



OBJECT ORIENTED PROGRAMMING

COURSE INSTRUCTOR: R. SHATHANAA



COURSE OVERVIEW



COURSE OBJECTIVES

The objective of this course is to enable the students to design and analyze applications using Object Oriented concepts. This course is offered to the learners who are familiar with at least one programming language (preferably C) and the basic data structures. This course involves a lab session in which students will learn to code the solutions for the challenging problems using object oriented concepts.

SYLLABUS

L-T-P-C: 2-1-1-4

Unit – 1 [7 Hours]: Introduction: Basics of OOP, Structure of Java Programs - Programming Environments, Program Control, Basics of Computation - Control Flow and Branching - Defining Classes, Objects and Methods - Benefits of Object-Oriented Programming Methodologies

Unit – 2 [10 Hours]: Constructors and Overloading Concepts: Default and Parameterized Constructors - Method Overloading - Packages, Access Specifiers, Composition - Accessor / Mutator(getter/setter) Methods - Keywords, Finals

Unit – 3 [7 Hours]: Access Control Modifiers: Nested Classes - Interfaces and Inner Classes - Abstract Classes

Unit – 4 [10 Hours]: Object Oriented Concepts: Inheritance - Polymorphism – Error Handling - Exception Handling, Different Streams - File I/O, and Networking

Unit – 5 [7 Hours]: Collections: Collections and Iterators - Dynamic Data Structures and Generics - Recursion

Unit – 6 [7 Hours]: Multithreading and Database Connectivity: Threads, Multi-threading Applications - Java Database Connectivity with MySQL



EVALUATION COMPONENTS

- TBD

TEXT BOOKS

- Robert Endre Tarjan, Data Structures and network algorithms, Society for Industrial and Applied Mathematics Philadelphia, PA, USA, 1983, ISBN:0-89871-187-8
- Paul Deitel and Harvey Deitel. 2011. Java how to Program (9th ed.). Prentice Hall Press, Upper Saddle River, NJ, USA
- David J. Eck. 2009. Programming: Introduction to Programming Using JAVA. CreateSpace, Paramount, CA
- Any decent material that clearly illustrates OOP Concepts
- State-of-the-art approaches: Research Papers / Seminar Papers / Case Studies

REFERENCE BOOKS (JAVA)

- Paul Deitel and Harvey Deitel. 2011. Java how to Program (11th ed.). Prentice Hall Press, Upper Saddle River, NJ, USA
- Cay S. Horstmann and Gary Cornell, Core Java by (11th ed.)
- Herbert Schildt, Java: The Complete Reference, Eleventh Edition

PREREQUISITE

- C Programming

CLASS POLICY

- Take notes
- Ask questions
- Be interactive
- Google is your friend (Although it gets benefits in return)

COURSE POLICY

- No Cheating !

MATERIALS

- Will be uploaded in Google drive



A QUICK POLL



Questions?

Object Oriented Programming Course

Monsoon 2021

Programming Paradigms

A high level overview

Dictionary

Search for a word



par·a·digm

/'perə,dīm/

See definitions in:

All

Philosophy

Language

noun

noun: **paradigm**; plural noun: **paradigms**

1. a typical example or pattern of something; a model.
"there is a new paradigm for public art in this country"

Similar:

model

pattern

example

standard

prototype

archetype

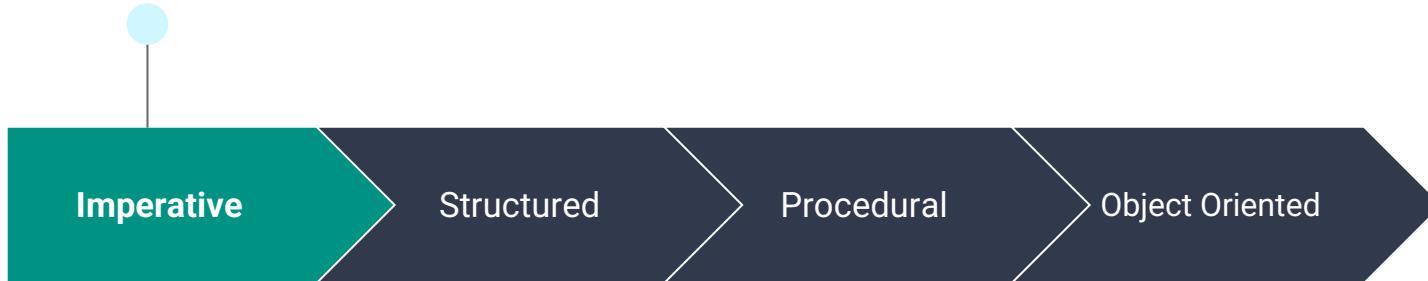
ideal

Programming paradigm

A programming paradigm is a style, or “way,” of programming.

A “way” to think about the how we write and organize programs

Programming using sequence of Imperative Statements



What Is the Imperative Mood? (with Examples)

The imperative mood is a verb form that gives a command. For example:

- Empty the bin, John.

(This is a verb **in** the imperative mood.)

- John **empties** the bin.

(This verb is not in the **imperative** mood. It is in the indicative mood.)

Commands can include orders, requests, advice, instructions, and warnings.

