**. **new** is a special operator that allocates memory.
You will look more closely at **new** in a later chapter, but you need to use it now to allocate memory for arrays. The general form of **new** as it applies to one-dimensional arrays appears as follows:

```
array-var = new type[size];
```

Here, *type* specifies the type of data being allocated, *size* specifies the number of elements in the array, and *array-var* is the array variable that is linked to the array. That is, to use **new** to allocate an array, you must specify the type and number of elements to allocate. The elements in the array allocated by **new** will automatically be initialized to zero.This example allocates a 12-element array of integers and links them to **month_days**.

```
month_days = new int[12];
```

After this statement executes, **month_days** will refer to an array of 12 integers. Further, all elements in the array will be initialized to zero.Let’s review: Obtaining an array is a two-step process. First, you must declare a variable of the desired array type. Second, you must allocate the memory that will hold the array, using **new**, and assign it to the array variable. Thus, in Java all arrays are dynamically allocated. If the concept of dynamic allocation is unfamiliar to you, don’t worry. It will be described at length later in this book. Once you have allocated an array, you can access a specific element in the array by specifying its index within square brackets. All array indexes start at zero. For example, this statement assigns the value 28 to the second element of **month_days**.

```
month_days[1] = 28;
```

The next line displays the value stored at index 3.

```
System.out.println(month_days[3]);
```

Putting together all the pieces, here is a program that creates an array of the number of days in each month.

```
// Demonstrate a one-dimensional
array. class Array {
public static void main(String args[])
{ int month_days[];
month_days = new
int[12]; month_days[0] =
31; month_days[1] = 28;
month_days[2] = 31;
month_days[3] = 30;
month_days[4] = 31;
month_days[5] = 30;
month_days[6] = 31;
month_days[7] = 31;
month_days[8] = 30;
month_days[9] = 31;
month_days[10] = 30;
month_days[11] = 31;
System.out.println("April has " + month_days[3] + " days.");
}
}
```

When you run this program, it prints the number of days in April. As mentioned, Java array indexes start with zero, so the number of days in April is **month_days[3]** or 30. It is possible to combine the declaration of the array variable with the allocation of the array itself, as shown here:

```
int month_days[] = new int[12];
```

This is the way that you will normally see it done in professionally written Java programs. Arrays can be initialized when they are declared. The process is much the same asthat used to initialize the simple types. An *array initializer* is a list of comma-separated expressions surrounded by curly braces. The commas separate the values of the array elements. The array will automatically be created large enough to hold the number of elements you specify in the array initializer. There is no need to use **new**. For example, to store the number of days in each month, the following code creates an initialized array of integers:

```
// An improved version of the previous
program. class AutoArray {
public static void main(String args[]) {
int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30,
31, 30, 31 };
System.out.println("April has " + month_days[3] + " days.");
}
}
```

When you run this program, you see the same output as that generated by the previous version. Java strictly checks to make sure you do not accidentally try to store or reference values outside of the range of the array. The Java run-time system will check to be sure that all array indexes are in the correct range. (In this regard, Java is fundamentally different from C/C++, which provide no run-time boundary checks.) For example, the run-time system will check the value of each index into **month_days** to make sure that it is between 0 and 11 inclusive. If you try to access elements outside the range of the array (negative numbers or numbers greater than the length of the array), you will cause a run-time error. Here is one more example that uses a one-dimensional array. It finds the average of a set of numbers.

```
// Average an array of
values. class Average {
public static void main(String args[]) {
double nums[] = {10.1, 11.2, 12.3, 13.4,
14.5}; double result = 0;
int i;
for(i=0; i<5; i++)
result = result + nums[i];
System.out.println("Average is " + result / 5);
}
}
```

Multidimensional Arrays

In Java, *multidimensional arrays* are actually arrays of arrays. These, as you might expect, look and act like regular multidimensional arrays. However, as you will see, there are a couple of subtle differences. To declare a multidimensional array variable, specify each additional index using another set of square brackets. For example, the following declares a two-dimensional array variable called **twoD**.

```
int twoD[ ][ ] = new int[4][5];
```

This allocates a 4 by 5 array and assigns it to **twoD**. Internally this matrix is implemented as an *array* of *arrays* of **int**. Conceptually, this array will look like the one shown in Figure 3-1.

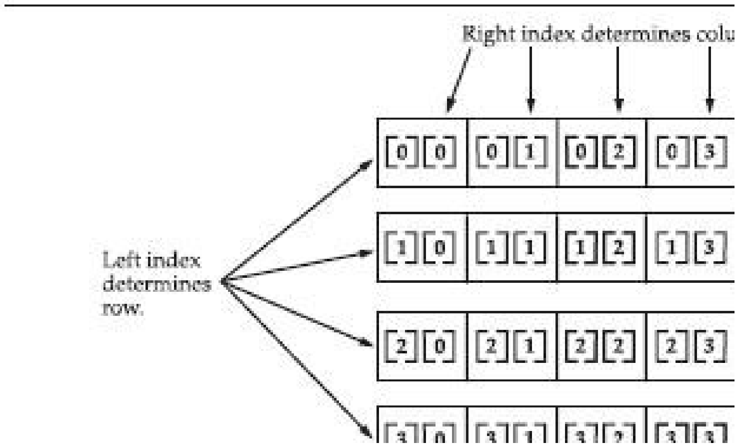


Figure 2.1 A conceptual view of a 4 by 5, two-dimensional array

The following program numbers each element in the array from left to right, top to bottom, and then displays these values:

```
// Demonstrate a two-dimensional
array. class TwoDArray {
public static void main(String args[ ])
{ int twoD[ ][ ]= new int[4][5];
int i, j, k = 0;
for(i=0; i<4; i++)
for(j=0; j<5; j++) {
twoD[i][j] = k;
k++;
}
for(i=0; i<4; i++) {
for(j=0; j<5; j++)
System.out.print(twoD[i][j] + " ");
System.out.println();
}
}
}
```

This program generates the following output:

```
01234
56789
10 11 12 13 14
15 16 17 18 19
```

3.2 A FEW WORDS ABOUT STRINGS

As you may have noticed, in the preceding discussion of data types and arrays there has been no mention of strings or a string data type. This is not because Java does not support such a type—it does. It is just that Java's string type, called **String**, is not a simple type. Nor is it simply an array of characters (as are strings in C/C++). Rather, **String** defines an object, and a full description of it requires an understanding of several object-related features. As such, it will be covered later in this book, after objects are described. However, so that you can use simple strings in example programs, the following brief introduction is in order. The **String** type is used to declare string variables. You can also declare arrays of strings. A quoted string constant can be assigned to a **String** variable. A variable of type **String** can be assigned to another variable of type **String**. You can use an object of type **String** as an argument to **println( )**. For example, consider the following fragment:

```
String str = "this is a test";
System.out.println(str);
```

Here, **str** is an object of type **String**. It is assigned the string "this is a test". This string is displayed by the **println( )** statement. As you will see later, **String** objects have many special features and attributes that make them quite powerful and easy to use. However, for the next few chapters, you will be using them only in their simplest form.

3.3 ARRAYLISTS

Java **ArrayList** class uses a *dynamic* array for storing the elements. It is like an array, but there is *no size limit*. We can add or remove elements anytime. So, it is much more flexible than the traditional array. It is found in the *java.util* package

Arraylist class implements List interface and it is based on an Array data structure. It is widely used because of the functionality and flexibility it offers.

ArrayList in Java, is a resizable-array implementation of the List interface. It implements all optional list operations and permits all elements, including null. Most of the developers choose Arraylist over Array as it is a very good alternative of traditional java arrays.

ARRAYS VS ARRAYLISTS IN JAVA

The main difference between array and arraylist is that arraylist can grow and shrink dynamically while an array cannot.

An array has a fixed length so if it is full you cannot add any more elements to it. Similarly, if number of elements are removed from ArrayList, the memory consumption remains same as it doesn’t shrink.

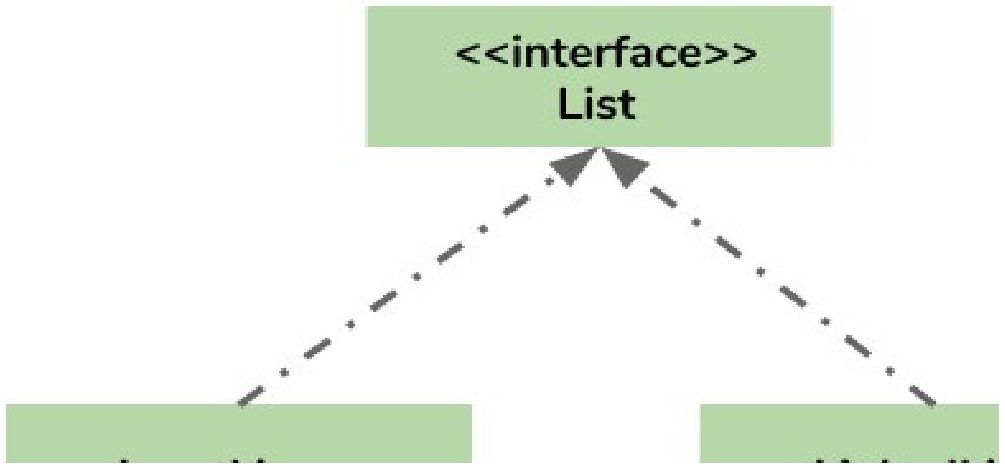
On the other hand, ArrayList can dynamically grow and shrink after addition and removal of elements. ArrayList class has several useful methods that can make our task easy.

ARRAYLISTS IN JAVA

- ArrayList can grow and shrink automatically based on the addition and removal of elements.
- ArrayList can contain duplicate elements
- ArrayList maintains the insertion order, which means the elements appear in the same order in which they are inserted.
- ArrayList is non synchronized. However you can make it synchronized.

HIERARCHY OF ARRAYLIST CLASS IN JAVA

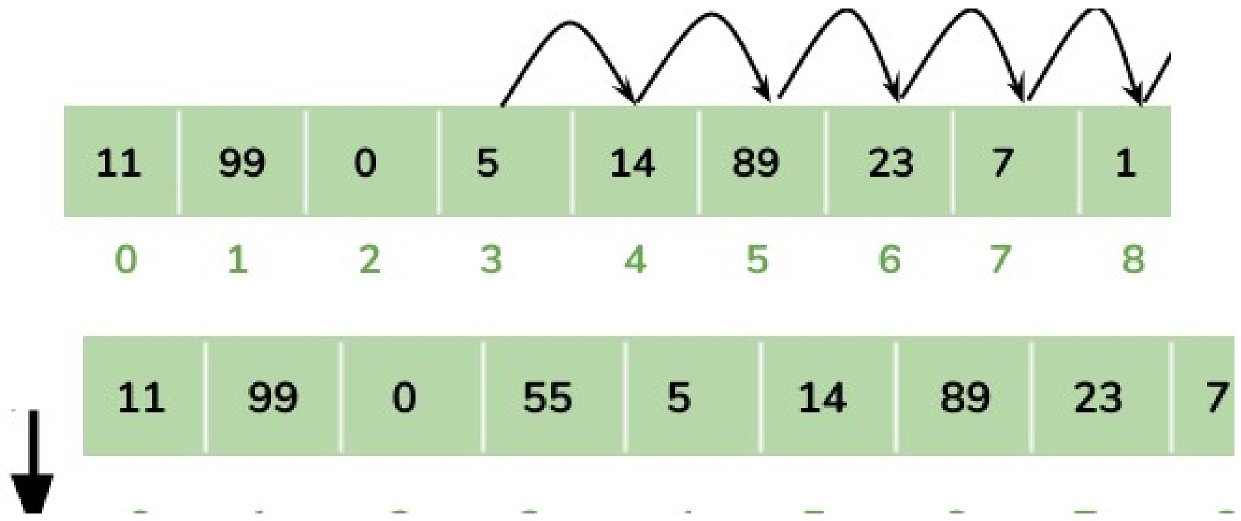
ArrayList class implements List interface and List interface extends Collection interface.



Arraylist in Java declaration
This is how you can declare an ArrayList of String type:
`ArrayList<String> list=new ArrayList<>();`
This is how you can declare an ArrayList of Integer type:
`ArrayList<Integer> list=new ArrayList<>();`

Adding elements to Arraylist in java

Adding Element in ArrayList at specified position:



You can add elements to an ArrayList by using add() method. This method has couple of variations, which you can use based on the requirement.

For example: If you want to add the element at the end of the List then you can simply call the add() method like this:

```
arrList.add("Chris"); //This will add "Chris" at the end of
List
```

To add the element at the specified location in ArrayList, you can specify the index in the add() method like this:

```
arrList.add(3, " Chris"); //This will add "Chris" at the
fourth position
```

Let's write the complete code:

```
import java.util.*;
class JavaExample{
    public static void main(String args[]){
      ArrayList<String> arrList=new ArrayList<String>();
      arrList.add("Chris");
      arrList.add("Tim");
      arrList.add("Lucy");
      arrList.add("Percy");
      arrList.add("Angela");
      arrList.add("Tom");

      //displaying elements
      System.out.println(arrList);
      //Adding " Chris" at the fourth position
      arrList.add(3, " Chris");

      //displaying elements
      System.out.println(arrList);
   }
}
```

```
Output:

[Chris, Tim, Lucy, Percy, Angela, Tom]
[Chris, Tim, Lucy, Chris, Percy, Angela, Tom]
```

Note: Since the index starts with 0, index 3 would represent fourth position not 3.

Change an element in ArrayList
You can use the set method to change an element in ArrayList. You need to provide the index and new element, this method then updates the element present at the given index with the new given element.

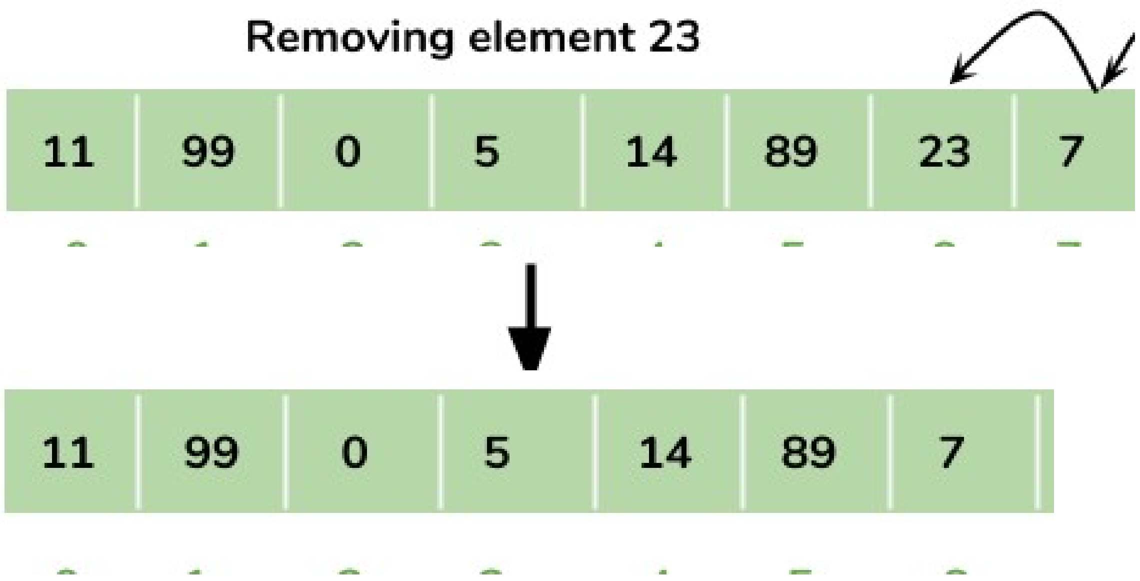
In the following example, we have given the index as 0 and new element as “Lucy” in the set() method. The method updated the element present at the index 0 (“Jim”) with the new String element “Lucy”.

```
import java.util.ArrayList;
public class JavaExample {
   public static void main(String[] args) {
      ArrayList<String> names = new ArrayList<String>();
      names.add("Jim");
      names.add("Jack");
      names.add("Mikhail");
      names.add("Chantelle");
      names.set(0, "Lucy");
      System.out.println(names);
   }
}
```

Output

```
[Lucy, Jack, Mikhail, Chantelle]
```

How to remove element from Arraylist in Java?
Removing Element from ArrayList:



You can use remove() method to remove elements from an ArrayList. Similar to add() method, this method also has couple of variations.

```
import java.util.*;
class JavaExample{
   public static void main(String args[]){
      ArrayList<String> alist=new ArrayList<String>();
      alist.add("Chris");
      alist.add("Tim");
      alist.add("Lucy");
      alist.add("Percy");
      alist.add("Angela");
      alist.add("Tom");

      //displaying elements
      System.out.println(alist);

      //Removing "Chris" and "Angela"
      alist.remove("Chris");
      alist.remove("Angela");

      //displaying elements
      System.out.println(alist);

      //Removing 3rd element
      alist.remove(2);

      //displaying elements
      System.out.println(alist);
   }
}
```

Output:

```
[Chris, Tim, Lucy, Percy, Angela, Tom]
```

```
[Tim, Lucy, Percy, Tom]
[Tim, Lucy, Tom]
```

ITERATING ARRAYLIST

Here, we are using enhanced for loop to iterate ArrayList elements. This one of the best ways to iterate an ArrayList of string type.

Example 1

```
import java.util.*;
class JavaExample{
  public static void main(String args[]){
     ArrayList<String> alist=new ArrayList<String>();
     alist.add("Gregor Clegane");
     alist.add("Khal Drogo");
     alist.add("Cersei Lannister");
     alist.add("Sandor Clegane");
     alist.add("Tyrion Lannister");

     //iterating ArrayList
     for(String str:alist)
        System.out.println(str);
     }
}
Output:

Gregor Clegane
Khal Drogo
Cersei Lannister
Sandor Clegane
Tyrion Lannister
```

Example 2

The following example shows how to iterate over an ArrayList using
forEach and lambda expression.
iterator().
iterator() and forEachRemaining() method.
listIterator().
Simple for-each loop.
for loop with index.

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.ListIterator;

public class IterateOverArrayListExample {
    public static void main(String[] args) {
       List<String> tvShows = new ArrayList<>();
        tvShows.add("Breaking Bad");
        tvShows.add("Game Of Thrones");
        tvShows.add("Friends");
        tvShows.add("Prison break");

System.out.println("=== Iterate using forEach and lambda ===");
```

```
        tvShows.forEach(tvShow -> {
            System.out.println(tvShow);
        });

System.out.println("\n=== Iterate using an iterator()     ===");
        Iterator<String> tvShowIterator = tvShows.iterator();
        while (tvShowIterator.hasNext()) {
            String tvShow = tvShowIterator.next();
            System.out.println(tvShow);
        }
 System.out.println("\n=== Iterate using an iterator() and
forEachRemaining() method ===");
        tvShowIterator = tvShows.iterator();
        tvShowIterator.forEachRemaining(tvShow -> {
            System.out.println(tvShow);
        });
System.out.println("\n=== Iterate using a listIterator() to traverse
in both directions ===");
// Here, we start from the end of the list and traverse backwards.
    ListIterator<String> tvShowListIterator =
tvShows.listIterator(tvShows.size());
        while (tvShowListIterator.hasPrevious()) {
            String tvShow = tvShowListIterator.previous();
            System.out.println(tvShow);
        }
System.out.println("\n=== Iterate using simple for-each loop ===");
        for(String tvShow: tvShows) {
            System.out.println(tvShow);
        }

System.out.println("\n=== Iterate using for loop with index ===");
        for(int i = 0; i < tvShows.size(); i++) {
            System.out.println(tvShows.get(i));
        }
    }
}
```

```
# Output
=== Iterate using forEach and lambda ===
Breaking Bad
Game Of Thrones
Friends
Prison break

=== Iterate using an iterator() ===
Breaking Bad
Game Of Thrones
Friends
Prison break

=== Iterate using an iterator() and Java 8 forEachRemaining() method
===
Breaking Bad
Game Of Thrones
Friends
```

```
Prison break

=== Iterate using a listIterator() to traverse in both directions ===
Prison break
Friends
Game Of Thrones
Breaking Bad

=== Iterate using simple for-each loop ===
Breaking Bad
Game Of Thrones
Friends
Prison break

=== Iterate using for loop with index ===
Breaking Bad
Game Of Thrones
Friends
Prison break
```

The iterator() and listIterator() methods are useful when you need to modify the ArrayList while traversing.

Consider the following example, where we remove elements from the ArrayList using iterator.remove() method while traversing through it −

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class ArrayListIteratorRemoveExample {
    public static void main(String[] args) {
        List<Integer> numbers = new ArrayList<>();
        numbers.add(13);
        numbers.add(18);
        numbers.add(25);
        numbers.add(40);

        Iterator<Integer> numbersIterator = numbers.iterator();
        while (numbersIterator.hasNext()) {
            Integer num = numbersIterator.next();
            if(num % 2 != 0) {
                numbersIterator.remove();
            }
        }

        System.out.println(numbers);
    }
}
# Output
[18, 40]
```

Searching for elements in an ArrayList
The example below shows how to:

Check if an ArrayList contains a given element | contains()

Find the index of the first occurrence of an element in an ArrayList | indexOf()

Find the index of the last occurrence of an element in an ArrayList | lastIndexOf()

```
import java.util.ArrayList;
import java.util.List;

public class SearchElementsInArrayListExample {
    public static void main(String[] args) {
        List<String> names = new ArrayList<>();
        names.add("John");
        names.add("Alice");
        names.add("Bob");
        names.add("Chris");
        names.add("John");
        names.add("Chris");
        names.add("Maria");

      // Check if an ArrayList contains a given element
System.out.println("Does names array contain \"Bob\"? : " +
names.contains("Bob"));

// Find the index of the first occurrence of an element in an
ArrayList
System.out.println("indexOf \"Chris\": " + names.indexOf("Chris"));
System.out.println("indexOf \"Mark\": "+names.indexOf("Mark"));

// Find the index of the last occurrence of an element in an ArrayList
System.out.println("lastIndexOf \"John\" : " +
names.lastIndexOf("John"));
System.out.println("lastIndexOf \"Bill\" : " +
names.lastIndexOf("Bill"));
    }
}
```

```
# Output
Does names array contain "Bob"? : true
indexOf "Chris": 3
indexOf "Mark": -1
lastIndexOf "John" : 4
lastIndexOf "Bill" : -1
```

LEARNER ACTIVITY 3

Question 1

a) Write a statement that declares a string array initialized with the following strings: "Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday" and "Saturday".
b) Write a loop that displays the contents of each element in the array that you declared.

Question 2

Write a program that creates an array of 10 elements size. Your program should prompt the user to input numbers in array and then display the sum of all array elements.

Question 3

The variable list1 and list2 are reference arrays that each have 5 elements. Write code that copies the values in list1 to list2. Values in list1 are input by user.

Question 4

Write a Java program to reverse the element of an integer 1-D array.

Question 5

Write a Java program to find the largest and smallest element of an array.

Question 6

Write a menu driven Java program with following option

a. Accept elements of an array
b. Display elements of an array
c. Search the element within array given by user
d. Sort the array using bubble sort method

Write methods for all options. The methods should be static and have one parameter name of the array.

Question 7

Suppose A, B, C are arrays of integers. The numbers in array A appear in ascending order while the numbers in array B appear in descending order. Write a user defined method in Java to produce third array C by merging arrays A and B in ascending order. Use A, B and C as arguments in the method.

Question 8

Write a menu driven program to do following operation on two dimensional array A of size m x n. You should use user-defined methods which accept 2-D array A, and its size m and n as arguments. The options are:

- To input elements into matrix of size m x n
- To display elements of matrix of size m x n
- Sum of all elements of matrix of size m x n
- To display row-wise sum of matrix of size m x n
- To display column-wise sum of matrix of size m x n
- To create transpose of matrix of size n x m

Question 9

Write code that creates an ArrayList that can hold string objects. Add the names of three cars to the ArrayList, and then display the contents of the ArrayList.

TOPIC 4

4. INTRODUCING CLASSES AND METHODS

LEARNING OUTCOMES

After studying this topic, you should be able to:

- Define A Class and it's form in declaration.
  - Declaring Objects.
  - Introducing Methods.
  - Constructors.
  - The This Keyword.
- Garbage Collection.
- Understand the static.

4.1. INTRODUCING CLASSES

The class is at the core of Java. It is the logical construct upon which the entire Java language is built because it defines the shape and nature of an object. As such, the class forms the basis for object-oriented programming in Java. Any concept you wish to implement in a Java program must be encapsulated within a class. Because the class is so fundamental to Java, this and the next few chapters will be devoted to it. Here, you will be introduced to the basic elements of a class and learn how a class can be used to create objects. You will also learn about methods, constructors, and the **this** keyword.

Class Fundamentals

Classes have been used since the beginning of this book. However, until now, only the most rudimentary form of a class has been used. The classes created in the preceding chapters primarily exist simply to encapsulate the **main( )** method, which has been used to demonstrate the basics of the Java syntax. As you will see, classes are substantially more powerful than the limited ones presented so far. Perhaps the most important thing to understand about a class is that it defines a new data type. Once defined, this new type can be used to create objects of that type. Thus, a class is a *template* for an object, and an object is an *instance* of a class. Because an object is an instance of a class, you will often see the two words *object* and *instance* used interchangeably.

The General Form of a Class

When you define a class, you declare its exact form and nature. You do this by specifying the data that it contains and the code that operates on that data. While very simple classes may contain only code or only data, most real-world classes contain both. As you will

see, a class' code defines the interface to its data. A class is declared by use of the **class** keyword. The classes that have been used up to this point are actually very limited examples of its complete form. Classes can (and usually do) get much more complex. The general form of a **class** definition is shown here:

```
class classname {
type instance-variable1;
type instance-variable2;
// ...
type instance-variableN;
type methodname1(parameter-list) {
// body of method
}
type methodname2(parameter-list) {
// body of method

}
// ...
type methodnameN(parameter-list) {
// body of method
}
}
```

The data, or variables, defined within a **class** are called *instance variables.* The code is contained within *methods.* Collectively, the methods and variables defined within a class are called *members* of the class. In most classes, the instance variables are acted upon and accessed by the methods defined for that class. Thus, it is the methods that determine how a class' data can be used.Variables defined within a class are called instance variables because each instance of the class (that is, each object of the class) contains its own copy of these variables.Thus, the data for one object is separate and unique from the data for another. We will come back to this point shortly, but it is an important concept to learn early. All methods have the same general form as **main( )**, which we have been using thusfar. However, most methods will not be specified as **static** or **public**. Notice that the general form of a class does not specify a **main( )** method. Java classes do not need to have a **main( )** method. You only specify one if that class is the starting point for your program. Further, applets don't require a **main( )** method at all.

Note: *C++ programmers will notice that the class declaration and the implementation of the methods are stored in the same place and not defined separately. This sometimes makesfor very large* ***.java*** *files, since any class must be entirely defined in a single source file.This design feature was built into Java because it was felt that in the long run, having specification, declaration, and implementation all in one place makes for code that is easier to maintain.*

A Simple Class

Let's begin our study of the class with a simple example. Here is a class called **Box** that defines three instance variables: **width**, **height**, and **depth**. Currently, **Box** does not contain any methods (but some will be added soon).

```
class Box {
double width;
double height;
double depth;
}
```

As stated, a class defines a new type of data. In this case, the new data type is called **Box**. You will use this name to declare objects of type **Box**. It is important to rememberthat a **class** declaration only creates a template; it does not create an actual object. Thus,the preceding code does not cause any objects of type **Box** to come into existence.To actually create a **Box** object, you will use a statement like the following: Box mybox = new Box();
// create a Box object called mybox. After this statement executes, **mybox** will be an instance of **Box**. Thus, it will have “physical” reality. For the moment, don’t worry about the details of this statement. Again, each time you create an instance of a class, you are creating an object that contains its own copy of each instance variable defined by the class. Thus, every **Box** object will contain its own copies of the instance variables **width**, **height**, and **depth**. To access these variables, you will use the *dot* (.) operator. The dot operator links the name of the object with the name of an instance variable. For example, to assign the **width**variable of **mybox** the value 100, you would use the following statement:

```
mybox.width = 100;
```

This statement tells the compiler to assign the copy of **width** that is contained within the **mybox** object the value of 100. In general, you use the dot operator to access both the instance variables and the methods within an object.
Here is a complete program that uses the **Box** class:

```
/* A program that uses the Box class.
Call this file BoxDemo.java
*/
class Box {
double width;
double height;
double depth;
}

// This class declares an object of type
Box. class BoxDemo {
public static void main(String args[])
{ Box mybox = new Box();
double vol;
// assign values to mybox's instance
variables mybox.width = 10;
mybox.height  =  20;
mybox.depth = 15;
// compute volume of box
vol = mybox.width * mybox.height * mybox.depth;
System.out.println("Volume is " + vol);
}
}
```

You should call the file that contains this program **BoxDemo.java**, because the **main( )** method is in the class called **BoxDemo**, not the class called **Box**. When you compile this program, you will find that two **.class** files have been created, one for **Box** and one for **BoxDemo**. The Java compiler automatically puts each class into its own **.class** file. It is not necessary for both the **Box** and the **BoxDemo** class to actually be in the same source file. You could put each class in its own file, called **Box.java** and **BoxDemo.java**, respectively.To run this program, you must execute **BoxDemo.class**. When you do, you will see the following output:

```
Volume is 3000.0
```

As stated earlier, each object has its own copies of the instance variables. This means that if you have two **Box** objects, each has its own copy of **depth**, **width**, and **height**. It is important to understand that changes to the instance variables of one object have no effect on the instance variables of another. For example, the following program declares two **Box** objects:

```
// This program declares two Box
objects. class Box {
double  width;
double height;
double depth;
}
class BoxDemo2 {
public static void main(String args[])
{ Box mybox1 = new Box();
Box mybox2 = new
Box(); double vol;
// assign values to mybox1's instance
variables mybox1.width = 10;
mybox1.height  =  20;
mybox1.depth = 15;
/* assign different values to
mybox2's instance variables */
mybox2.width = 3; mybox2.height =
6; mybox2.depth = 9;

// compute volume of first box
vol = mybox1.width * mybox1.height *
mybox1.depth; System.out.println("Volume is " + vol);
// compute volume of second box
vol = mybox2.width * mybox2.height *
mybox2.depth; System.out.println("Volume is " + vol);
}
}
```

The output produced by this program is shown here:

```
Volume is 3000.0
Volume is 162.0
```

As you can see, **mybox1**'s data is completely separate from the data contained in **mybox2**.

4.2. DECLARING OBJECTS

As just explained, when you create a class, you are creating a new data type. You can use this type to declare objects of that type. However, obtaining objects of a class is a two-step process. First, you must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can *refer* to an object. Second, you must acquire an actual, physical copy of the object and assign it to that variable. You can do this using the **new** operator. The **new** operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it. This reference is, more or less, the address in memory of the object allocated by **new**. This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated. Let's look at the details of this procedure. In the preceding sample programs, a line similar to the following is used to declare an object of type **Box**:

```
Box mybox = new Box();
```

This statement combines the two steps just described. It can be rewritten like this
to show each step more clearly:

```
Box mybox; // declare reference to object
mybox = new Box(); // allocate a Box object
```

The first line declares **mybox** as a reference to an object of type **Box**. After this line executes, **mybox** contains the value **null**, which indicates that it does not yet point to an actual object. Any attempt to use **mybox** at this point will result in a compile-time error. The next line allocates an actual object and assigns a reference to it to **mybox**. After the second line executes, you can use **mybox** as if it were a **Box** object. But in reality, **mybox** simply holds the memory address of the actual **Box** object. The effect of these two lines of code.

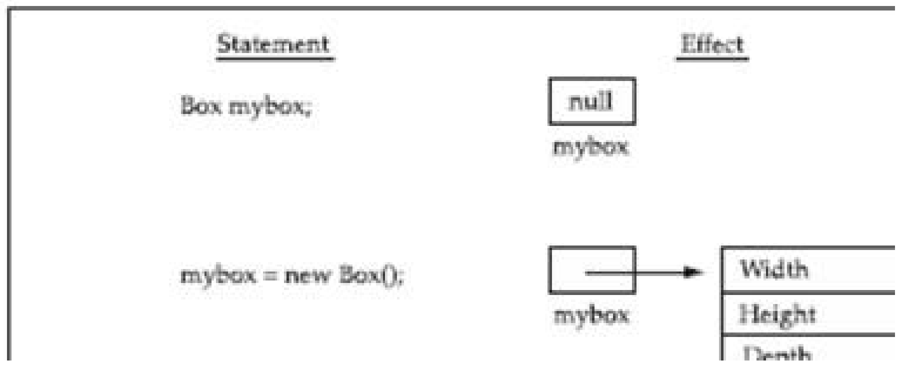


Figure 4.1 Declaring an Object of type Box

A Closer Look at keyword new

As just explained, the **new** operator dynamically allocates memory for an object. It has this general form:

```
        class-var = new classname( );
```

Here, *class-var* is a variable of the class type being created. The *classname* is the name of the class that is being instantiated. The class name followed by parentheses specifies the *constructor* for the class. A constructor defines what occurs when an object of a class is created. Constructors are an important part of all classes and have many significant attributes. Most real-world classes explicitly define their own constructors within their class definition. However, if no explicit constructor is specified, then Java will automatically supply a default constructor. This is the case with **Box**. For now, we will use the default constructor. Soon, you will see how to define your own constructors. At this point, you might be wondering why you do not need to use **new** for such things as integers or characters. The answer is that Java's simple types are not implemented as objects. Rather, they are implemented as "normal" variables. This is done in the interest of efficiency. As you will see, objects have many features and attributes that require Java to treat them differently than it treats the simple types. By not applying the same overhead to the simple types that applies to objects, Java can implement the simple types more efficiently. Later, you will see object versions of the simple types that are available for your use in those situations in which complete objects of these types are needed. It is important to understand that **new** allocates memory for an object during run time. The advantage of this approach is that your program can create as many or as few objects as it needs during the execution of your program. However, since memory is finite, it is possible that **new** will not be able to allocate memory for an object because insufficient memory exists. If this happens, a run-time exception will occur. (You will learn how to handle this and other exceptions in Chapter 10.) For the sample programs in this book, you won't need to worry about running out of memory, but you will need to consider this possibility in real-world programs that you write. Let's once again review the distinction between a class and an object. A class creates a new data type that can be used to create objects. That is, a class creates a logical framework that defines the relationship between its members. When you declare an object of a class, you are creating an instance of that class. Thus, a class is a logical construct. An object has physical reality. (That is, an object occupies space in memory.) It is important to keep this distinction clearly in mind.