Imperative Programming

Writing programs in the form of imperative statements (aka commands)

“Do this” and “Do that”

Imperative Statement in English

“ Go to a bakery ”

“ Buy a cookie ”

“ Eat the cookie ”

“ Drink some water ”

Declarative Statements in English

“ There is a bakery in 2nd Street ”

Imperative Programming

You may think of this as the “default” paradigm

This is how the machine code works

“Move val to register”

“Load val from register”

“Add values”

“Jump to location”

EXAMPLE OF ASSEMBLY LANGUAGE

High level code

$D = A * B + A * C;$

Assume R1 contains the
base address of 100

Address

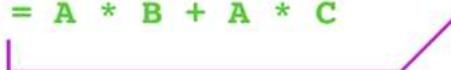
100	5	A
104	12	B
108	8	C
112		D

Assembly code

```
LD  R2, 0(R1) ; load A
LD  R3, 4(R1) ; load B
LD  R4, 8(R1) ; load C
MUL R5, R2, R3 ; A * B
MUL R6, R2, R4 ; A * C
ADD R5, R5, R6 ; A * B + A * C
SD   12(R1), R5 ; D = A * B + A * C
```

Address

100	5	A
104	12	B
108	8	C
112	100	D



Problems

Verbose

More error prone

Unnecessary/Accidental Redundancy

Difficult to organize

Difficult to form a mental model

Difficult to debug

Hence, difficult to work in large teams

Problems

Verbose

More error prone

Unnecessary/Accidental Redundancy

Difficult to organize

Difficult to form a mental model

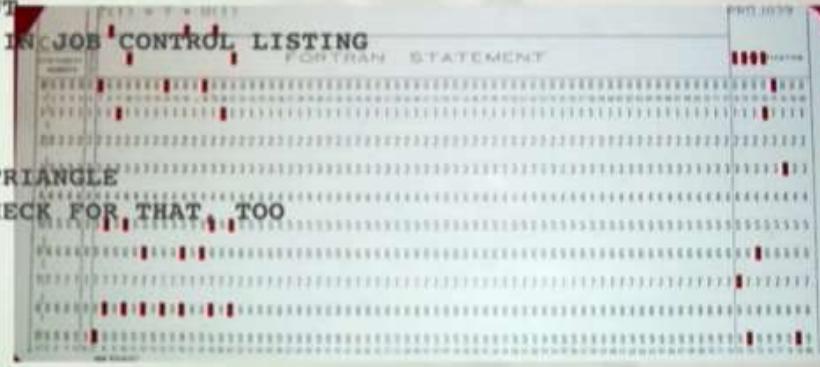
Difficult to debug

Hence, difficult to work in large teams

1953 FORTRAN

```
C AREA OF A TRIANGLE WITH A STANDARD SQUARE ROOT FUNCTION
C INPUT - TAPE READER UNIT 5, INTEGER INPUT
C OUTPUT - LINE PRINTER UNIT 6, REAL OUTPUT
C INPUT ERROR DISPLAY ERROR OUTPUT CODE 1 IN JOB CONTROL LISTING
      READ INPUT TAPE 5, 501, IA, IB, IC
 501 FORMAT (3I5)
C IA, IB, AND IC MAY NOT BE NEGATIVE
C FURTHERMORE, THE SUM OF TWO SIDES OF A TRIANGLE
C IS GREATER THAN THE THIRD SIDE, SO WE CHECK FOR THAT, TOO
      IF (IA) 777, 777, 701
701 IF (IB) 777, 777, 702
702 IF (IC) 777, 777, 703
703 IF (IA+IB-IC) 777,777,704
704 IF (IA+IC-IB) 777,777,705
705 IF (IB+IC-IA) 777,777,799
    777 STOP 1
C USING HERON'S FORMULA WE CALCULATE THE
C AREA OF THE TRIANGLE
    799 S = FLOATF (IA + IB + IC) / 2.0
      AREA = SQRT( S * (S - FLOATF(IA)) * (S - FLOATF(IB)) *
+          (S - FLOATF(IC)))
      WRITE OUTPUT TAPE 6, 601, IA, IB, IC, AREA
 601 FORMAT (4H A= ,I5,5H B= ,I5,5H C= ,I5,8H AREA= ,F10.2,
+           13H SQUARE UNITS)
      STOP
END
```

Designed by: John Backus



Problems

Verbose

More error prone

Unnecessary/Accidental Redundancy

Difficult to organize

Difficult to form a mental model

Difficult to debug

Hence, difficult to work in large teams

Problems



Verbose

More error prone

Unnecessary/Accidental Redundancy

Difficult to organize

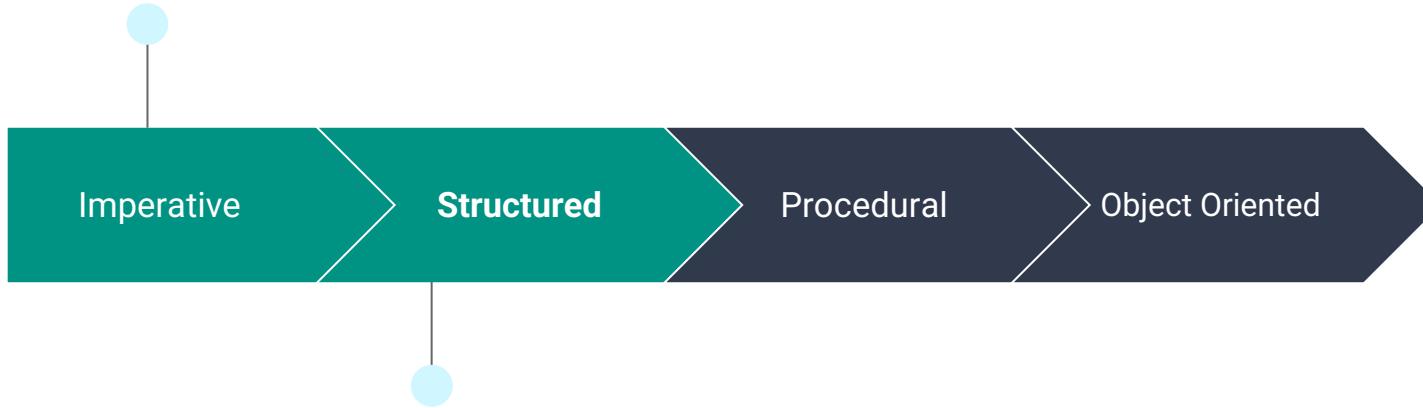
Difficult to form a mental model

Difficult to debug

Difficult to work in large teams

Dijkstra, E. W. (1968). Letters to the editor: **go to** statement considered harmful. *Communications of the ACM*, 11(3), 147-148.