Assigning Object Reference Variables

Object reference variables act differently than you might expect when an assignment takes place. For example, what do you think the following fragment does?

```
        Box b1 = new Box();
        Box b2 = b1;
```

You might think that **b2** is being assigned a reference to a copy of the object referred to by **b1**. That is, you might think that **b1** and **b2** refer to separate and distinct objects. However, this would be wrong. Instead, after this fragment executes, **b1** and **b2** will both refer to the *same* object. The assignment of **b1** to **b2** did not allocate any memory or copy any part of the original object. It simply makes **b2** refer to the same object as does **b1**. Thus, any changes made to the object through **b2** will affect the object to which **b1** is referring, since they are the same object. This situation is depicted here:

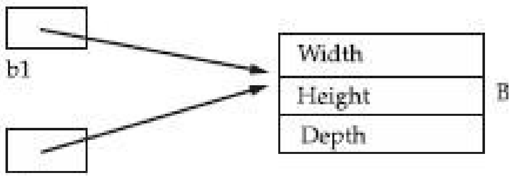


Figure 4.2

Although **b1** and **b2** both refer to the same object, they are not linked in any other way. For example, a subsequent assignment to **b1** will simply *unhook* **b1** from the original object without affecting the object or affecting **b2**. For example:

```
Box b1 = new Box();
Box b2 = b1;
// ...
b1 = null;
```

Here, **b1** has been set to **null**, but **b2** still points to the original object.

Note: *When you assign one object reference variable to another object reference variable, you are not creating a copy of the object, you are only making a copy of the reference.*

4.3 INTRODUCING METHODS

As mentioned at the beginning of this chapter, classes usually consist of two things: instance variables and methods. The topic of methods is a large one because Java gives them so much power and flexibility. In fact, much of the next chapter is devoted to methods. However, there are some fundamentals that you need to learn now so that you can begin to add methods to your classes.

This is the general form of a method:

```
type name(parameter-list) {
// body of method
}
```

Here, *type* specifies the type of data returned by the method. This can be any valid type, including class types that you create. If the method does not return a value, its return type must be **void**. The name of the method is specified by *name.* This can be any legal identifier other than those already used by other items within the current scope. The *parameter-list* is a sequence of type and identifier pairs separated by commas. Parameters are essentially variables that receive the value of the *arguments* passed to the method when it is called. If the method has no parameters, then the parameter list will be empty. Methods that have a return type other than **void** return a value to the calling routine using the following form of the **return** statement:

```
return value;
```

Here, *value* is the value returned.
In the next few sections, you will see how to create various types of methods, including those that take parameters and those that return values.

Adding a Method to the Box Class

Although it is perfectly fine to create a class that contains only data, it rarely happens. Most of the time you will use methods to access the instance variables defined by the class. In fact, methods define the interface to most classes. This allows the class implementor to hide the specific layout of internal data structures behind cleaner method abstractions. In addition to defining methods that provide access to data, you can also define methods that are used internally by the class itself. Let's begin by adding a method to the **Box** class. It may have occurred to you while looking at the preceding programs that the computation of a box's volume was something that was best handled by the **Box** class rather than the **BoxDemo** class. After all, since the volume of a box is dependent upon the size of the box, it makes sense to have the **Box** class compute it. To do this, you must add a method to **Box**, as shown here:

```
// This program includes a method inside the box
class. class Box {
double  width;
double height;
double depth;
// display volume of a box
void volume() {
System.out.print("Volume is ");
System.out.println(width * height * depth);
}
}
class BoxDemo3 {
public static void main(String args[]) {
Box mybox1 = new Box();
Box mybox2 = new Box();
// assign values to mybox1's instance
variables mybox1.width = 10;
mybox1.height  =  20;
mybox1.depth = 15;
/* assign different values to
mybox2's instance variables */
mybox2.width = 3; mybox2.height =
6; mybox2.depth = 9;

// display volume of first
box mybox1.volume();
// display volume of second
box mybox2.volume();
}
}
```

This program generates the following output, which is the same as the previous version.

```
Volume is 3000.0
Volume is 162.0
```

Look closely at the following two lines of code:

```
mybox1.volume();
mybox2.volume();
```

The first line here invokes the **volume( )** method on **mybox1**. That is, it calls **volume( )** relative to the **mybox1** object, using the object's name followed by the dot operator. Thus, the call to **mybox1.volume( )** displays the volume of the box defined by **mybox1**, and the call to **mybox2.volume( )** displays the volume of the box defined by **mybox2**. Each time **volume( )** is invoked, it displays the volume for the specified box. If you are unfamiliar with the concept of calling a method, the following discussion will help clear things up. When **mybox1.volume( )** is executed, the Java run-time system transfers control to the code defined inside **volume( )**. After the statements inside **volume( )** have executed, control is returned to the calling routine, and execution resumes with the line of code following the call. In the most general sense, a method is Java's way of implementing subroutines. There is something very important to notice inside the **volume( )** method: the instance variables **width**, **height**, and **depth** are referred to directly, without preceding them with an object name or the dot operator. When a method uses an instance variable that is defined by its class, it does so directly, without explicit reference to an object and without use of the dot operator. This is easy to understand if you think about it. A method is always invoked relative to some object of its class. Once this invocation has occurred, the object is known. Thus, within a method, there is no need to specify the object a second time. This means that **width**, **height**, and **depth** inside **volume( )** implicitly refer to the copies of those variables found in the object that invokes **volume( )**. Let's review: When an instance variable is accessed by code that is not part of the class in which that instance variable is defined, it must be done through an object, by use of the dot operator. However, when an instance variable is accessed by code that is part of the same class as the instance variable, that variable can be referred to directly. The same thing applies to methods.

Returning a Value

While the implementation of **volume( )** does move the computation of a box's volume inside the **Box** class where it belongs, it is not the best way to do it. For example, what if another part of your program wanted to know the volume of a box, but not display its value? A better way to implement **volume( )** is to have it compute the volume of the box and return the result to the caller. The following example, an improved version of the preceding program, does just that:

```
// Now, volume() returns the volume of a
box. class Box {
double  width;
double height;
double depth;
// compute and return volume
double volume() {
return width * height * depth;
}
}

class BoxDemo4 {
public static void main(String args[ ]) {
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// assign values to mybox1's instance
variables mybox1.width = 10;
mybox1.height  =  20;
mybox1.depth = 15;
/* assign different values to
mybox2's instance variables */
mybox2.width = 3; mybox2.height =
6; mybox2.depth = 9;

// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box vol =
mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

As you can see, when **volume( )** is called, it is put on the right side of an assignment statement. On the left is a variable, in this case **vol**, that will receive the value returned by **volume( )**. Thus, after

```
vol = mybox1.volume();
```

executes, the value of **mybox1.volume( )** is 3,000 and this value then is stored in **vol**. There are two important things to understand about returning values:

- The type of data returned by a method must be compatible with the return type specified by the method. For example, if the return type of some method is **boolean**, you could not return an integer.
- The variable receiving the value returned by a method (such as **vol**, in this case) must also be compatible with the return type specified for the method.

One more point: The preceding program can be written a bit more efficiently because there is actually no need for the **vol** variable. The call to **volume( )** could have been used in the **println( )** statement directly, as shown here:

```
System.out.println("Volume is " + mybox1.volume());
```

In this case, when **println( )** is executed, **mybox1.volume( )** will be called automatically and its value will be passed to **println( )**.

Adding a Method That Takes Parameters

While some methods don't need parameters, most do. Parameters allow a method to be generalized. That is, a parameterized method can operate on a variety of data and/or be used in a number of slightly different situations. To illustrate this point, let's use a very simple example. Here is a method that returns the square of the number 10:

```
int square()
{
return 10 * 10;
}
```

While this method does, indeed, return the value of 10 squared, its use is very limited. However, if you modify the method so that it takes a parameter, as shown next, then you can make **square( )** much more useful.

```
int square(int i)
{
return i * i;
}
```

Now, **square( )** will return the square of whatever value it is called with. That is, **square( )** is now a general-purpose method that can compute the square of any integer value, rather than just 10.
Here is an example:

```
int x, y;
x = square(5); // x equals 25
x = square(9); // x equals 81
y = 2;
x = square(y); // x equals 4
```

In the first call to **square( )**, the value 5 will be passed into parameter **i**. In the second call, **i** will receive the value 9. The third invocation passes the value of **y**, which is 2 in this example. As these examples show, **square( )** is able to return the square of whatever data it is passed. It is important to keep the two terms *parameter* and *argument* straight. A *parameter* is a variable defined by a method that receives a value when the method is called. For example, in **square( )**, **i** is a parameter. An *argument* is a value that is passed to a method when it is invoked. For example, **square(100)** passes 100 as an argument.

Inside **square( )**, the parameter **i** receives that value. You can use a parameterized method to improve the **Box** class. In the preceding examples, the dimensions of each box had to be set separately by use of a sequence of statements, such as:

```
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
```

While this code works, it is troubling for two reasons. First, it is clumsy and error prone. For example, it would be easy to forget to set a dimension. Second, in well-designed Java programs, instance variables should be accessed only through methods defined by their class. In the future, you can change the behavior of a method, but you can't change the behavior of an exposed instance variable. Thus, a better approach to setting the dimensions of a box is to create a method that takes the dimension of a box in its parameters and sets each instance variable appropriately. This concept is implemented by the following program:

```
// This program uses a parameterized
method. class Box {
double  width;
double height;
double depth;
// compute and return volume
double volume() {
return width * height * depth;
}
// sets dimensions of box
void setDim(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
}

class BoxDemo5 {
public static void main(String args[ ]) {
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// initialize each box
mybox1.setDim(10, 20, 15);
mybox2.setDim(3, 6, 9);
// get volume of first box vol =
mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

As you can see, the **setDim( )** method is used to set the dimensions of each box. For example, when mybox1.setDim(10, 20, 15);is executed, 10 is copied into parameter **w**, 20 is copied into **h**, and 15 is copied into **d**. Inside **setDim( )** the values of **w**, **h**, and **d** are then assigned to **width**, **height**, and **depth**, respectively. For many readers, the concepts presented in the preceding sections will be familiar. However, if such things as method calls, arguments, and parameters are new to you, then you might want to take some time to experiment before moving on. The concepts of the method invocation, parameters, and return values are fundamental to Java programming.

4.4 CONSTRUCTORS

It can be tedious to initialize all of the variables in a class each time an instance is created.Even when you add convenience functions like **setDim( )**, it would be simpler and more concise to have all of the setup done at the time the object is first created. Because the requirement for initialization is so common, Java allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of a constructor. A *constructor* initializes an object immediately upon creation. It has the same name as the class in which it resides and is syntactically similar to a method. Once defined, the constructor is automatically called immediately after the object is created, before the **new** operator completes. Constructors look a little strange because they have no return type, not even **void**. This is because the implicit return type of a class' constructor is the class type itself. It is the constructor's job to initialize the internal state of an object so that the code creating an instance will have a fully initialized, usable object immediately. You can rework the **Box** example so that the dimensions of a box are automatically initialized when an object is constructed. To do so, replace **setDim( )** with a constructor. Let's begin by defining a simple constructor that simply sets the dimensions of each box to the same values. This version is shown here:

```
/* Here, Box uses a constructor to initialize the
dimensions of a box.
*/
class Box {
double width;
double height;
double depth;
// This is the constructor for Box. Box()
{ System.out.println("Constructing
Box"); width = 10;

height = 10;
depth = 10;
}
// compute and return
volume double volume() {
return width * height * depth;
}
}
class BoxDemo6 {
public static void main(String args[]) {
// declare, allocate, and initialize Box
objects Box mybox1 = new Box();
Box mybox2 = new
Box(); double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box vol =
mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

When this program is run, it generates the following results:

```
Constructing Box
Constructing Box
Volume is 1000.0
Volume is 1000.0
```

As you can see, both **mybox1** and **mybox2** were initialized by the **Box( )** constructor when they were created. Since the constructor gives all boxes the same dimensions, 10 by 10 by 10, both **mybox1** and **mybox2** will have the same volume. The **println( )** statement inside **Box( )** is for the sake of illustration only. Most constructors will not display anything. They will simply initialize an object. Before moving on, let's reexamine the **new** operator. As you know, when you allocate an object, you use the following general form:

```
class-var = new classname( );
```

Now you can understand why the parentheses are needed after the class name. What is actually happening is that the constructor for the class is being called. Thus, in the line Box mybox1 = new Box(); **new Box( )** is calling the **Box( )** constructor. When you do not explicitly define a constructor for a class, then Java creates a default constructor for the class. This is why the preceding line of code worked in earlier versions of **Box** that did not define a constructor. The default constructor automatically initializes all instance variables to zero. The default constructor is often sufficient for simple classes, but it usually won't do for more sophisticated ones. Once you define your own constructor, the default constructor is no longer used.

Parameterized Constructors

While the **Box( )** constructor in the preceding example does initialize a **Box** object, it is not very useful—all boxes have the same dimensions. What is needed is a way to construct **Box** objects of various dimensions. The easy solution is to add parameters to the constructor. As you can probably guess, this makes them much more useful. For example, the following version of **Box** defines a parameterized constructor which sets the dimensions of a box as specified by those parameters. Pay special attention to how **Box** objects are created.

```
/* Here, Box uses a parameterized constructor to
initialize the dimensions of a box.
*/
class Box {
double width;
double height;
double depth;

// This is the constructor for Box.
Box(double w, double h, double d)
{ width = w;
height = h;
depth = d;
}
// compute and return
volume double volume() {
return width * height * depth;
}
}
class BoxDemo7 {
public static void main(String args[]) {
// declare, allocate, and initialize Box
objects Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box(3, 6,
9); double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box vol =
mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

The output from this program is shown here:

```
Volume is 3000.0
Volume is 162.0
```

As you can see, each object is initialized as specified in the parameters to its constructor. For example, in the following line, Box mybox1 = new Box(10, 20, 15); the values 10, 20, and 15 are passed to the **Box( )** constructor when **new** creates the object. Thus, **mybox1**'s copy of **width**, **height**, and **depth** will contain the values 10, 20, and 15, respectively.

4.5 THE this KEYWORD

Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the **this** keyword. **this** can be used inside any method to refer to the *current* object. That is, **this** is always a reference to the object on which the method was invoked. You can use **this** anywhere a reference to an object of the current class' type is permitted. To better understand what **this** refers to, consider the following version of **Box( )**:

```
// A redundant use of this.
Box(double w, double h, double d)
{ this.width = w;
this.height = h;
this.depth = d;
}
```

This version of **Box( )** operates exactly like the earlier version. The use of **this** is redundant, but perfectly correct. Inside **Box( )**, **this** will always refer to the invoking object. While it is redundant in this case, **this** is useful in other contexts, one of which is explained in the next section.

4.6. GARBAGE COLLECTION

Since objects are dynamically allocated by using the **new** operator, you might be wondering how such objects are destroyed and their memory released for later reallocation. In some languages, such as C++, dynamically allocated objects must be manually released by use of a **delete** operator. Java takes a different approach; it handles deallocation for you automatically. The technique that accomplishes this is called *garbage collection.* It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed. There is no explicit need to destroy objects as in C++. Garbage collection only occurs sporadically (if at all) during the execution of your program. It will not occur simply because one or more objects exist that are no longer used. Furthermore, different Java run-time implementations will take varying approaches to garbage collection, but for the most part, you should not have to think about it while writing your programs.

The finalize( ) Method

Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-Java resource such as a file handle or window character font, then you might want to make sure these resources are freed before an object is destroyed. To handle such situations, Java provides a mechanism called *finalization.* By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.To add a finalizer to a class, you simply define the **finalize( )** method. The Java run time calls that method whenever it is about to recycle an object of that class. Inside the **finalize( )** method you will specify those actions that must be performed before an object is destroyed. The garbage collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects. Right before an asset is freed, the Java run time calls the **finalize( )** method on the object. The **finalize( )** method has this general form:

```
protected void finalize( )
{
// finalization code here
}
```

Here, the keyword **protected** is a specifier that prevents access to **finalize( )** by code defined outside its class. This and the other access specifiers are explained in Topic 7.
It is important to understand that **finalize( )** is only called just prior to garbage collection. It is not called when an object goes out-of-scope, for example. This means that you cannot know when—or even if—**finalize( )** will be executed. Therefore, your program should provide other means of releasing system resources, etc., used by the object. It must not rely on **finalize( )** for normal program operation.

A Stack Class
While the **Box** class is useful to illustrate the essential elements of a class, it is of little practical value. To show the real power of classes, this chapter will conclude with a more sophisticated example. As you recall from the discussion of object-oriented programming (OOP) presented in Chapter 2, one of OOP's most important benefits is the encapsulation of data and the code that manipulates that data. As you have seen, the class is the mechanism by which encapsulation is achieved in Java. By creating a class, you are creating a new data type that defines both the nature of the data being manipulated and the routines used to manipulate it. Further, the methods define a consistent and controlled interface to the class' data. Thus, you can use the class through its methods without having to worry about the details of its implementation or how the data is actually managed within the class. In a sense, a class is like a "data engine." No knowledge of what goes on inside the engine is required to use the engine through its controls. In fact, since the details are hidden, its inner workings can be changed as needed. As long as your code uses the class through its methods, internal details can change without causing side effects outside the class. To see a practical application of the preceding discussion, let's develop one of the archetypal examples of encapsulation: the stack. A *stack* stores data using first-in, last-out ordering. That is, a stack is like a stack of plates on a table—the first plate put down on the table is the last plate to be used. Stacks are controlled through two operations traditionally called *push* and *pop.* To put an item on top of the stack, you will use push. To take an item off the stack, you will use pop. As you will see, it is easy to encapsulate

the entire stack mechanism.
Here is a class called **Stack** that implements a stack for integers:

```
// This class defines an integer stack that can hold 10
values. class Stack {
int stck[ ] = new int[10];
int tos;
// Initialize top-of-stack
Stack() {
tos = -1;
}
// Push an item onto the stack
void push(int item) { if(tos==9)
System.out.println("Stack is full.");
else

stck[++tos] = item;
}
// Pop an item from the stack
int pop() {
if(tos < 0) {
System.out.println("Stack underflow.");
return 0;
}
else
return stck[tos--];
}
}
```

As you can see, the **Stack** class defines two data items and three methods. The stack of integers is held by the array **stck**. This array is indexed by the variable **tos**, which always contains the index of the top of the stack. The **Stack( )** constructor initializes **tos** to –1, which indicates an empty stack. The method **push( )** puts an item on the stack. To retrieve an item, call **pop( )**. Since access to the stack is through **push( )** and **pop( )**, the fact that the stack is held in an array is actually not relevant to using the stack. For example, the stack could be held in a more complicated data structure, such as a linked list, yet the interface defined by **push( )** and **pop( )** would remain the same. The class **TestStack**, shown here, demonstrates the **Stack** class. It creates two integer stacks, pushes some values onto each, and then pops them off.

A Closer Look at Argument Passing
In general, there are two ways that a computer language can pass an argument to a subroutine. The first way is *call-by-value.* This method copies the *value* of an argument into the formal parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument. The second way an argument can be passed is *call-by-reference.* In this method, a reference to an argument (not the value of the argument) is passed to the parameter. Inside the subroutine, this reference is used to access the actual argument specified in the call. This means that changes made to the

parameter will affect the argument used to call the subroutine. As you will see, Java uses both approaches, depending upon what is passed. In Java, when you pass a simple type to a method, it is passed by value. Thus, what occurs to the parameter that receives the argument has no effect outside the method. For example, consider the following program:

```
// Simple types are passed by
value. class Test {
void meth(int i, int j)
{ i *= 2;
j /= 2;
}
}

class CallByValue {
public static void main(String args[]) {
Test ob = new Test();
int a = 15, b = 20;
System.out.println("a and b before call: " +
a + " " + b);
ob.meth(a, b);
System.out.println("a and b after call: " +
a + " " + b);
}
}
```

The output from this program is shown here:

```
a and b before call: 15 20
a and b after call: 15 20
```

As you can see, the operations that occur inside **meth( )** have no effect on the values of **a** and **b** used in the call; their values here did not change to 30 and 10. When you pass an object to a method, the situation changes dramatically, because objects are passed by reference. Keep in mind that when you create a variable of a class type, you are only creating a reference to an object. Thus, when you pass this reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument. This effectively means that objects are passed to methods by use of call-by-reference. Changes to the object inside the method *do* affect the object used as an argument. For example, consider the following program:

```
// Objects are passed by
reference. class Test {
int a, b; Test(int
i, int j) { a = i;

b = j;
}

// pass an object
void meth(Test o) {
o.a *= 2;
o.b /= 2;
}
}
class CallByRef {
public static void main(String args[]) {
Test ob = new Test(15, 20);
System.out.println("ob.a and ob.b before call: " +
ob.a + " " + ob.b);
ob.meth(ob);
System.out.println("ob.a and ob.b after call: " +
ob.a + " " + ob.b);
}
}
```

This program generates the following output:

```
ob.a and ob.b before call: 15 20
ob.a and ob.b after call: 30 10
```

As you can see, in this case, the actions inside **meth( )** have affected the object used as an argument. As a point of interest, when an object reference is passed to a method, the reference itself is passed by use of call-by-value. However, since the value being passed refers to an object, the copy of that value will still refer to the same object that its corresponding argument does.

4.7 UNDERSTANDING static KEYWORD

There will be times when you will want to define a class member that will be used independently of any object of that class. Normally a class member must be accessed only in conjunction with an object of its class. However, it is possible to create a member that can be used by itself, without reference to a specific instance. To create such a member, precede its declaration with the keyword **static**. When a member is declared **static**, it can be accessed before any objects of its class are created, and without reference to any object. You can declare both methods and variables to be **static**. The most common example of a **static** member is **main( )**. **main( )** is declared as **static** because it must be called before any objects exist. Instance variables declared as **static** are, essentially, global variables. When objects of its class are declared, no copy of a **static** variable is made. Instead, all instances of the class share the same **static** variable. Methods declared as **static** have several restrictions:

- They can only call other **static** methods.
- They must only access **static** data.
- They cannot refer to **this** or **super** in any way. (The keyword **super** relates to inheritance and is described in the next topic.)

If you need to do computation in order to initialize your **static** variables, you can declare a **static** block which gets executed exactly once, when the class is first loaded.The following example shows a class that has a **static** method, some **static** variables, and a **static** initialization block:

```
// Demonstrate static variables, methods, and
blocks. class UseStatic {
static int a =
3; static int b;
static void meth(int x) {
System.out.println("x = " + x);
System.out.println("a = " + a);
System.out.println("b = " + b);
}
static {
System.out.println("Static block
initialized."); b = a * 4;
}
public static void main(String args[])
{ meth(42);
}
}
```

As soon as the **UseStatic** class is loaded, all of the **static** statements are run. First, **a** is set to **3**, then the **static** block executes (printing a message), and finally, **b** is initialized to **a * 4** or **12**. Then **main( )** is called, which calls **meth( )**, passing **42** to **x**. The three **println( )** statements refer to the two **static** variables **a** and **b**, as well as to the local variable **x**.

Note: *It is illegal to refer to any instance variables inside of a **static** method.*

Here is the output of the program:

```
Static block initialized.
x = 42
a = 3
b = 12
```

Outside of the class in which they are defined, **static** methods and variables can be used independently of any object. To do so, you need only specify the name of their class followed by the dot operator. For example, if you wish to call a **static** method from outside its class, you can do so using the following general form:

classname.method( )

Here, *classname* is the name of the class in which the **static** method is declared. As you can see, this format is similar to that used to call non-**static** methods through object-reference variables. A **static** variable can be accessed in the same way—by use of the dot operator on the name of the class. This is how Java implements a controlled version of global methods and global variables. Here is an example. Inside **main( )**, the **static** method **callme( )** and the **static** variable **b** are accessed outside of their class.

```
class StaticDemo {
static int a = 42;
static int b = 99;
static void callme() {
System.out.println("a = " + a);
}
}
class StaticByName {
public static void main(String args[]) {
StaticDemo.callme();
System.out.println("b = " + StaticDemo.b);
}
}
```

Here is the output of this program:

```
a = 42
b = 99
```

4.8 INTRODUCING THE final MODIFIER

A variable can be declared as **final**. Doing so prevents its contents from being modified. This means that you must initialize a **final** variable when it is declared.(In this usage, **final** is similar to **const** in C/C++/C#.) For example:

```
final int FILE_NEW = 1;
final int FILE_OPEN = 2;
final int FILE_SAVE = 3;
final int FILE_SAVEAS = 4;
final int FILE_QUIT = 5;
```

Subsequent parts of your program can now use **FILE_OPEN**, etc., as if they were constants, without fear that a value has been changed. It is a common coding convention to choose all uppercase identifiers for **final** variables. Variables declared as **final** do not occupy memory on a per-instance basis.Thus, a **final** variable is essentially a constant.
The keyword **final** can also be applied to methods, but its meaning is substantially different than when it is applied to variables. This second usage of **final** is described in the next topic, when inheritance is described.

4.9 ARRAYS REVISITED

Arrays were introduced earlier in this book, before classes had been discussed. Now that you know about classes, an important point can be made about arrays: they are implemented as objects. Because of this, there is a special array attribute that you will want to take advantage of. Specifically, the size of an array—that is, the number of elements that an array can hold—is found in its **length** instance variable. All arrays have this variable, and it will always hold the size of the array. Here is a program that demonstrates this property:

```
// This program demonstrates the length array
member. class Length {
public static void main(String args[])
{ int a1[] = new int[10];
int a2[] = {3, 5, 7, 1, 8, 99, 44, -10};
int a3[] = {4, 3, 2, 1};
System.out.println("length of a1 is " + a1.length);
System.out.println("length of a2 is " + a2.length);
System.out.println("length of a3 is " + a3.length);
}
}
```

This program displays the following output:

```
length of a1 is 10
length of a2 is 8
length of a3 is 4
```

As you can see, the size of each array is displayed. Keep in mind that the value of **length** has nothing to do with the number of elements that are actually in use. It only reflects the number of elements that the array is designed to hold.

LEARNING ACTIVITY4:

---

1. Define Class and it's form in declaration.
2. How to Declare Objects.
3. Explain about Methods.
4. Define Constructors and it's importance in programming.
5. Explain about THIS keyword.
6. What is Garbage Collection.
7. Define the static to members of class.

TOPIC 5

5. INHERITANCE, POLYMORPHISM AND EXCEPTION HANDLING

LEARNING OUTCOMES

After studying this topic, you should be able to:

- Describe features of Inheritance.
  - Inheritance Basics.
  - Polymorphism.
  - Method Overloading.
  - Method Overriding.
- Understand Exception Handling.
- Describe Exception Types.
- Understand Java's Built-in Exception.

5.1. INHERITANCE

Inheritance is one of the cornerstones of object-oriented programming because it allows the creation of hierarchical classifications. Using inheritance, you can create a general class that defines traits common to a set of related items. This class can then be inherited by other, more specific classes, each adding those things that are unique to it. In the terminology of Java, a class that is inherited is called a *superclass*. The class that does the inheriting is called a *subclass*. Therefore, a subclass is a specialized version of a superclass. It inherits all of the instance variables and methods defined by the superclass and adds its own, unique elements.

Inheritance Basics

To inherit a class, you simply incorporate the definition of one class into another by using the **extends** keyword. To see how, let's begin with a short example. The following program creates a superclass called **A** and a subclass called **B**. Notice how the keyword **extends** is used to create a subclass of **A**.

```
// A simple example of inheritance.
// Create a superclass.
class A {
int i, j;
void showij() {
System.out.println("i and j: " + i + " " + j);
}
}


// Create a subclass by extending class
A. class B extends A {
int k;
void showk() {
System.out.println("k: " + k);
}
void sum() { System.out.println("i+j+k:
" + (i+j+k));
}
}

class SimpleInheritance {
public static void main(String args[]) {
A superOb = new A();
B subOb = new B();
// The superclass may be used by
itself. superOb.i = 10;
superOb.j = 20;
System.out.println("Contents of superOb:
"); superOb.showij(); System.out.println();

/* The subclass has access to all public members
of its superclass. */
subOb.i = 7; subOb.j = 8; subOb.k = 9;
System.out.println("Contents of subOb:
"); subOb.showij();


subOb.showk();
System.out.println();
System.out.println("Sum of i, j and k in
subOb:"); subOb.sum();
}
}
```

The output from this program is shown here:

```
Contents of superOb:
i and j: 10 20
Contents of subOb:
i and j: 7 8
k: 9
Sum of i, j and k in subOb:
i+j+k: 24
```

As you can see, the subclass **B** includes all of the members of its superclass, **A**. This is why **subOb** can access **i** and **j** and call **showij( )**. Also, inside **sum( )**, **i** and **j** can be referred to directly, as if they were part of **B**. Even though **A** is a superclass for **B**, it is also a completely independent, stand-alone class. Being a superclass for a subclass does not mean that the superclass cannot be used by itself. Further, a subclass can be a superclass for another subclass. The general form of a **class** declaration that inherits a superclass is shown here:

```
class subclass-name extends superclass-name {
// body of class
}
```

You can only specify one superclass for any subclass that you create. Java does not support the inheritance of multiple superclasses into a single subclass. (This differs from C++, in which you can inherit multiple base classes.) You can, as stated, create a hierarchy of inheritance in which a subclass becomes a superclass of another subclass. However, no class can be a superclass of itself.

Member Access and Inheritance

Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as **private**. For example, consider the following simple class hierarchy:

```
/* In a class hierarchy, private members remain
private to their class.
This program contains an error and will not
compile.
*/
// Create a
superclass. class A {
int i; // public by default
private int j; // private to
A void setij(int x, int y) {
i = x;
j = y;
}
}

// A's j is not accessible
here. class B extends A {
int total;
void sum() {
total = i + j; // ERROR, j is not accessible here
}
}

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

```
B subOb = new B();
subOb.setij(10, 12);
subOb.sum();
System.out.println("Total is " + subOb.total);
}
}
```

This program will not compile because the reference to **j** inside the **sum( )** method of **B** causes an access violation. Since **j** is declared as **private**, it is only accessible by other members of its own class. Subclasses have no access to it.
A class member that has been declared as private will remain private to its class. It is not accessible by any code outside its class, including subclasses.

A More Practical Example

Let's look at a more practical example that will help illustrate the power of inheritance. Here, the final version of the **Box** class developed in the preceding chapter will be extended to include a fourth component called **weight**. Thus, the new class will contain a box's width, height, depth, and weight.

```
// This program uses inheritance to extend
Box. class Box {
double  width;
double height;
double depth;
// construct clone of an object
Box(Box ob) { // pass object to constructor
width = ob.width;
height = ob.height;

depth = ob.depth;
}
// constructor used when all dimensions
specified Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// constructor used when no dimensions
specified Box() {
width = -1; // use -1 to
indicate height = -1; // an
uninitialized depth = -1; // box
}
// constructor used when cube is
created Box(double len) {
width = height = depth = len;
}
// compute and return
volume double volume() {
return width * height * depth;
}
}

// Here, Box is extended to include
weight. class BoxWeight extends Box {
double weight; // weight of box
// constructor for BoxWeight
BoxWeight(double w, double h, double d, double m) {
width = w;
height = h;
depth = d;
weight = m;
}
}

class DemoBoxWeight {
public static void main(String args[]) {
BoxWeight mybox1 = new BoxWeight(10, 20, 15,
34.3); BoxWeight mybox2 = new BoxWeight(2, 3, 4,
0.076); double vol;
vol = mybox1.volume();
System.out.println("Volume of mybox1 is " + vol);
System.out.println("Weight of mybox1 is " + mybox1.weight);
System.out.println();
vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);
System.out.println("Weight of mybox2 is " + mybox2.weight);
}
}
```

The output from this program is shown here:

```
Volume of mybox1 is 3000.0
Weight of mybox1 is 34.3
Volume of mybox2 is 24.0
Weight of mybox2 is 0.076
```

BoxWeight inherits all of the characteristics of **Box** and adds to them the **weight** component. It is not necessary for **BoxWeight** to re-create all of the features found in **Box**. It can simply extend **Box** to meet its own purposes. A major advantage of inheritance is that once you have created a superclass that defines the attributes common to a set of objects, it can be used to create any number of more specific subclasses. Each subclass can precisely tailor its own classification. For example, the following class inherits **Box** and adds a color attribute:

```
// Here, Box is extended to include
color. class ColorBox extends Box {
int color; // color of box
ColorBox(double w, double h, double d, int c)
{ width = w;
height = h;
depth = d;
color = c;
}
}
```

Remember, once you have created a superclass that defines the general aspects of an object, that superclass can be inherited to form specialized classes. Each subclass simply adds its own, unique attributes. This is the essence of inheritance.

When Constructors Are Called?

When a class hierarchy is created, in what order are the constructors for the classes that make up the hierarchy called? For example, given a subclass called **B** and a superclass called **A**, is **A**'s constructor called before **B**'s, or vice versa? The answer is that in a class hierarchy, constructors are called in order of derivation, from superclass to subclass.Further, since **super( )** must be the first statement executed in a subclass' constructor,this order is the same whether or not **super( )** is used. If **super( )** is not used, then the default or parameterless constructor of each superclass will be executed. The following program illustrates when constructors are executed:

```
// Demonstrate when constructors are called.
// Create a super class.
class A {
A() {
System.out.println("Inside A's constructor.");
}
}

// Create a subclass by extending class
A. class B extends A {
B() {
System.out.println("Inside B's constructor.");
}
}

// Create another subclass by extending
B. class C extends B {
C() {
System.out.println("Inside C's constructor.");
}
}
```

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

C c = new C();
}
}
```

The output from this program is shown here:

```
Inside A's constructor
Inside B's constructor
Inside C's constructor
```

As you can see, the constructors are called in order of derivation. If you think about it, it makes sense that constructors are executed in order of derivation. Because a superclass has no knowledge of any subclass, any initialization it needs to perform is separate from and possibly prerequisite to any initialization performed by the subclass. Therefore, it must be executed first.