Programming using sequence of
Imperative Statements



**Let's do away with harmful goto
and introduce control structures
IF, FOR, etc.**

Structured program theorem

- The structured program theorem, also called the Böhm–Jacopini theorem is a result in programming language theory. It states that a class of control-flow graphs (historically called flowcharts in this context) can compute any computable function if it combines subprograms in only three specific ways (control structures). These are
 - Executing one subprogram, and then another subprogram (sequence)
 - Executing one of two subprograms according to the value of a boolean expression (selection)
 - Repeatedly executing a subprogram as long as a boolean expression is true (iteration)

Structured Programming

Writing programs using nested-control structures without any **harmful goto** statement

Two types of control structures

Branching

IF/ELSE, SWITCH-CASE

Loops

FOR, WHILE, DO-WHILE

Problems

Verbose

More error prone

Unnecessary/Accidental Redundancy

Difficult to organize

Difficult to form a mental model

Difficult to debug

Hence, difficult to work in large teams

Problems

Verbose

More error prone

Unnecessary/Accidental Redundancy

Difficult to organize

Difficult to form a mental model

Difficult to debug

Hence, difficult to work in large teams

Note: This did not eliminate all errors. But got rid of plenty of errors and confusion which were primarily caused by goto statements.

Problems

Verbose

More error prone

Unnecessary/Accidental Redundancy

Difficult to organize

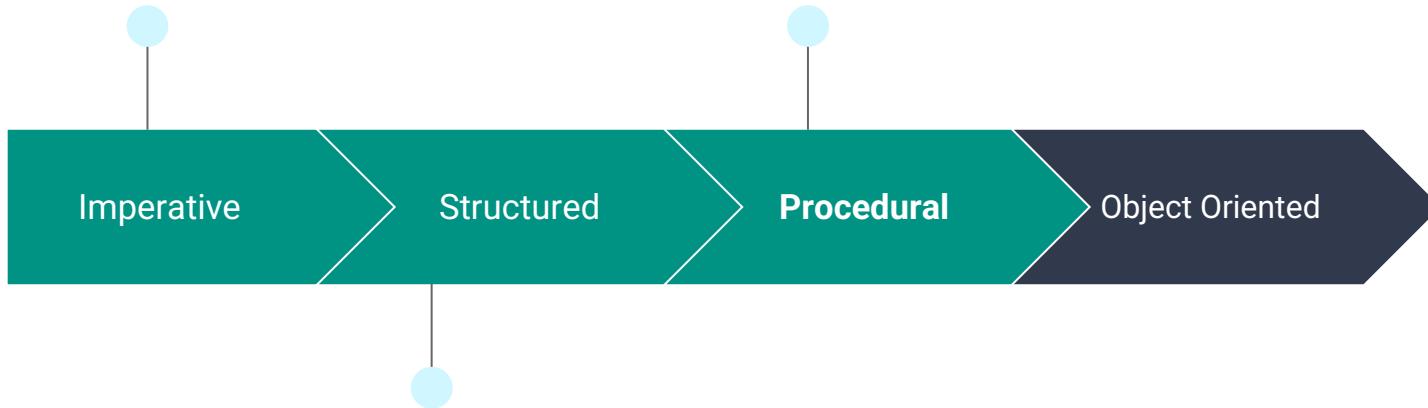
Difficult to form a mental model

Difficult to debug

Hence, difficult to work in large teams

Programming using sequence of
Imperative Statements

**Organize code into Procedures - to
avoid redundancy and to improve
organization/understanding**



Let's do away with harmful
goto and introduce control
structures IF, FOR, etc.

START

Do A

Do B

Do C

Do D

Do A

Do B

Do E

Do F

Do C

Do D

Do G

End

An example of an Imperative Program

- Sequence of Imperative Statements

START

Do A

Do B

Do C

Do D

Do A

Do B

Do E

Do F

Do C

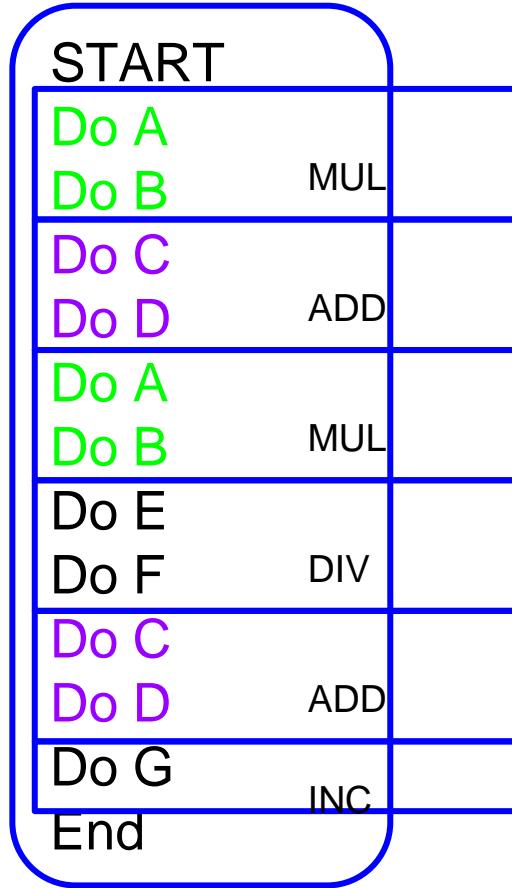
Do D

Do G

End

Redundant Code

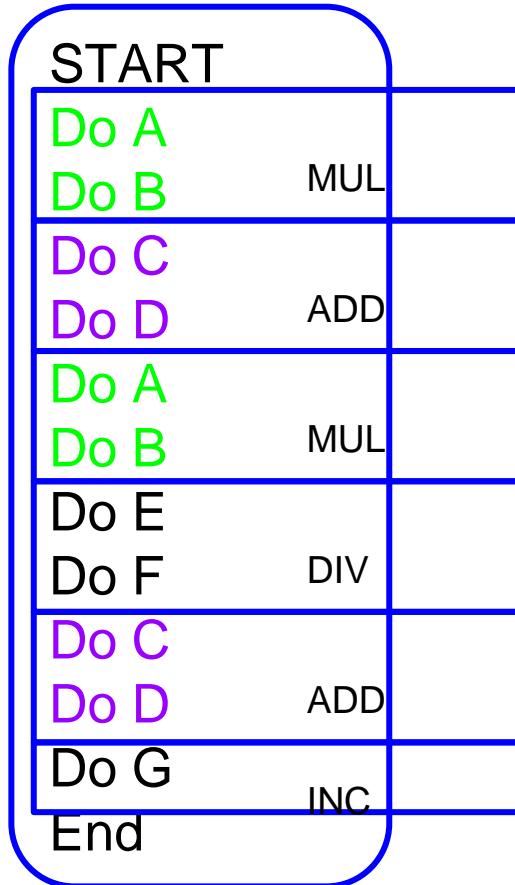
- Pattern of Code that repeats again and again



Related Code

- Sequence of Code which represents a human understandable unit of operation (a *task*)

- A Procedural Approach



Do A
Do B
MUL

Do C
Do D
ADD

Do E
Do F
DIV

Do G
INC

START
MUL
ADD
MUL
DIV
ADD
INC
End
MAIN

C Language

Imperative

- *Sequence of Actions*

Structured

- *Nested Control Structures*

Procedural

- *Organizes code as procedures*

Problems

Verbose

More error prone

Unnecessary/Accidental Redundancy

Difficult to organize

Difficult to form a mental model

Difficult to debug

Hence, difficult to work in large teams



Object Oriented Programming

Course Instructor: Dr. R. Shathanaa

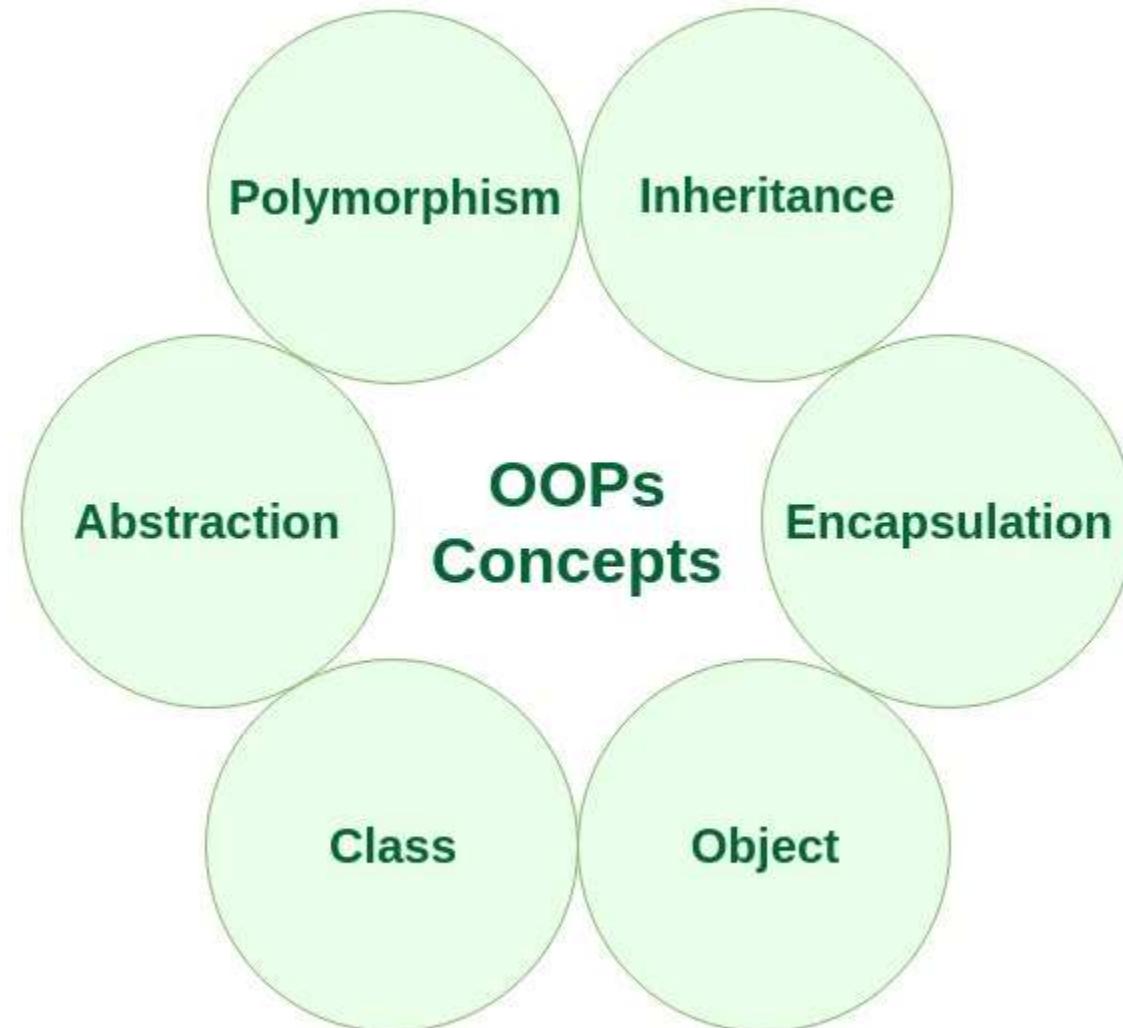


What is OOP?

- ▶ *Object-oriented programming (OOP)* is a programming paradigm that allows you to package together data states and functionality to modify those data states, while keeping the details hidden away. As a result, code with OOP design is flexible, modular, and abstract. This makes it particularly useful when you create larger programs.