5.2. POLYMORPHISM

Polymorphism gives us the ultimate flexibility in extensibility. Polymorphism is a term that describes a situation where one name may refer to different methods. In java there are two type of polymorphism: overloading type and overriding type.When you override methods, java determines the proper methods to call at the program's run time, not at the compile time. Overriding occurs when a class method has the same name and signature as a method in parent class. Overloading occurs when several methods have same names with different method signature. Overloading is determined at the compile time.

Example:

```
Class Book{
String title;
String publisher;
float price;
setBook(String title){
}
setBook(String title, String publisher){
}
setBook(String title, String publisher,float price){
}
}
```

5.3. METHOD OVERRIDING

Example: Overriding

```
Class Tool{
void operate_it(){
}
}
 Class ScrewDriver extends Tool{
            void operate_it(){
}
}
```

In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to *override* the method in the superclass. When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden. Consider the following:

```
// Method
overriding. class A {
int i, j;
A(int a, int b)
{ i = a;
j = b;
}
// display i and j
void show() {
System.out.println("i and j: " + i + " " + j);
}
}

class B extends A {
int k;
B(int a, int b, int c) {
super(a, b);
k = c;
}
// display k – this overrides show() in
A void show() { System.out.println("k:
" + k);
}
}
class Override {
public static void main(String args[]) {
```

```
B subOb = new B(1, 2, 3);
subOb.show(); // this calls show() in B
}
}
```

The output produced by this program is shown here:

```
k: 3
```

When **show( )** is invoked on an object of type **B**, the version of **show( )** defined within **B** is used. That is, the version of **show( )** inside **B** overrides the version declared in **A**. If you wish to access the superclass version of an overridden function, you can do so by using **super**. For example, in this version of **B**, the superclass version of **show( )** is invoked within the subclass' version. This allows all instance variables to be displayed.

```
class B extends A {
int k;
B(int a, int b, int c) {
super(a, b);
k = c;
}
void show() {
super.show(); // this calls A's show()
System.out.println("k: " + k);
}
}
```

If you substitute this version of **A** into the previous program, you will see
the following output:

```
i and j: 1 2
k: 3
```

Here, **super.show( )** calls the superclass version of **show( )**.Method overriding occurs *only* when the names and the type signatures of the two methods are identical. If they are not, then the two methods are simply overloaded. For example, consider this modified version of the preceding example:

```
// Methods with differing type signatures are overloaded – not
// overridden.
class A {
int i, j;
A(int a, int b) {
i = a;
j = b;
}
// display i and j void show() {
System.out.println("i and j: " + i + " " +
j);
}
}

// Create a subclass by extending class A.
```

```
class B extends A {
int k;
B(int a, int b, int c) {
super(a, b);
k = c;

}
// overload show() void
show(String msg) {
System.out.println(msg + k);
}
}

class Override {
public static void main(String args[]) {
B subOb = new B(1, 2, 3);
subOb.show("This is k: "); // this calls show() in B
subOb.show(); // this calls show() in A
}
}
```

The output produced by this program is shown here:

```
This is k: 3
i and j: 1 2
```

The version of **show( )** in **B** takes a string parameter. This makes its type signature different from the one in **A**, which takes no parameters. Therefore, no overriding (or name hiding) takes place.

Why Overridden Methods?

As stated earlier, overridden methods allow Java to support run-time polymorphism. Polymorphism is essential to object-oriented programming for one reason: it allows a general class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods. Overridden methods are another way that Java implements the "one interface, multiple methods" aspect of polymorphism. Part of the key to successfully applying polymorphism is understanding that the superclasses and subclasses form a hierarchy which moves from lesser to greater specialization. Used correctly, the superclass provides all elements that a subclass can use directly. It also defines those methods that the derived class must implement on its own. This allows the subclass the flexibility to define its own methods, yet still enforces a consistent interface. Thus, by combining inheritance with overridden methods, a superclass can define the general form of the methods that will be used by all of its subclasses.

Using final with Inheritance

The keyword **final** has three uses. First, it can be used to create the equivalent of a named constant. This use was described in the preceding chapter. The other two uses of **final** apply to inheritance. Both are examined here.

Using final to Prevent Overriding

While method overriding is one of Java's most powerful features, there will be times when you will want to prevent it from occurring. To disallow a method from being overridden, specify **final** as a modifier at the start of its declaration. Methods declared as **final** cannot be overridden. The following fragment illustrates **final**:

```
class A {
final void meth() {
System.out.println("This is a final method.");
}
}

class B extends A {
void meth() { // ERROR! Can't override.
System.out.println("Illegal!");
}
}
```

Because **meth( )** is declared as **final**, it cannot be overridden in **B**. If you attempt to do so, a compile-time error will result. Methods declared as **final** can sometimes provide a performance enhancement: The compiler is free to *inline* calls to them because it "knows" they will not be overridden by a subclass. When a small **final** method is called, often the Java compiler can copy the bytecode for the subroutine directly inline with the compiled code of the calling method, thus eliminating the costly overhead associated with a method call. Inlining is only an option with **final** methods. Normally, Java resolves calls to methods dynamically, at run time. This is called *late binding.* However, since **final** methods cannot be overridden, a call to one can be resolved at compile time. This is called *early binding.*

Using final to Prevent Inheritance

Sometimes you will want to prevent a class from being inherited. To do this, precede the class declaration with **final**. Declaring a class as **final** implicitly declares all of its methods as **final**, too. As you might expect, it is illegal to declare a class as both **abstract** and **final** since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.
Here is an example of a **final** class:

```
final class A {
// ...
}
// The following class is illegal.
class B extends A { // ERROR! Can't subclass A
// ...
}
```

As the comments imply, it is illegal for **B** to inherit **A** since **A** is declared as **final**.

5.4 EXCEPTION HANDILING

This chapter examines Java's exception-handling mechanism. An *exception* is an abnormal condition that arises in a code sequence at run time. In other words, an exception is a run-time error. In computer languages that do not support exception handling, errors must be checked and handled manually—typically through the use of error codes, and so on. This approach is as cumbersome as it is troublesome. Java's exception handling avoids these problems and, in the process, brings run-time error management into the object-oriented world. For the most part, exception handling has not changed since the original version of Java. However, Java 2, version 1.4 has added a new subsystem called the *chained exception facility*. This feature is described near the end of this chapter.

Exception-Handling Fundamentals

A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code. When an exceptional condition arises, an object representing that exception is created and *thrown* in the method that caused the error.
That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is *caught* and processed. Exceptions can be generated by the Java run-time system, or they can be manually generated by your code. Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment. Manually generated exceptions are typically used to report some error condition to the caller of a method. Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**. Briefly, here is how they work. Program statements that you want to monitor for exceptions are contained within a **try** block. If an exception occurs within the **try** block, it is thrown. Your code can catch this exception (using **catch**) and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword **throw**. Any exception that is thrown out of a method must be specified as such by a **throws** clause. Any code that absolutely must be executed before a method returns is put in a **finally** block.

This is the general form of an exception-handling block:

```
try {
// block of code to monitor for errors
}
catch (ExceptionType1 exOb) {
// exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
```

```
// exception handler for ExceptionType2
}
// ...
finally {
// block of code to be executed before try block ends
}
```

Here, *ExceptionType* is the type of exception that has occurred. The remainder of this chapter describes how to apply this framework.

Exception Types

All exception types are subclasses of the built-in class **Throwable**. Thus, **Throwable** is at the top of the exception class hierarchy. Immediately below **Throwable** are twosubclasses that partition exceptions into two distinct branches. One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch. This is also the class that you will subclass to create your own custom exception types. There is an important subclass of **Exception**, called **RuntimeException**. Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing. The other branch is topped by **Error**, which defines exceptions that are not expected to be caught under normal circumstances by your program. Exceptions of type **Error** are used by the Java run-time system to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error. This chapter will not be dealing with exceptions of type **Error**, because these are typically created in response to catastrophic failures that cannot usually be handled by your program.

Uncaught Exceptions

Before you learn how to handle exceptions in your program, it is useful to see what happens when you don't handle them. This small program includes an expression that intentionally causes a divide-by-zero error.

```
class Exc0 {
public static void main(String args[]) {
int d = 0;
int a = 42 / d;
}
}
```

When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then *throws* this exception. This causes the execution of **Exc0** to stop, because once an exception has been thrown, it must be *caught* by an exception handler and dealt with immediately. In this example, we haven't supplied any exception handlers of our own, so the exception is caught by the default handler provided by the Java run-time system. Any exception that is not caught by your program will ultimately be processed by the default handler. The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.

Here is the output generated when this example is executed.
java.lang.ArithmeticException: / by zero
at Exc0.main(Exc0.java:4) Notice how the class name, **Exc0**; the method name, **main**; the filename, **Exc0.java**; and the line number, **4**, are all included in the simple stack trace. Also, notice that thetype of the exception thrown is a subclass of **Exception** called **ArithmeticException**,which more specifically describes what type of error happened. As discussed later inthis chapter, Java supplies several built-in exception types that match the various sortsof run-time errors that can be generated.The stack trace will always show the sequence of method invocations that led up to the error. For example, here is another version of the preceding program that introduces the same error but in a method separate from **main( )**:

```
class Exc1 {
static void subroutine() {
int d = 0;
int a = 10 / d;
}
public static void main(String args[]) {
Exc1.subroutine();
}
}
```

The resulting stack trace from the default exception handler shows how the entire call stack is displayed:

```
java.lang.ArithmeticException: / by zero
at Exc1.subroutine(Exc1.java:4)
at Exc1.main(Exc1.java:7)
```

As you can see, the bottom of the stack is **main**'s line 7, which is the call to **subroutine( )**, which caused the exception at line 4. The call stack is quite useful for debugging, because it pinpoints the precise sequence of steps that led to the error.

Using try and catch

Although the default exception handler provided by the Java run-time system is useful for debugging, you will usually want to handle an exception yourself. Doing so provides two benefits. First, it allows you to fix the error. Second, it prevents the program from automatically terminating. Most users would be confused (to say the least) if your program stopped running and printed a stack trace whenever an error occurred! Fortunately, it is quite easy to prevent this. To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a **try** block. Immediately following the **try** block, include a **catch** clause that specifies the exception type that you wish to catch. To illustrate how easily this can be done, the following program includes a **try** block and a **catch** clause which processes the **ArithmeticException** generated by the division-by-zero error:

```
class Exc2 {
```

```
public static void main(String args[ ]) {
int d, a;
try { // monitor a block of code.
d = 0;
a = 42 / d;
System.out.println("This will not be printed.");
} catch (ArithmeticException e) { // catch divide-by-zero
error System.out.println("Division by zero.");
}
System.out.println("After catch statement.");
}
}
```

This program generates the following output:

```
Division by zero.
After catch statement.
```

Notice that the call to **println( )** inside the **try** block is never executed. Once an exception is thrown, program control transfers out of the **try** block into the **catch** block. Put differently, **catch** is not “called,” so execution never “returns” to the **try** block from a **catch**. Thus, the line “This will not be printed.” is not displayed. Once the **catch** statement has executed, program control continues with the next line in the program following the entire **try**/**catch** mechanism.

A **try** and its **catch** statement form a unit. The scope of the **catch** clause is restricted to those statements specified by the immediately preceding **try** statement. A **catch** statement cannot catch an exception thrown by another **try** statement (except in the case of nested **try** statements, described shortly). The statements that are protected by **try** must be surrounded by curly braces. (That is, they must be within a block.) You cannot use **try** on a single statement. The goal of most well-constructed **catch** clauses should be to resolve the exceptional condition and then continue on as if the error had never happened. For example, in the next program each iteration of the **for** loop obtains two random integers. Those two integers are divided by each other, and the result is used to divide the value 12345. The final result is put into **a**. If either division operation causes a divide-by-zero error, it is caught, the value of **a** is set to zero, and the program continues.

```
// Handle an exception and move
on. import java.util.Random;
class HandleError {
public static void main(String args[])
{ int a=0, b=0, c=0;
Random r = new
Random(); for(int i=0;
i<32000; i++) { try {
b = r.nextInt();
c = r.nextInt();
a = 12345 / (b/c);
} catch (ArithmeticException e) {
System.out.println("Division by zero.");
a = 0; // set a to zero and continue
}
System.out.println("a: " + a);
}
}
}
```

A **try** and its **catch** statement form a unit. The scope of the **catch** clause is restricted to those statements specified by the immediately preceding **try** statement. A **catch** statement cannot catch an exception thrown by another **try** statement (except in the case of nested **try** statements, described shortly). The statements that are protected by **try** must be surrounded by curly braces. (That is, they must be within a block.) You cannot use **try** on a single statement. The goal of most well-constructed **catch** clauses should be to resolve the exceptional condition and then continue on as if the error had never happened. For example, in the next program each iteration of the **for** loop obtains two random integers. Those two integers are divided by each other, and the result is used to divide the value 12345. The final result is put into **a**. If either division operation causes a divide-by-zero error, it is caught, the value of **a** is set to zero, and the program continues.

```
// Handle an exception and move
on. import java.util.Random;
class HandleError {
public static void main(String args[])
{ int a=0, b=0, c=0;
Random r = new
Random(); for(int i=0;
i<32000; i++) { try {
b = r.nextInt();
c = r.nextInt();
a = 12345 / (b/c);
} catch (ArithmeticException e) {
System.out.println("Division by zero.");
a = 0; // set a to zero and continue
}
System.out.println("a: " + a);
}
}
}
```

Displaying a Description of an Exception

Throwable overrides the **toString( )** method (defined by **Object**) so that it returns a string containing a description of the exception. You can display this description in a **println( )** statement by simply passing the exception as an argument. For example, the **catch** block in the preceding program can be rewritten like this:

```
catch (ArithmeticException e) {
```

```
System.out.println("Exception: " + e);
a = 0; // set a to zero and continue
}
```

When this version is substituted in the program, and the program is run, each divide-by-zero error displays the following message:

```
Exception: java.lang.ArithmeticException: / by zero
```

While it is of no particular value in this context, the ability to display a description of an exception is valuable in other circumstances—particularly when you are experimenting with exceptions or when you are debugging.

Multiple catch Clauses

In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, you can specify two or more **catch** clauses, each catching a different type of exception. When an exception is thrown, each **catch** statement is inspected in order, and the first one whose type matches that of the exception is executed. After one **catch** statement executes, the others are bypassed, and execution continues after the **try**/**catch** block. The following example traps two different exception types:

```
// Demonstrate multiple catch
statements. class MultiCatch {
public static void main(String args[])
{ try {
int a = args.length;
System.out.println("a = " + a);
int b = 42 / a;
int c[ ] = { 1
}; c[42] = 99;
}
catch(ArithmeticException e) {
System.out.println("Divide by 0: " + e);
}
catch(ArrayIndexOutOfBoundsException e) {
System.out.println("Array index oob: " + e);
}
System.out.println("After try/catch blocks.");
}
}
```

This program will cause a division-by-zero exception if it is started with no commandline parameters, since **a** will equal zero. It will survive the division if you provide a command-line argument, setting **a** to something larger than zero. But it will cause an **ArrayIndexOutOfBoundsException**, since the **int** array **c** has a length of 1, yet the program attempts to assign a value to **c[42]**.