- ▶ Let's consider a physical object: a light bulb. A light (usually) has two possible states: on and off. It also has functionality that allows you to change its state: you can turn it on and you turn it off. Thankfully, you don't need to know electrical engineering to use the light! You only need to know how to interact with it.





Why So Many Languages?

- ▶ Bring the language “closer” to the problem.
- ▶ Many advanced languages are typically focused on specialized domains (e.g., relational databases).
- ▶ We want a language that is general purpose, yet can easily be “tailored” to any domain.



Object-Oriented Languages

- ▶ Smalltalk, C++, Java, etc...
- ▶ How different from procedural languages?
 - Not different at all: Every (reasonable) language is “Turing complete”
 - Very different: Make expression easier, less error-prone



O-O Languages

- ▶ Everything is an object.
- ▶ A program is a bunch of objects telling each other what to do, by sending messages.
- ▶ Each object has its own memory, and is made up of other objects.
- ▶ Every object has a type (class).
- ▶ All objects of the same type can receive the same messages.



Introduction to Object Technology

- ▶ Objects, or more precisely, the classes objects come from, are essentially reusable software components.
 - There are date objects, time objects, audio objects, video objects, automobile objects, people objects, etc.
 - Almost any noun can be reasonably represented as a software object in terms of attributes (e.g., name, color and size) and behaviors (e.g., calculating, moving and communicating).



Introduction to Object Technology (Cont.)

► The Automobile as an Object

- Let's begin with a simple analogy.
- Suppose you want to *drive a car and make it go faster by pressing its accelerator pedal*.
- Before you can drive a car, someone has to *design it*.
- A car typically begins as engineering drawings, similar to the *blueprints that describe the design of a house*.
- Drawings include the design for an accelerator pedal.
- Pedal *hides from the driver the complex mechanisms that actually make the car go faster, just as the brake pedal hides the mechanisms that slow the car, and the steering wheel "hides" the mechanisms that turn the car*.



Introduction to Object Technology (Cont.)

- Enables people with little or no knowledge of how engines, braking and steering mechanisms work to drive a car easily.
- Before you can drive a car, it must be *built* from the engineering drawings that describe it.
- A completed car has an *actual* accelerator pedal to make the car go faster, but even that's not enough—the car won't accelerate on its own (hopefully!), so the driver must press the pedal to accelerate the car.



Introduction to Object Technology

(Cont.)

► Methods and Classes

- Performing a task in a program requires a **method**.
- The method houses the program statements that actually perform its tasks.
- Hides these statements from its user, just as the accelerator pedal of a car hides from the driver the mechanisms of making the car go faster.
- In Java, we create a program unit called a **class** to house the set of methods that perform the class' s tasks.
- A class is similar in concept to a car' s engineering drawings, which house the design of an accelerator pedal, steering wheel, and so on.



Introduction to Object Technology (Cont.)

▶ Instantiation

- Just as someone has to build a car from its engineering drawings before you can actually drive a car, you must build an object of a class before a program can perform the tasks that the class's methods define.
- An object is then referred to as an **instance** of its class.



Introduction to Object Technology (Cont.)

- ▶ Reuse
 - Just as a car's engineering drawings can be *reused* many times to build many cars, you can reuse a class many times to build many objects.
 - Reuse of existing classes when building new classes and programs saves time and effort.
 - Reuse also helps you build more reliable and effective systems, because existing classes and components often have gone through extensive *testing, debugging and performance tuning*.
 - Just as the notion of *interchangeable parts* was crucial to the Industrial Revolution, reusable classes are crucial to the software revolution that has been spurred by object technology.



Software Engineering Observation 1.1

Use a building-block approach to creating your programs. Avoid reinventing the wheel—use existing pieces wherever possible. This software reuse is a key benefit of object-oriented programming.



Introduction to Object Technology (Cont.)

- ▶ Attributes and Instance Variables
 - A car has *attributes*
 - Color, its number of doors, the amount of gas in its tank, its current speed and its record of total miles driven (i.e., its odometer reading).
 - The car's attributes are represented as part of its design in its engineering diagrams.
 - Every car maintains its *own attributes*.
 - Each car knows how much gas is in its own gas tank, but *not* how much is in the tanks of other cars.



Introduction to Object Technology (Cont.)

► Messages and Methods Calls

- When you drive a car, pressing its gas pedal sends a *message* to the car to perform a task—that is, to go faster.
- Similarly, you send messages to an object.
- Each message is implemented as a **method call** that tells a method of the object to perform its task.



Introduction to Object Technology (Cont.)

- An object, has attributes that it carries along as it's used in a program.
- Specified as part of the object's class.
- A bank account object has a *balance* attribute that represents the amount of money in the account.
- Each bank account object knows the balance in the account it represents, but *not* the balances of the other accounts in the bank.
- Attributes are specified by the class's **instance variables**.



Introduction to Object Technology (Cont.)

▶ Encapsulation

- Classes **encapsulate** (i.e., wrap) attributes and methods into objects—an object's attributes and methods are intimately related.
- Objects may communicate with one another, but they're normally not allowed to know how other objects are implemented—implementation details are *hidden* within the objects themselves.
- **Information hiding**, as we'll see, is crucial to good software engineering.



Introduction to Object Technology (Cont.)

▶ Inheritance

- A new class of objects can be created quickly and conveniently by **inheritance**—the new class absorbs the characteristics of an existing class, possibly customizing them and adding unique characteristics of its own.
- In our car analogy, an object of class “convertible” certainly *is an* object of the more general class “automobile,” but more specifically, the roof can be raised or lowered.



Object Oriented Programming

Course Instructor: Dr. R. Shathanaa



Java

Java and a Typical Java Development Environment



- ▶ Microprocessors are having a profound impact in intelligent consumer-electronic devices.
- ▶ 1991
 - Recognizing this, **Sun Microsystems** funded an internal corporate research project led by **James Gosling**, which resulted in a C++-based object-oriented programming language Sun called Java.
 - Key goal of Java is to be able to write programs that will run on a great variety of computer systems and computer-control devices.
 - This is sometimes called “write once, run anywhere.”



Java and a Typical Java Development Environment (Cont.)

- ▶ 1993
 - The web exploded in popularity
 - Sun saw the potential of using Java to add dynamic content to web pages.
- ▶ Java garnered the attention of the business community because of the phenomenal interest in the web.
- ▶ Java is used to develop large-scale enterprise applications, to enhance the functionality of web servers, to provide applications for consumer devices and for many other purposes.



Java and a Typical Java Development Environment (Cont.)

- ▶ Sun Microsystems was acquired by Oracle in 2010.
- ▶ As of 2010 97% of enterprise desktops, three billion handsets, and 80 million television devices run Java.
- ▶ Java was the most widely used software development language in the world.



1.1 Why Java

- ▶ Java is the preferred language for meeting many organizations' enterprise programming needs.
- ▶ Java has become the language of choice for implementing Internet-based applications and software for devices that communicate over a network.
- ▶ Java USED TO BE most widely used computer programming language.
- ▶ **Java and JavaScript**
- ▶ Although their names are quite similar and they are both used to create dynamic tools and games on a Web page, Java and JavaScript are different languages.



1.1 Introduction (Cont.)

- ▶ The language and its frameworks allow building software that is scalable, highly secure and powerful, which are the three pillars of modern applications
- ▶ OOL:
- ▶ Java, C++, C#, Python, R, PHP, JavaScript, Kotlin, MATLAB



1.1 Introduction (Cont.)

- ▶ Java Editions: SE, EE and ME
 - SE – Standard Edition (Java SE)
 - Used for developing cross-platform, general-purpose applications.
 - Java is used in such a broad spectrum of applications that it has two other editions.
 - The Java Enterprise Edition (Java EE)
 - Geared toward developing large-scale, distributed networking applications and web-based applications.



1.1 Introduction (Cont.)

- Java Micro Edition (Java ME)
- geared toward developing applications for small, memory-constrained devices, such as smartphones.
- Google’s Android operating system
- used on numerous smartphones, tablets (small, lightweight mobile computers with touch screens), e-readers and other devices—uses a customized version of Java not based on Java ME.



Version	Date
JDK Beta	1995
JDK1.0	January 23, 1996
JDK 1.1	February 19, 1997
J2SE 1.2	December 8, 1998
J2SE 1.3	May 8, 2000
J2SE 1.4	February 6, 2002
J2SE 5.0	September 30, 2004
Java SE 6	December 11, 2006
Java SE 7	July 28, 2011
Java SE 8	March 18, 2014
Java SE 9	September 21, 2017
Java SE 10	March 20, 2018
Java SE 11	September 25, 2018
Java SE 12	March 19, 2019
Java SE 13	September 17, 2019
Java SE 14	March 17, 2020
Java SE 15	September 15, 2020
Java SE 16	March 16, 2021



1.9 Java and a Typical Java Development Environment (Cont.)

▶ Java Class Libraries

- Rich collections of existing classes and methods
- Also known as the **Java APIs (Application Programming Interfaces)**.

Bookmark this:

- <https://docs.oracle.com/en/java/javase/16/docs/api/index.html>



Performance Tip 1.1

Using Java API classes and methods instead of writing your own versions can improve program performance, because they're carefully written to perform efficiently. This also shortens program development time.



Portability Tip 1.1

Although it's easier to write portable programs (i.e., programs that can run on many different types of computers) in Java than in most other programming languages, differences between compilers, JVMs and computers can make portability difficult to achieve. Simply writing programs in Java does not guarantee portability.



1.9 Java and a Typical Java Development Environment (Cont.)

- ▶ Java programs normally go through five phases
 - edit
 - compile
 - load
 - verify
 - execute.
 - ▶ Download the JDK and its documentation from
 - www.oracle.com/technetwork/java/javase/downloads/index.html.
 - ▶ Visit Oracle's New to Java Center at:
 - www.oracle.com/technetwork/topics/newtojava/overview/index.html
- What are the phases in executing a C program?



1.9 Java and a Typical Java Development Environment (Cont.)

- ▶ Phase 1 consists of editing a file
 - Type a Java program ([source code](#)) using the editor.
 - Make any necessary corrections.
 - Save the program.
 - A file name ending with the [.java extension](#) indicates that the file contains Java source code.

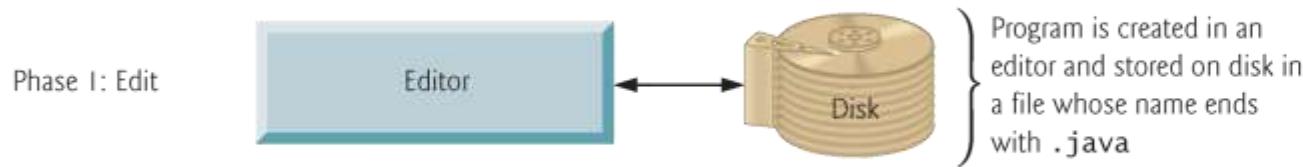


Fig. 1.6 | Typical Java development environment—editing phase.



1.9 Java and a Typical Java Development Environment (Cont.)

- ▶ Linux editors: `vi` and `gedit`.
- ▶ Windows editors:
 - Notepad
 - Notepad++
- ▶ Integrated development environments (IDEs)
 - Provide tools that support the software development process, including editors for writing and editing programs and debuggers for locating **logic errors**—errors that cause programs to execute incorrectly.