Here is the output generated by running it both ways:

```
C:\>java MultiCatch
a = 0
Divide by 0: java.lang.ArithmeticException: / by zero

After try/catch blocks.

C:\>java MultiCatch TestArg
a = 1
```

```
Array index oob: java.lang.ArrayIndexOutOfBoundsException
After try/catch blocks.
```

When you use multiple **catch** statements, it is important to remember that exception subclasses must come before any of their superclasses. This is because a **catch** statement that uses a superclass will catch exceptions of that type plus any of its subclasses. Thus, a subclass would never be reached if it came after its superclass. Further, in Java, unreachable code is an error. For example, consider the following program:

```
/* This program contains an error.
A subclass must come before its superclass in
a series of catch statements. If not,
unreachable code will be created and a
compile-time error will result.
*/

class SuperSubCatch {
public static void main(String args[]) {
try {
int a = 0;
int b = 42 / a;
}
 catch(Exception e) {
System.out.println("Generic Exception catch.");
}
/* This catch is never reached because
ArithmeticException is a subclass of Exception. */
catch(ArithmeticException e) { // ERROR - unreachable
System.out.println("This is never reached.");

}
}
}
```

If you try to compile this program, you will receive an error message stating that the second **catch** statement is unreachable because the exception has already been caught.

Since **ArithmeticException** is a subclass of **Exception**, the first **catch** statement will handle all **Exception**-based errors, including **ArithmeticException**. This means that the second **catch** statement will never execute. To fix the problem, reverse the order of the **catch** statements.

Throw

So far, you have only been catching exceptions that are thrown by the Java run-time system. However, it is possible for your program to throw an exception explicitly, using the **throw** statement. The general form of **throw** is shown here:

throw *ThrowableInstance*;

Here, *ThrowableInstance* must be an object of type **Throwable** or a subclass of **Throwable**. Simple types, such as **int** or **char**, as well as non-**Throwable** classes, such as **String** and **Object**, cannot be used as exceptions. There are two ways you can obtain a **Throwable** object: using a parameter into a **catch** clause, or creating one with the **new** operator.The flow of execution stops immediately after the **throw** statement; any subsequent statements are not executed. The nearest enclosing **try** block is inspected to see if it has a **catch** statement that matches the type of the exception. If it does find a match, control is transferred to that statement. If not, then the next enclosing **try** statement is inspected, and so on. If no matching **catch** is found, then the default exception handler halts the program and prints the stack trace. Here is a sample program that creates and throws an exception. The handler that catches the exception rethrows it to the outer handler.

```
// Demonstrate throw.
class ThrowDemo {
static void demoproc()
{ try {
throw new NullPointerException("demo"); }
catch(NullPointerException e) {
System.out.println("Caught inside demoproc.");
throw e; // rethrow the exception
}
}
public static void main(String args[])
{ try {
demoproc();
} catch(NullPointerException e) {
System.out.println("Recaught: " + e);
}
}
}
```

This program gets two chances to deal with the same error. First, **main( )** sets up anexception context and then calls **demoproc( )**. The **demoproc( )** method then sets up

another exception-handling context and immediately throws a new instance of **NullPointerException,** which is caught on the next line. The exception is then rethrown.
Here is the resulting output:

```
Caught inside demoproc.
Recaught: java.lang.NullPointerException: demo
```

The program also illustrates how to create one of Java's standard exception objects.
Pay close attention to this line:throw new NullPointerException("demo");Here, **new** is used to construct an instance of **NullPointerException**. All of Java's built-in run-time exceptions have at least two constructors: one with no parameter and one that takes a string parameter. When the second form is used, the argument specifies a string that describes the exception. This string is displayed when the object is used as an argument to **print( )** or **println( )**. It can also be obtained by a call to **getMessage( )**, which is defined by **Throwable**.

Throws Clause

If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception. You do this by including a **throws** clause in the method's declaration. A **throws** clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type **Error** or **RuntimeException**, or any of their subclasses. All other exceptions that a method can throw must be declared in the **throws** clause. If they are not, a compile-time error will result. This is the general form of a method declaration that includes a **throws** clause:

```
type method-name(parameter-list) throws exception-list
{
// body of method
}
```

Here, *exception-list* is a comma-separated list of the exceptions that a method can throw.

Following is an example of an incorrect program that tries to throw an exception that it does not catch. Because the program does not specify a **throws** clause to declare this fact, the program will not compile.

```
// This program contains an error and will not
compile. class ThrowsDemo {
static void throwOne() {
System.out.println("Inside throwOne.");
throw new IllegalAccessException("demo");
}
public static void main(String args[ ])
{ throwOne();
}
}
```

To make this example compile, you need to make two changes. First, you need to declare that **throwOne( )** throws **IllegalAccessException**. Second, **main( )** must define a **try**/**catch** statement that catches this exception. The corrected example is shown here:

```
// This is now correct.
class ThrowsDemo {
static void throwOne() throws IllegalAccessException
{ System.out.println("Inside throwOne.");
throw new IllegalAccessException("demo");
}
public static void main(String args[])
{ try {
throwOne();
} catch (IllegalAccessException e) {
System.out.println("Caught " + e);
}
}
}
```

Here is the output generated by running this example program: inside throwOne caught java.lang.IllegalAccessException: demo

Finally Keyword.
When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method. Depending upon how the method is coded, it is even possible for an exception to cause the method to return prematurely. This could be a problem in some methods. For example, if a method opens a file upon entry and closes it upon exit, then you will not want the code that closes the file to be bypassed by the exception-handling mechanism. The **finally** keyword is designed to address this contingency. **finally** creates a block of code that will be executed after a **try**/**catch** block has completed and before the code following the **try/catch** block. The **finally** block will execute whether or not an exception is thrown. If an exception is thrown, the **finally** block will execute even if no **catch** statement matches the exception. Any time a method is about to return to the caller from inside a **try/catch** block, via an uncaught exception or an explicit return statement, the **finally** clause is also executed just before the method returns. This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning. The **finally** clause is optional. However, each **try** statement requires at least one **catch** or a **finally** clause. Here is an example program that shows three methods that exit in various ways, none without executing their **finally** clauses:

```
// Demonstrate finally.
class FinallyDemo {
// Through an exception out of the
method. static void procA() {
```

```
try {
System.out.println("inside procA");
throw new RuntimeException("demo");
} finally {
System.out.println("procA's finally");
}
}
// Return from within a try
block. static void procB() {
try {
System.out.println("inside
procB"); return;
} finally { System.out.println("procB's
finally");
}
}
// Execute a try block
normally. static void procC() {
try {
System.out.println("inside procC"); }
finally { System.out.println("procC's
finally");
}
}
public static void main(String args[])
{ try {
procA();
} catch (Exception e) {
System.out.println("Exception caught");
}
procB();
procC();
}
}
```

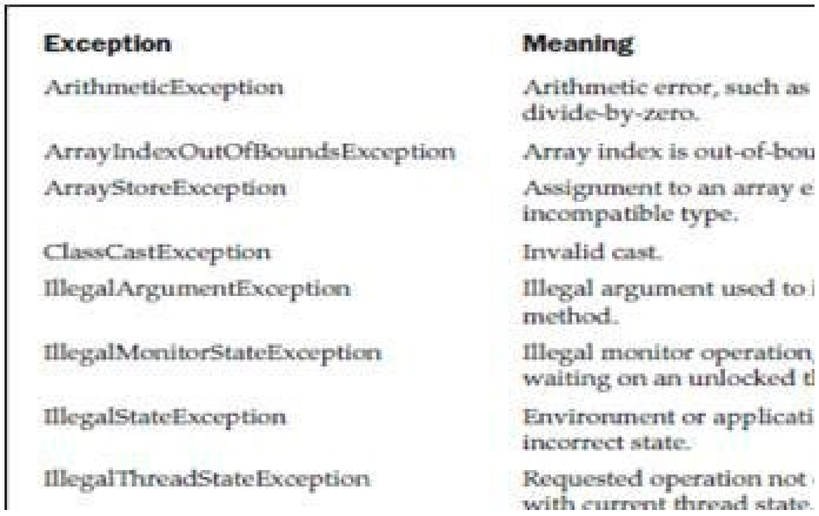
In this example, **procA( )** prematurely breaks out of the **try** by throwing an exception. The **finally** clause is executed on the way out. **procB( )**'s **try** statement is exited via a **return** statement. The **finally** clause is executed before **procB( )** returns. In **procC( )**, the **try** statement executes normally, without error. However, the **finally** block is still executed.
*If a **finally** block is associated with a **try**, the **finally** block will be executed upon conclusion of the **try**.*
Here is the output generated by the preceding program:

```
inside procA
procA's finally
Exception caught
inside procB
procB's finally
```

```
inside procC
procC's finally
```

5.6. JAVA'S BUILT-IN EXCEPTIONS

Inside the standard package **java.lang**, Java defines several exception classes. A few have been used by the preceding examples. The most general of these exceptions are subclasses of the standard type **RuntimeException**. Since **java.lang** is implicitly imported into all Java programs, most exceptions derived from **RuntimeException** are automatically available. Furthermore, they need not be included in any method's **throws** list. In the language of Java, these are called *unchecked exceptions* because the compiler does not check to see if a method handles or throws these exceptions. The unchecked exceptions defined in **java.lang** are listed in Table 10-1. Table 10-2 lists those exceptions defined by **java.lang** that must be included in a method's **throws** list if that method can generate one of these exceptions and does not handle it itself. These are called *checked exceptions.* Java defines several other types of exceptions that relate to its various class libraries.



Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bou
ArrayStoreException	Assignment to an array e incompatible type.
ClassCastException	Invalid cast.
IllegalArgumentException	Illegal argument used to i method.
IllegalMonitorStateException	Illegal monitor operation, waiting on an unlocked t
IllegalStateException	Environment or applicati incorrect state.
IllegalThreadStateException	Requested operation not with current thread state.

Table 5.1 Java's unchecked RuntimeException subclasses

LEARNING ACTIVITY 5

- What are Features of Inheritance?
- Describe Inheritance Basics.
- Explain about Polymorphism.
- Give detail Explanation about Method Overloading.
- Explain Method Overriding.
- Describe Exception Handling.
- Give List of Exception Types.
- Describe Java's Built in Exception.

Topic 6

EXCEPTIONS HANDLING AND FILES

Learning Outcome

After this topic, you should be able to:

- Describe exceptions
- Try code and catch exceptions
- Throw and catch multiple exceptions
- Use the finally block
- Specify the exceptions that a method can throw
- Trace exceptions through the call stack
- Understand I/O
- Writing to and reading from files

6.1 Learning about the Exceptions

An exception is an unexpected or error condition. The programs you write can generate many types of potential exceptions. As examples:

- A program might issue a command to read a file from a disk, but the file does not exist there.
- A program might attempt to write data to a disk, but the disk is full or unformatted.
- A program might ask for user input, but the user enters an invalid data type.
- A program might attempt to divide a value by 0.
- A program might try to access an array with a subscript that is too large or too mall.

These errors are called exceptions because, presumably, they are not usual occurrences; they are "exceptional." Exception handling is the name for the object-oriented techniques that manage or resolve such errors. Unplanned exceptions that occur during a program's execution are also called runtime exceptions, in contrast with syntax errors that are discovered during program compilation.

In Java, all exceptions are represented by classes. All exception classes are derived from a class called **Throwable**. Thus, when an exception occurs in a program, an object of some type of exception class is generated. There are two direct subclasses of **Throwable**: **Exception** and **Error**. Exceptions of type **Error** are related to errors that occur in the Java Virtual Machine itself, and not in your program. These types of exceptions are beyond your control, and your program will not usually deal with them. Thus, these types of exceptions are not described here.
Errors that result from program activity are represented by subclasses of **Exception**. For example, divide-by-zero, array boundary, and file errors fall into this category. In general, your program should handle exceptions of these types. An important subclass of **Exception** is **RuntimeException**, which is used to represent various common types of run-time errors.

Exception handling fundamentals

A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code. When an exceptional condition arises, an object representing that exception is created and *thrown* in the method that caused the error. That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is *caught* and processed. Exceptions can be generated by the Java run-time system, or they can be manually generated by your code. Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment. Manually generated exceptions are typically used to report some error condition to the caller of a method. Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**. Briefly, here is how they work. Program statements that you want to monitor for exceptions are contained within a **try** block. If an exception occurs within the **try** block, it is thrown. Your code can catch this exception (using **catch**) and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java runtime system. To manually throw an exception, use the keyword **throw**. Any exception that is thrown out of a method must be specified as such by a **throws** clause. Any code that absolutely must be executed after a **try** block completes is put in a **finally** block.

```
//This is the general form of an exception-handling block:
try {
// block of code to monitor for errors
}
catch (ExceptionType1 exOb) {
// exception handler for ExceptionType1
}
```

Here, *ExceptionType* is the type of exception that has occurred.

Exceptions enables you to handle errors gracefully. One of the key benefits of exception handling is that it enables your program to respond to an error and then continue running.

Exception Types

All exception types are subclasses of the built-in class **Throwable**. Thus, **Throwable** is at the top of the exception class hierarchy. Immediately below **Throwable** are two subclasses that partition exceptions into two distinct branches. One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch. This is also the class that you will subclass to create your own custom exception types. There is an important subclass of **Exception**, called **RuntimeException**. Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing.
The other branch is topped by **Error**, which defines exceptions that are not expected to be caught under normal circumstances by your program. Exceptions of type **Error** are used by the Java run-time system to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error. This chapter will not be dealing with

exceptions of type **Error**, because these are typically created in response to catastrophic failures that cannot usually be handled by your program.

Uncaught Exceptions

Before you learn how to handle exceptions in your program, it is useful to see what happens when you don't handle them. This small program includes an expression that intentionally causes a divide-by-zero error:

```
class Exc0 {
    public static void main(String args[]) {
    int d = 0;
    int a = 42 / d;
    }
}
```

When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then *throws* this exception. This causes the execution of **Exc0** to stop, because once an exception has been thrown, it must be *caught* by an exception handler and dealt with immediately. In this example, we haven't supplied any exception handlers of our own, so the exception is caught by the default handler provided by the Java run-time system. Any exception that is not caught by your program will ultimately be processed by the default handler. The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program. Here is the exception generated when this example is executed:

```
java.lang.ArithmeticException: / by zero
at Exc0.main(Exc0.java:4)
```

Notice how the class name, **Exc0**; the method name, **main**; the filename, **Exc0.java**; and the line number, **4**, are all included in the simple stack trace. Also, notice that the type of exception thrown is a subclass of **Exception** called **ArithmeticException**, which more specifically describes what type of error happened. As discussed later in this chapter, Java supplies several built-in exception types that match the various sorts of run-time errors that can be generated.

```
class Exc2 {
    public static void main(String args[]) {
        int d, a;
        try { // monitor a block of code.
            d = 0;
            a = 42 / d;
            System.out.println("This will not be printed.");
        } catch (ArithmeticException e) { // catch divide-by-zero error
            System.out.println("Division by zero.");
        }
        System.out.println("After catch statement.");
    }
}
```

This program generates the following output:

```
Division by zero.
```

```
After catch statement.
```

Notice that the call to **println( )** inside the **try** block is never executed. Once an exception is thrown, program control transfers out of the **try** block into the **catch** block. Put differently, **catch** is not "called," so execution never "returns" to the **try** block from a **catch**. Thus, the line "This will not be printed." is not displayed. Once the **catch** statement has executed, program control continues with the next line in the program following the entire **try** / **catch** mechanism.
A **try** and its **catch** statement form a unit. The scope of the **catch** clause is restricted to those statements specified by the immediately preceding **try** statement. A **catch** statement cannot catch an exception thrown by another **try** statement (except in the case of nested **try** statements, described shortly). The statements that are protected by **try** must be surrounded by curly braces. (That is, they must be within a block.) You cannot use **try** on a single statement.
The goal of most well-constructed **catch** clauses should be to resolve the exceptional condition and then continue on as if the error had never happened. For example, in the next program each iteration of the **for** loop obtains two random integers. Those two integers are divided by each other, and the result is used to divide the value 12345. The final result is put into **a**. If either division operation causes a divide-by-zero error, it is caught, the value of **a** is set to zero, and the program continues.

Using Multiple catch Statements

you can associate more than one catch statement with a try. In fact, it is common to do so. However, each catch must catch a different type of exception. For example, the program shown here catches both array boundary and divide-by-zero errors:

```java
//Use multiple catch statements.
public class ExcDemo4 {
    public static void main(String args[]) {
        //Here, number is longer than denom
        int number[] = { 4, 8, 16, 32, 64, 128, 256, 512 };
        int denom[] = { 2, 0, 4, 4, 0, 8 };

        for (int i = 0; i < number.length; i++) {
            try{
                System.out.println(number[i] + " / " +
                denom[i] + " is " + number[i] / denom[i]);
            }
            catch (ArithmeticException exc) {
                //catch the exception
                System.out.println("Can't divide by Zero!");
            }
            catch (ArrayIndexOutOfBoundsException exc) {
                //Catch the exception
                System.out.println("No matching element found.");
            }
        }
    }
}
```

This program produces the following output:

```
4 / 2 is 2
Can't divide by Zero!
16 / 4 is 4
32 / 4 is 8
Can't divide by Zero!
128 / 8 is 16
No matching element found.
No matching element found.
```

As the output confirms, each catch statement responds only to its own type of exception. In general, catch expressions are checked in the order in which they occur in a program. Only a matching statement is executed. All other catch blocks are ignored.