1.9 Java and a Typical Java Development Environment (Cont.)

- ▶ Popular IDEs
 - Eclipse (www.eclipse.org)
 - NetBeans (www.netbeans.org).
 - jGRASP™ IDE (www.jgrasp.org)
 - DrJava IDE (www.drjava.org/download.shtml)
 - BlueJ IDE (www.bluej.org/)
 - TextPad® Text Editor for Windows® (www.textpad.com/)



1.9 Java and a Typical Java Development Environment (Cont.)

- ▶ Phase 2: Compiling a Java Program into Bytecodes
 - Use the command `javac` (the **Java compiler**) to **compile** a program. For example, to compile a program called `Welcome.java`, you'd type
 - `javac Welcome.java`
 - If the program compiles, the compiler produces a `.class` file called `Welcome.class` that contains the compiled version of the program.

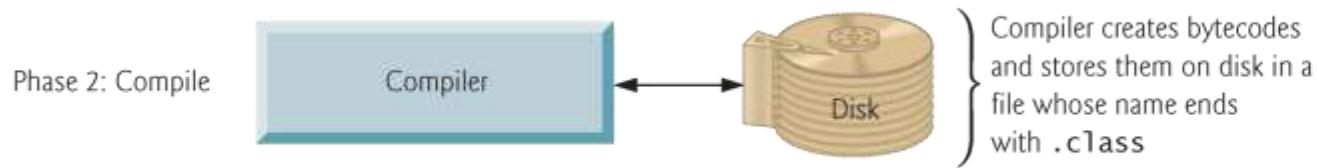


Fig. 1.7 | Typical Java development environment—compilation phase.



1.9 Java and a Typical Java Development Environment (Cont.)

- ▶ Java compiler translates Java source code into **bytecodes** that represent the tasks to execute.
- ▶ Bytecodes are executed by the **Java Virtual Machine (JVM)**—a part of the JDK and the foundation of the Java platform.
- ▶ **Virtual machine (VM)**—a software application that simulates a computer
 - Hides the underlying operating system and hardware from the programs that interact with it.
- ▶ If the same VM is implemented on many computer platforms, applications that it executes can be used on all those platforms.



1.9 Java and a Typical Java Development Environment (Cont.)

- ▶ Bytecodes are platform independent
 - They do not depend on a particular hardware platform.
- ▶ Bytecodes are **portable**
 - The same bytecodes can execute on any platform containing a JVM that understands the version of Java in which the bytecodes were compiled.
- ▶ The JVM is invoked by the **java** command. For example, to execute a Java application called **welcome**, you'd type the command
 - `java welcome`



1.9 Java and a Typical Java Development Environment (Cont.)

- ▶ Phase 3: Loading a Program into Memory
 - The JVM places the program in memory to execute it—this is known as **loading**.
 - **Class loader** takes the **.class** files containing the program's bytecodes and transfers them to primary memory.
 - Also loads any of the **.class** files provided by Java that your program uses.
- ▶ The **.class** files can be loaded from a disk on your system or over a network.

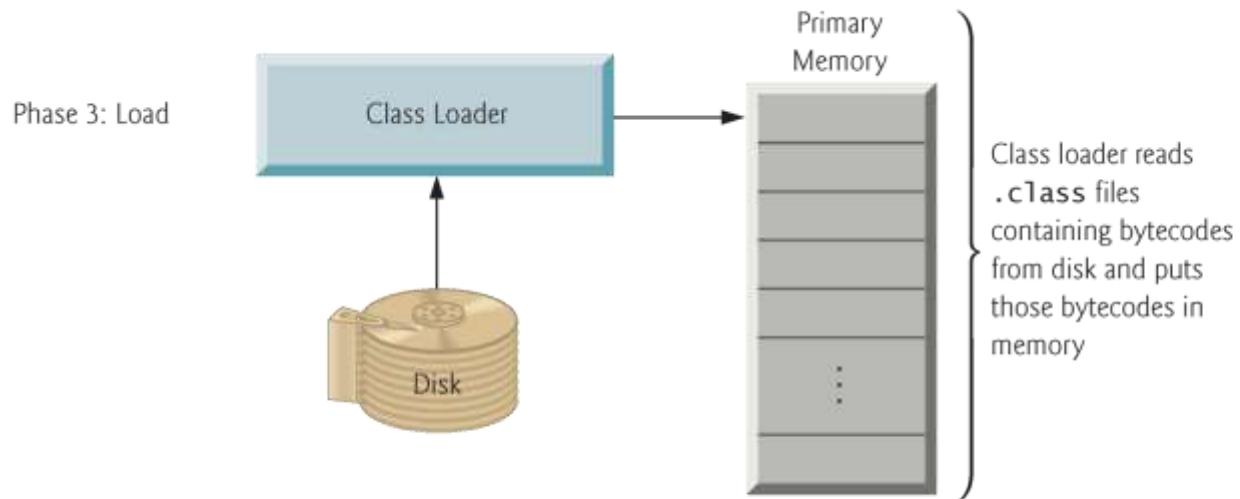


Fig. 1.8 | Typical Java development environment—loading phase.



1.9 Java and a Typical Java Development Environment (Cont.)

- ▶ Phase 4: Bytecode Verification
 - As the classes are loaded, the **bytecode verifier** examines their bytecodes
 - Ensures that they're valid and do not violate Java's security restrictions.
- ▶ Java enforces strong security to make sure that Java programs arriving over the network do not damage your files or your system (as computer viruses and worms might).