Nested try block

One **try** block can be nested within another. An exception generated within the inner **try** block that is not caught by a **catch** associated with that **try** is propagated to the outer **try** block. For example, here the **ArrayIndexOutOfBoundsException** is not caught by the inner **catch**, but by the outer **catch**:

```java
//Use nested try block.
public class NestedTry {
    public static void main(String args[]) {
        // Here, number is longer than denom
        int number[] = { 4, 8, 16, 32, 64, 128, 256, 512 };
        int denom[] = { 2, 0, 4, 4, 0, 8 };

        try {
            for (int i = 0; i < number.length; i++) {
                try {
                    System.out.println(number[i] + " / " +
                            denom[i] + " is " + number[i] / denom[i]);
                } catch (ArithmeticException exc) {
                    // catch the exception
                    System.out.println("Can't divide by Zero!");
                }
            }
        } catch (ArrayIndexOutOfBoundsException exc) {
            // Catch the exception
            System.out.println("No matching element found.");
            System.out.println("Fatal error - program terminated.");
        }
    }
}
```

The output from the program is shown here:

```
4 / 2 is 2
Can't divide by Zero!
16 / 4 is 4
32 / 4 is 8
Can't divide by Zero!
128 / 8 is 16
No matching element found.
Fatal error - program terminated.
```

In this example, an exception that can be handled by the inner **try**—in this case, a divide-by-zero error—allows the program to continue. However, an array boundary error is caught by the outer **try**, which causes the program to terminate. Although certainly not the only reason for nested **try** statements, the preceding program makes an important point that can be generalized. Often nested **try** blocks are used to allow different categories of errors to be handled in different ways. Some types of errors are catastrophic and cannot be fixed. Some are minor and can be handled immediately. You might use an outer **try** block to catch the most severe errors, allowing inner **try** blocks to handle less serious ones.

Throwing an Exception

The preceding examples have been catching exceptions generated automatically by the JVM. However, it is possible to manually throw an exception by using the **throw** statement. Its general form is shown here:
throw *exceptOb*;
Here, *exceptOb* must be an object of an exception class derived from **Throwable**.
Here is an example that illustrates the **throw** statement by manually throwing an **ArithmeticException**:

```
// Manually throw an exception.
class ThrowDemo {
    public static void main(String[] args) {
        try {
            System.out.println("Before throw.");
            throw new ArithmeticException();
        } catch (ArithmeticException exc) {
            // catch the exception
            System.out.println("Exception caught.");
        }
        System.out.println("After try/catch block.");
    }
}
```

The output from the program is shown here:

```
Before throw.
Exception caught.
After try/catch block.
```

Notice how the **ArithmeticException** was created using **new** in the **throw** statement. Remember, **throw** throws an object. Thus, you must create an object for it to throw. That is, you can't just throw a type.

Using Finally

When you have actions you must perform at the end of a try…catch sequence, you can use a **finally block**. The code within a finally block executes regardless of whether the preceding try block identifies an exception. Usually, you use a finally block to perform cleanup tasks that must happen regardless of whether any exceptions occurred and whether any exceptions that occurred were caught. Here is the format of a try…catch sequence that uses a finally block.

```
try
    {
        // statements to try
    }
catch(Exception e)
    {
        // actions that occur if exception was thrown
    }
finally
    {
        // actions that occur whether catch block executed or not
    }
```

Sometimes you will want to define a block of code that will execute when a try/catch block is left. For example, an exception might cause an error that terminates the current method, causing its premature return. However, that method may have opened a file or a network connection that needs to be closed. Such types of circumstances are common in programming, and Java provides a convenient way to handle them: finally. Here is an example of **finally**:

```
// Use finally.
class UseFinally {
    public static void genException(int what) {
        int t;
        int[] nums = new int[2];
        System.out.println("Receiving " + what);
        try {
            switch (what) {
                case 0:
                    t = 10 / what; // generate div-by-zero error
                    break;
                case 1:
                    nums[4] = 4; // generate array index error.
                    break;
                case 2:
                    return; // return from try block
            }
        } catch (ArithmeticException exc) {
            // catch the exception
            System.out.println("Can't divide by Zero!");
            return; // return from catch
        } catch (ArrayIndexOutOfBoundsException exc) {
            // catch the exception
            System.out.println("No matching element found.");
        } finally {
            System.out.println("Leaving try.");
        }
    }
}
```

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

        for (int i = 0; i < 3; i++) {
            UseFinally.genException(i);
            System.out.println();
        }
    }
}
```

Here is the output produced by the program:

```
Receiving 0
Can't divide by Zero!
Leaving try.

Receiving 1
No matching element found.
Leaving try.

Receiving 2
Leaving try.
```

As the output shows, no matter how the try block is exited, the finally block is executed.

Using Throws

In some cases, if a method generates an exception that it does not handle, it must declare that exception in a **throws** clause. Here is the general form of a method that includes a **throws** clause:

```
ret-type methName(param-list) throws except-list {
 // body
}
```

Here, *except-list* is a comma-separated list of exceptions that the method might throw outside of itself.

You might be wondering why you did not need to specify a **throws** clause for some of the preceding examples, which threw exceptions outside of methods. The answer is that exceptions that are subclasses of **Error** or **RuntimeException** don't need to be specified in a **throws** list. Java simply assumes that a method may throw one. All other types of exceptions *do* need to be declared. Failure to do so causes a compile-time error.

Actually, you saw an example of a **throws** clause earlier in this book. As you will recall, when performing keyboard input, you needed to add the clause throws java.io.IOException to **main( )**. Now you can understand why. An input statement might generate an **IOException**, and at that time, we weren't able to handle that exception. Thus, such an exception would be thrown out of **main( )** and needed to be specified as such. Now that you know about exceptions, you can easily handle **IOException.**

Let's look at an example that handles **IOException**. It creates a method called **prompt( )**, which displays a prompting message and then reads a character from the keyboard. Since input is being performed, an **IOException** might occur. However, the **prompt( )** method does not handle **IOException** itself. Instead, it uses a **throws** clause, which means that the calling method must handle it. In this example, the calling method is **main( )**, and it deals with the error.

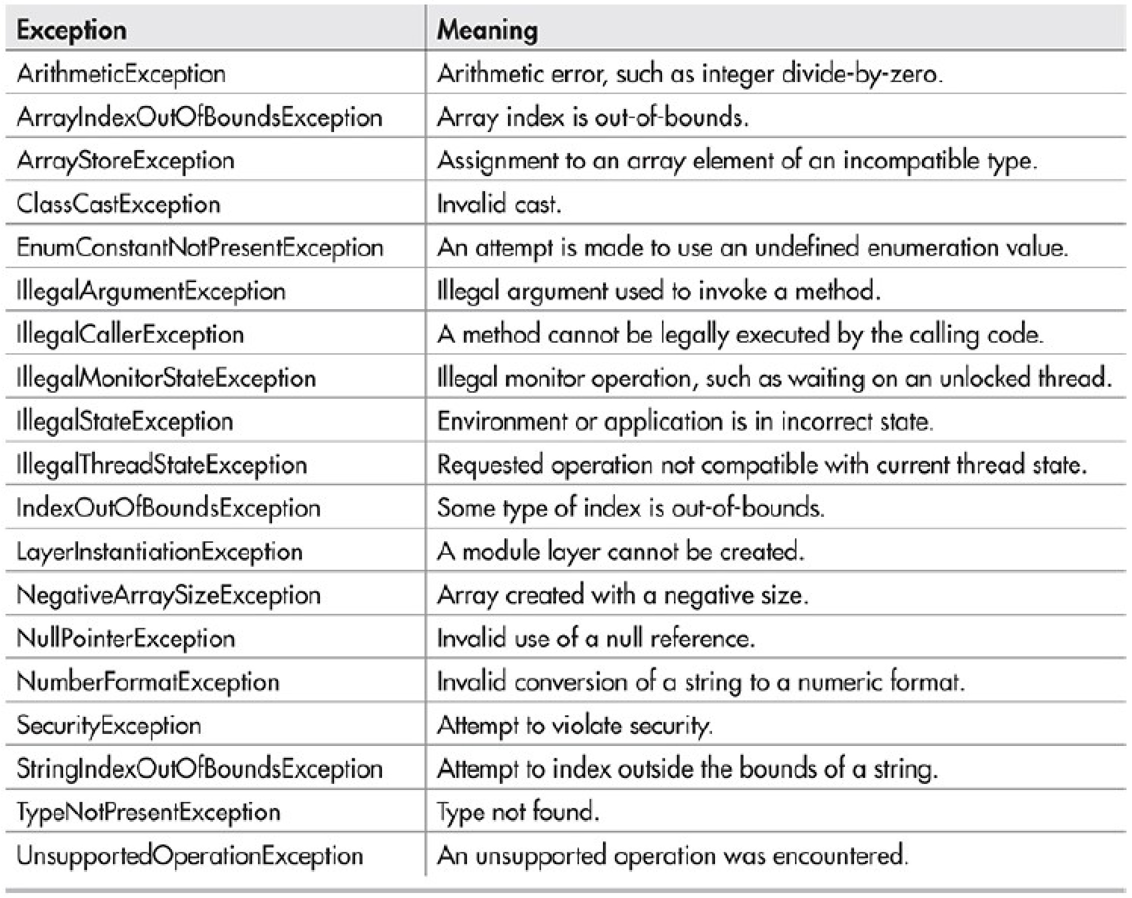
```
// Use throws.
class ThrowsDemo {
    public static char prompt(String str)
            throws java.io.IOException {
        System.out.print(str + ": ");25
        return (char) System.in.read();
    }

    public static void main(String[] args) {
        char ch;
        try {
            ch = prompt("Enter a letter");
        } catch (java.io.IOException exc) {
            System.out.println("I/O exception occurred.");
            ch = 'X';
        }
        System.out.println("You pressed " + ch);
    }
}
```

On a related point, notice that IOException is fully qualified by its package name java.io. Thus, the IOException is also contained there. It would also have been possible to import java.io and then refer to IOException directly.

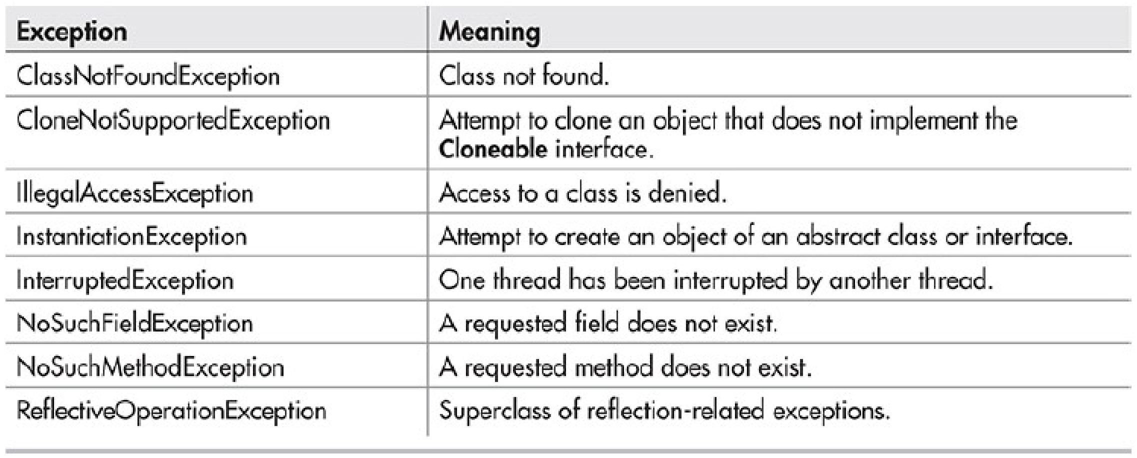
Java's Built-in Exceptions

Inside the standard package **java.lang**, Java defines several exception classes. A few have been used by the preceding examples. The most general of these exceptions are subclasses of the standard type **RuntimeException**. Since **java.lang** is implicitly imported into all Java programs, many exceptions derived from **RuntimeException** are automatically available. Furthermore, they need not be included in any method's **throws** list. In the language of Java, these are called *unchecked exceptions* because the compiler does not check to see if a method handles or throws these exceptions. The unchecked exceptions defined in **java.lang** are listed in Table 6.1. Table 6.2 lists those exceptions defined by **java.lang** that must be included in a method's **throws** list if that method can generate one of these exceptions and does not handle it itself. These are called *checked exceptions*. In addition to the exceptions in **java.lang**, Java defines several other types of exceptions that relate to other packages, such as **IOException** mentioned earlier.



Exception	Meaning
ArithmeticException	Arithmetic error, such as integer divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalCallerException	A method cannot be legally executed by the calling code.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
LayerInstantiationException	A module layer cannot be created.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBoundsException	Attempt to index outside the bounds of a string.
TypeNotPresentException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.

Table 6.1 The Unchecked Exceptions Defined in **java.lang**



Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the **Cloneable** interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.
ReflectiveOperationException	Superclass of reflection-related exceptions.

Table 6.2 The Checked Exceptions Defined in **java.lang**

Input and Output (Working with Files)

Since the beginning, you have been using parts of the Java I/O system, such as **println( ).** However, you have been doing so without much formal explanation. Because the Java I/O system is based upon a hierarchy of classes, it was not possible to present its theory and details without first discussing classes, inheritance, and exceptions. Before we begin, an important point needs to be made. The I/O classes described in this chapter support text-based console I/O and file I/O. They are not used to create graphical user interfaces (GUIs). Thus, you will not use them to create windowed applications.

Streams

Java programs perform I/O through streams. A stream is an abstraction that either produces or consumes information. A stream is linked to a physical device by the Java I/O system. All streams behave in the same manner, even if the actual physical devices to which they are linked differ. Thus, the same I/O classes and methods can be applied to different types of devices. This means that an input stream can abstract many kinds of input: from a disk file, a keyboard, or a network socket. Likewise, an output stream may refer to the console, a disk file, or a network connection. Streams are a clean way to deal with input/ output without having every part of your code understand the difference between a keyboard and a network, for example. Java implements streams within class hierarchies defined in the java.io package.

Byte Streams Classes

Byte streams are defined by using two class hierarchies. At the top are two abstract classes: **InputStream** and **OutputStream**. Each of these abstract classes has several concrete subclasses that handle the differences among various devices, such as disk files, network connections, and even memory buffers. Remember, to use the stream classes, you must import **java.io**.

Stream Class	Meaning
BufferedInputStream	Buffered input stream
BufferedOutputStream	Buffered output stream
ByteArrayInputStream	Input stream that reads from a byte array
ByteArrayOutputStream	Output stream that writes to a byte array
DataInputStream	An input stream that contains methods for reading the Java standard data types
DataOutputStream	An output stream that contains methods for writing the Java standard data types
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that writes to a file
FilterInputStream	Implements **InputStream**
FilterOutputStream	Implements **OutputStream**
InputStream	Abstract class that describes stream input
ObjectInputStream	Input stream for objects
ObjectOutputStream	Output stream for objects
OutputStream	Abstract class that describes stream output
PipedInputStream	Input pipe
PipedOutputStream	Output pipe
PrintStream	Output stream that contains **print( )** and **println( )**
PushbackInputStream	Input stream that supports one-byte "unget," which returns a byte to the input stream
SequenceInputStream	Input stream that is a combination of two or more input streams that will be read sequentially, one after the other

Table 6.3 The Byte Stream Classes in **java.io**

The abstract classes **InputStream** and **OutputStream** define several key methods that the other stream classes implement. Two of the most important are **read( )** and **write( )**, which, respectively, read and write bytes of data. Each has a form that is abstract and must be overridden by derived stream classes. Character Stream classes
Character streams are defined by using two class hierarchies. At the top are two abstract classes: **Reader** and **Writer**. These abstract classes handle Unicode character streams. Java has several concrete subclasses of each of these.

Character Stream Classes

Character streams are defined by using two class hierarchies. At the top are two abstract classes: Reader and Writer. These abstract classes handle Unicode character streams. Java has several concrete subclasses of each of these. The character stream classes in java.io are shown in Table 6.4.

Stream Class	Meaning
BufferedReader	Buffered input character stream
BufferedWriter	Buffered output character stream
CharArrayReader	Input stream that reads from a character array
CharArrayWriter	Output stream that writes to a character array
FileReader	Input stream that reads from a file
FileWriter	Output stream that writes to a file
FilterReader	Filtered reader
FilterWriter	Filtered writer
InputStreamReader	Input stream that translates bytes to characters
LineNumberReader	Input stream that counts lines
OutputStreamWriter	Output stream that translates characters to bytes
PipedReader	Input pipe
PipedWriter	Output pipe
PrintWriter	Output stream that contains **print( )** and **println( )**
PushbackReader	Input stream that allows characters to be returned to the input stream
Reader	Abstract class that describes character stream input
StringReader	Input stream that reads from a string
StringWriter	Output stream that writes to a string
Writer	Abstract class that describes character stream output

Table 6.4 The Character Stream I/O Classes in **java.io**

The abstract classes **Reader** and **Writer** define several key methods that the other stream classes implement. Two of the most important methods are **read( )** and **write( )**, which read and write characters of data, respectively. Each has a form that is abstract and must be overridden by derived stream classes.

Reading and Writing files using Byte Streams

Java provides a number of classes and methods that allow you to read and write files.
Of course, the most common types of files are disk files. In Java, all files are byte-oriented, and Java provides methods to read and write bytes from and to a file. Thus, reading and writing files using byte streams is very common. However, Java allows you to wrap a byte-oriented file stream within a characterbased object, which is shown later in this chapter.
To create a byte stream linked to a file, use **FileInputStream** or **FileOutputStream**. To open a file, simply create an object of one of these classes, specifying the name of the file as an argument to the constructor. Once the file is open, you can read from or write to it.

Inputting from and outputting to a File
A file is opened for input by creating a **FileInputStream** object. Here is a commonly used constructor:
FileInputStream(String *fileName*) throws FileNotFoundException

Here, *fileName* specifies the name of the file you want to open. If the file does not exist, then **FileNotFoundException** is thrown. **FileNotFoundException** is a subclass of **IOException**.
To read from a file, you can use **read( ).** The version that we will use is shown here:
int read( ) throws IOException
Each time it is called, **read( )** reads a single byte from the file and returns it as an integer value. It returns –1 when the end of the file is encountered. It throws an **IOException** when an error occurs. Thus, this version of **read( )** is the same as the one used to read from the console. When you are done with a file, you must close it by calling **close(**). Its general form is shown here:
void close( ) throws IOException
Closing a file releases the system resources allocated to the file, allowing them to be used by another file. Failure to close a file can result in "memory leaks" because of unused resources remaining allocated.

```
import java.io.*;

class FileStreamsTest {
    public static void main(String[] args) {
        try {
            //Text with sample content
            //This file have to exist
            File inputFile = new File("inputText.txt");
            //File to write the content to
            //This file can be created at run-time
            File outputFile = new File("outputText.txt");

            //Using FileInputStream to read the data of the inputText file
            FileInputStream fis = new FileInputStream(inputFile);
            //Using FileOutputStream to write to outputText file
            FileOutputStream fos = new FileOutputStream(outputFile);
            int c;

            //Read the data from the inputText file while it has not reached the end of
file
            //Write whatever is read from inputText to outputText file.
            while ((c = fis.read()) != -1) {
                fos.write(c);

            }
            //Close the streams
            fis.close();
            fos.close();
        } catch (FileNotFoundException e) {
            System.err.println("FileStreamsTest: " + e);
        } catch (IOException e) {
            System.err.println("FileStreamsTest: " + e);
        }
    }
}
```

Self Evaluation

1. Why would you want to catch exceptions?
2. Why would you decide to throw an exception?
3. When should you use exception handling in a program?
4. What is the difference between exception and error in Java?
5. Name the different types of exceptions in Java
6. What are the types of I/O streams?
7. What is common and how do the following streams differ: InputStream, OutputStream, Reader, Writer?
8. What is common and how do the following streams differ: InputStream, OutputStream, Reader, Writer?

TOPIC 7

7. MULTITHREADPROGRAMMING AND JAVA COLLECTION FRAMEWORK

LEARNING OUTCOMES

After studying this topic, you should be able to:

- Understanding Multithreading.
  - Multi Threaded programming.
  - Synchronization.
  - Collection Framework, Java.Util Package.

7.1 MULTITHREADED PROGRAMMING

The Java Thread Model

The Java run-time system depends on threads for many things, and all the class libraries are designed with multithreading in mind. In fact, Java uses threads to enable the entire environment to be asynchronous. This helps reduce inefficiency by preventing the waste of CPU cycles. The value of a multithreaded environment is best understood in contrast to its counterpart. Single-threaded systems use an approach called an *event loop* with *polling.*In this model, a single thread of control runs in an infinite loop, polling a single event queue to decide what to do next. Once this polling mechanism returns with, say, a signal that a network file is ready to be read, then the event loop dispatches control to the appropriate event handler. Until this event handler returns, nothing else can happen in the system. This wastes CPU time. It can also result in one part of a program dominating the system and preventing any other events from being processed. In general, in a singled-threaded environment, when a thread *blocks* (that is, suspends execution) because it is waiting for some resource, the entire program stops running. The benefit of Java's multithreading is that the main loop/polling mechanism is eliminated. One thread can pause without stopping other parts of your program. For example, the idle time created when a thread reads data from a network or waits for user input can be utilized elsewhere. Multithreading allows animation loops to sleepfor a second between each frame without causing the whole system to pause. When a thread blocks in a Java program, only the single thread that is blocked pauses. All other threads continue to run. Threads exist in several states. A thread can be *running*. It can be *ready to run* as soon as it gets CPU time. A running thread can be *suspended,* which temporarily suspends its activity. A suspended thread can then be *resumed,* allowing it to pick up where it left off. A thread can be *blocked* when waiting for a resource. At any time, a thread can be terminated, which halts its execution immediately. Once terminated, a thread cannot be resumed.

Thread Priorities
Java assigns to each thread a priority that determines how that thread should be treated with respect to the others. Thread priorities are integers that specify the relative priority of one thread to another. As an absolute value, a priority is meaningless; a higher-priority thread doesn't run any faster than a lower-priority thread if it is the only thread running. Instead, a thread's priority is used to decide when to switch from one running thread to the next. This is called a *context switch.* The rules that determine when a context switch takes place are simple:

■ *A thread can voluntarily relinquish control.* This is done by explicitly yielding, sleeping, or blocking on pending I/O. In this scenario, all other threads are examined, and the highest-priority thread that is ready to run is given the CPU.

A thread can be preempted by a higher-priority thread. In this case, a lower-priority thread that does not yield the processor is simply preempted—no matter what it is doing—by a higher-priority thread. Basically, as soon as a higher-priority thread wants to run, it does. This is called *preemptive multitasking.*

In cases where two threads with the same priority are competing for CPU cycles, the situation is a bit complicated. For operating systems such as Windows 98, threads of equal priority are time-sliced automatically in round-robin fashion. For other types of operating systems, threads of equal priority must voluntarily yield control to their peers. If they don't, the other threads will not run.

7.2 SYNCHRONIZATION

Because multithreading introduces an asynchronous behavior to your programs, there must be a way for you to enforce synchronicity when you need it. For example, if you want two threads to communicate and share a complicated data structure, such as a linked list, you need some way to ensure that they don't conflict with each other. That is, you must prevent one thread from writing data while another thread is in the middle of reading it. For this purpose, Java implements an elegant twist on an age-old model of interprocess synchronization: the *monitor.* The monitor is a control mechanism first defined by C.A.R. Hoare. You can think of a monitor as a very small box that can hold only one thread. Once a thread enters a monitor, all other threads must wait until that thread exits the monitor. In this way, a monitor can be used to protect a shared asset from being manipulated by more than one thread at a time.

Most multithreaded systems expose monitors as objects that your program must explicitly acquire and manipulate. Java provides a cleaner solution. There is no class "Monitor"; instead, each object has its own implicit monitor that is automatically entered when one of the object's synchronized methods is called. Once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object. This enables you to write very clear and concise multithreaded code, because synchronization support is built in to the language.

Messaging

After you divide your program into separate threads, you need to define how they will communicate with each other. When programming with most other languages, you must depend on the operating system to establish communication between threads. This, of course, adds overhead. By contrast, Java provides a clean, low-cost way for two or more threads to talk to each other, via calls to predefined methods that all objects.

Synchronization
When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called *synchronization.* As you will see, Java provides unique, language-level support for it.

Key to synchronization is the concept of the monitor (also called a *semaphore*). A *monitor* is an object that is used as a mutually exclusive lock, or *mutex.* Only one thread can *own* a monitor at a given time. When a thread acquires a lock, it is said to have *entered* the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread *exits* the monitor. These other threads are said to be *waiting* for the monitor. A thread that owns a monitor can reenter the same monitor if it so desires. If you have worked with synchronization when using other languages, such as C or C++, you know that it can be a bit tricky to use. This is because most languages do not, themselves, support synchronization. Instead, to synchronize threads, your programs need to utilize operating system primitives. Fortunately, because Java implements synchronization through language elements, most of the complexity associated with synchronization has been eliminated. You can synchronize your code in either of two ways. Both involve the use of the **synchronized** keyword, and both are examined here.

Using Synchronized Methods
Synchronization is easy in Java, because all objects have their own implicit monitor associated with them. To enter an object's monitor, just call a method that has been modified with the **synchronized** keyword. While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait. To exit the monitor and relinquish control of the object to the next waiting thread, the owner of the monitor simply returns from the synchronized method.

To understand the need for synchronization, let's begin with a simple example that does not use it—but should. The following program has three simple classes. The first one, **Callme**, has a single method named **call( )**. The **call( )** method takes a **String** parameter called **msg**. This method tries to print the **msg** string inside of square brackets. The interesting thing to notice is that after **call( )** prints the opening bracket and the **msg** string, it calls **Thread.sleep(1000)**, which pauses the current thread for one second.
The constructor of the next class, **Caller**, takes a reference to an instance of the **Callme** class and a **String**, which are stored in **target** and **msg**, respectively. The constructor also creates a new thread that will call this object's **run( )** method. The thread is started immediately. The **run( )** method of **Caller** calls the **call( )** method on the **target** instance of **Callme**, passing in the **msg** string. Finally, the **Synch** class starts by creating a single

instance of **Callme**, and three instances of **Caller**, each with a unique message string. The same instance of **Callme** is passed to each **Caller**.

```
// This program is not
synchronized. class Callme {
    void call(String msg) {
System.out.print("[" + msg);
try {
Thread.sleep(1000);
} catch(InterruptedException e) {
System.out.println("Interrupted");
}
System.out.println("]");
}
}

class Caller implements Runnable {
String msg;
Callme target;
Thread t;
public Caller(Callme targ, String s) {
target = targ;
msg = s;
t = new Thread(this);
t.start();
}
public void run() {
target.call(msg);
}

}

class Synch {
public static void main(String args[]) {
Callme target = new Callme();
Caller ob1 = new Caller(target, "Hello");
Caller ob2 = new Caller(target, "Synchronized");
Caller ob3 = new Caller(target, "World");
// wait for threads to
end try {
ob1.t.join();
ob2.t.join();
ob3.t.join();
}
catch(InterruptedException e) {
System.out.println("Interrupted");
}
}
}
```

Here is the output produced by this program:

```
Hello[Synchronized[World]
]
]
```

As you can see, by calling **sleep( )**, the **call( )** method allows execution to switch to another thread. This results in the mixed-up output of the three message strings. In this program, nothing exists to stop all three threads from calling the same method, on the same object, at the same time. This is known as a *race condition,* because the three threads are racing each other to complete the method. This example used **sleep( )** to make the effects repeatable and obvious. In most situations, a race condition is more subtle and less predictable, because you can't be sure when the context switch will occur. This can cause a program to run right one time and wrong the next. To fix the preceding program, you must *serialize* access to **call( )**. That is, you must restrict its access to only one thread at a time. To do this, you simply need to precede **call( )**'s definition with the keyword **synchronized**, as shown here:

```
class Callme {
synchronized void call(String msg) {
..
```

This prevents other threads from entering **call( )** while another thread is using it. After **synchronized** has been added to **call( )**, the output of the program is as follows:

```
[Hello]
[Synchronized]
[World]
```

Any time that you have a method, or group of methods, that manipulates the internal state of an object in a multithreaded situation, you should use the **synchronized** keyword to guard the state from race conditions. Remember, once a thread enters any synchronized method on an instance, no other thread can enter any other synchronized method on the same instance. However, nonsynchronized methods on that instance will continue to be callable.

The synchronized Statement

While creating **synchronized** methods within classes that you create is an easy and effective means of achieving synchronization, it will not work in all cases. To understand why, consider the following. Imagine that you want to synchronize access to objects of
a class that was not designed for multithreaded access. That is, the class does not use **synchronized** methods. Further, this class was not created by you, but by a third party, and you do not have access to the source code. Thus, you can't add **synchronized** to the appropriate methods within the class. How can access to an object of this class be synchronized? Fortunately, the solution to this problem is quite easy: You simply put calls to the methods defined by this class inside a **synchronized** block.

This is the general form of the **synchronized** statement:

```
synchronized(object) {
// statements to be synchronized
}
```

Here, *object* is a reference to the object being synchronized. A synchronized block ensures that a call to a method that is a member of *object* occurs only after the current thread has successfully entered *object*'s monitor.

Here is an alternative version of the preceding example, using a synchronized block within the **run( )** method:

```
// This program uses a synchronized
block. class Callme {
void call(String msg) {
System.out.print("[" +
msg); try {
Thread.sleep(1000);
} catch (InterruptedException e) {
System.out.println("Interrupted");
}
System.out.println("]");
}
}
class Caller implements Runnable
{ String msg;
Callme target;
Thread t;
public Caller(Callme targ, String s)
{ target = targ;
msg = s;
t = new Thread(this);
t.start();
}
// synchronize calls to call()
public void run() {
synchronized(target) { // synchronized block
target.call(msg);
}
}
}

class Synch1 {
public static void main(String args[]) {
Callme target = new Callme();
Caller ob1 = new Caller(target, "Hello");
Caller ob2 = new Caller(target, "Synchronized");
Caller ob3 = new Caller(target, "World");
// wait for threads to
end try {
ob1.t.join();
ob2.t.join();
ob3.t.join();
} catch(InterruptedException e) {
System.out.println("Interrupted");
}
}
}
```

Here, the **call( )** method is not modified by **synchronized**. Instead, the **synchronized** statement is used inside **Caller**'s **run( )** method. This causes the same correct output as the preceding example, because each thread waits for the prior one to finish before proceeding.

7.3 THE COLLECTION FRAME WORK AND JAVA.UTIL PACKAGE

The java.util package contains one of Java's most powerful subsystems: collections. Collections were added by the initial release of Java 2, and enhanced by Java 2, version 1.4. A collection is a group of objects. The addition of collections caused fundamental alterations in the structure and architecture of many elements in java.util. It also expanded the domain of tasks to which the package can be applied. Collections are a state-of-the-art technology that merits close attention by all Java programmers. In addition to collections, java.util contains a wide assortment of classes and interfaces that support a broad range of functionality. These classes and interfaces are used throughout the core Java packages and, of course, are also available for use in programs that you write. Their applications include generating pseudorandom numbers, manipulating date and time, observing events, manipulating sets of bits, and tokenizing strings. Because of its many features, java.util is one of Java's most widely used packages. The java.util classes are listed here.

- AbstractCollection (Java 2) EventObject PropertyResourceBundle
- AbstractList (Java 2) GregorianCalendar Random
- AbstractMap (Java 2) HashMap (Java 2) ResourceBundle
- AbstractSequentialList (Java 2) HashSet (Java 2) SimpleTimeZone
- AbstractSet (Java 2) Hashtable Stack
- ArrayList (Java 2) IdentityHashMap (Java 2, v1.4) StringTokenizer
- Arrays (Java 2) LinkedHashMap (Java 2, v1.4) Timer (Java 2, v1.3)
- BitSet LinkedHashSet (Java 2, v1.4) TimerTask (Java 2, v1.3)
- Calendar LinkedList (Java 2) TimeZone
- Collections (Java 2) ListResourceBundle TreeMap (Java 2)
- Currency (Java 2, v1.4) Locale TreeSet (Java 2)
- Date Observable Vector
- Dictionary Properties WeakHashMap (Java 2)
- EventListenerProxy (Java 2, v1.4) PropertyPermission (Java 2)
- java.util defines the following interfaces. Notice that most were added by Java

- Collection (Java 2) List (Java 2) RandomAccess (Java 2, v1.4)
- Comparator (Java 2) ListIterator (Java 2) Set (Java 2)
- Enumeration Map (Java 2) SortedMap (Java 2)
- EventListener Map.Entry (Java 2) SortedSet (Java 2)
- Iterator (Java 2) Observer

The ResourceBundle, ListResourceBundle, and PropertyResourceBundle classes aid in the internationalization of large programs with many locale-specific resources. These classes are not examined here. PropertyPermission, which allows you to grant a read/write permission to a system property, is also beyond the scope of this book. EventObject, EventListener, and EventListenerProxy.

Theremaining classes and interfaces are examined in detail. Because java.util is quite large, its description is broken into two chapters. This chapter examines those members of java.util that relate to collections of objects. Discuss the other classes and interfaces.

LEARNER ACTIVITY 6

- Describe Multi Threaded Programming.
- Explain Synchronization.
- Describe Collection Framework and Java.Util package.