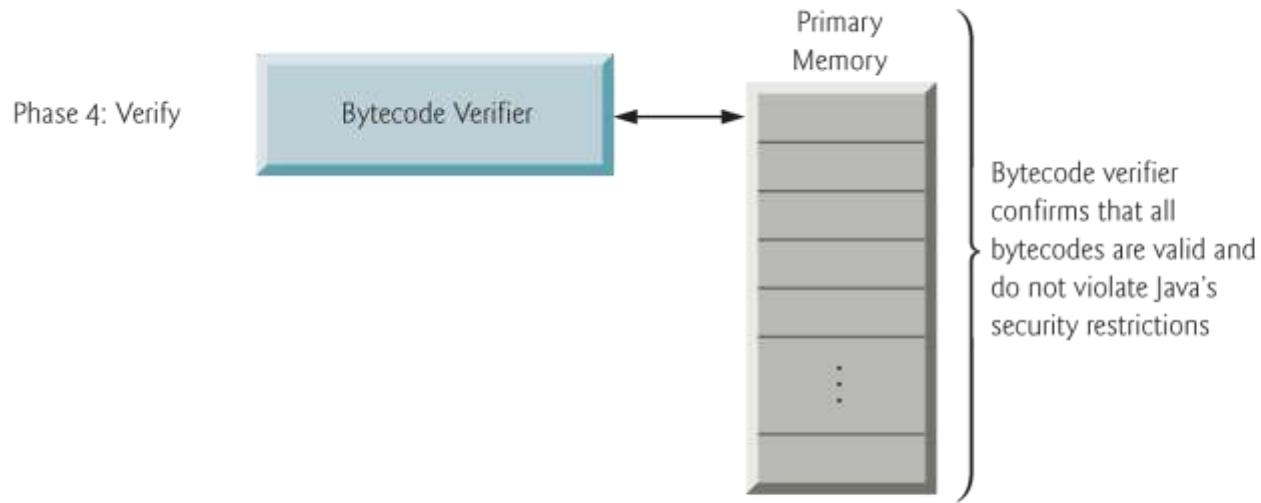


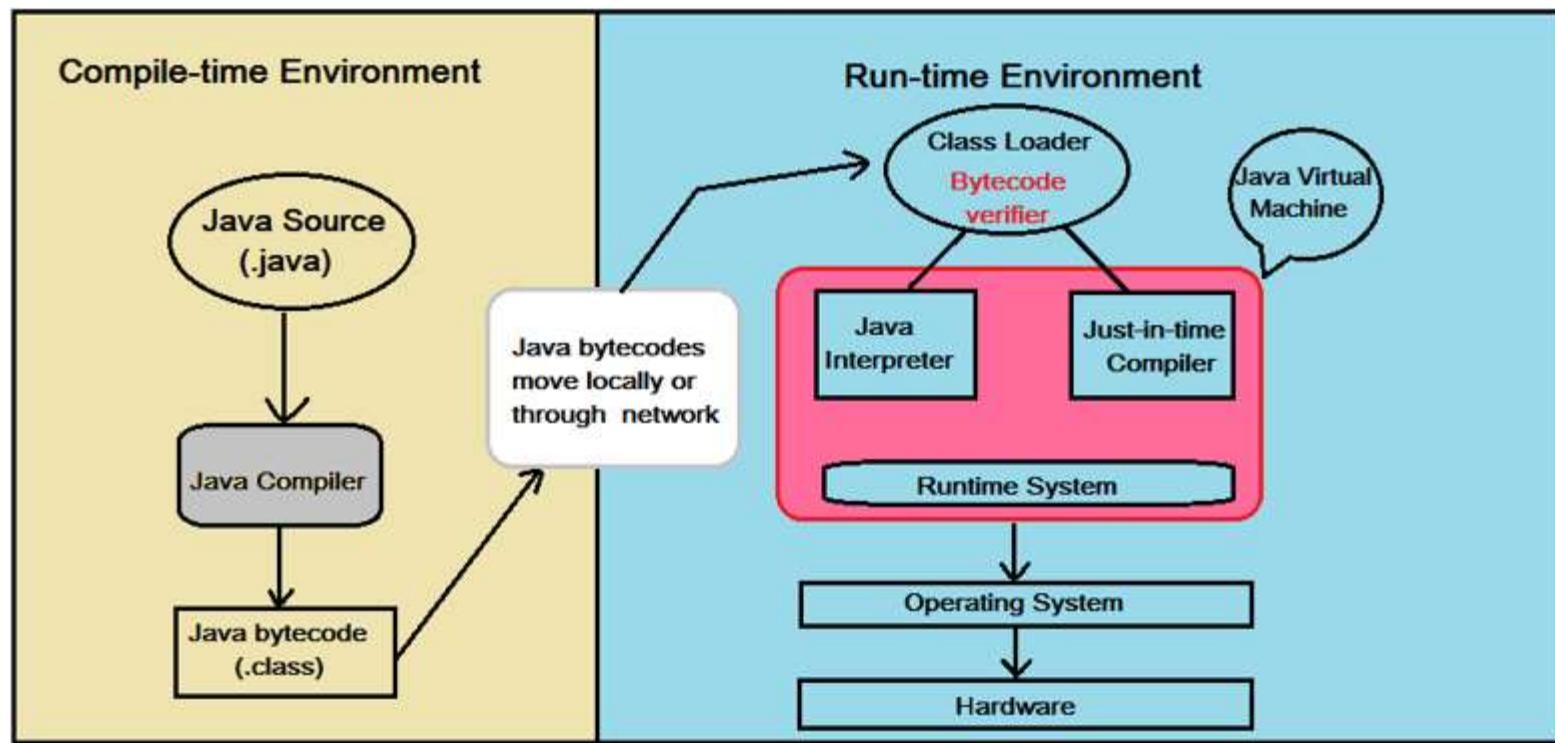
Fig. 1.9 | Typical Java development environment—verification phase.



1.9 Java and a Typical Java Development Environment (Cont.)

► Phase 5: Execution

- The JVM executes the program's bytecodes.
- JVMs typically execute bytecodes using a combination of interpretation and so-called just-in-time (JIT) compilation.
- Analyzes the bytecodes as they're interpreted
- A just-in-time (JIT) compiler—known as the Java HotSpot compiler—translates the bytecodes into the underlying computer's machine language.





1.9 Java and a Typical Java Development Environment (Cont.)

- When the JVM encounters these compiled parts again, the faster machine-language code executes.
- Java programs go through *two compilation phases*
- One in which source code is translated into bytecodes (for portability across JVMs on different computer platforms) and
- A second in which, during execution, the bytecodes are translated into machine language for the actual computer on which the program executes.

Java – Compiled and Interpreted language

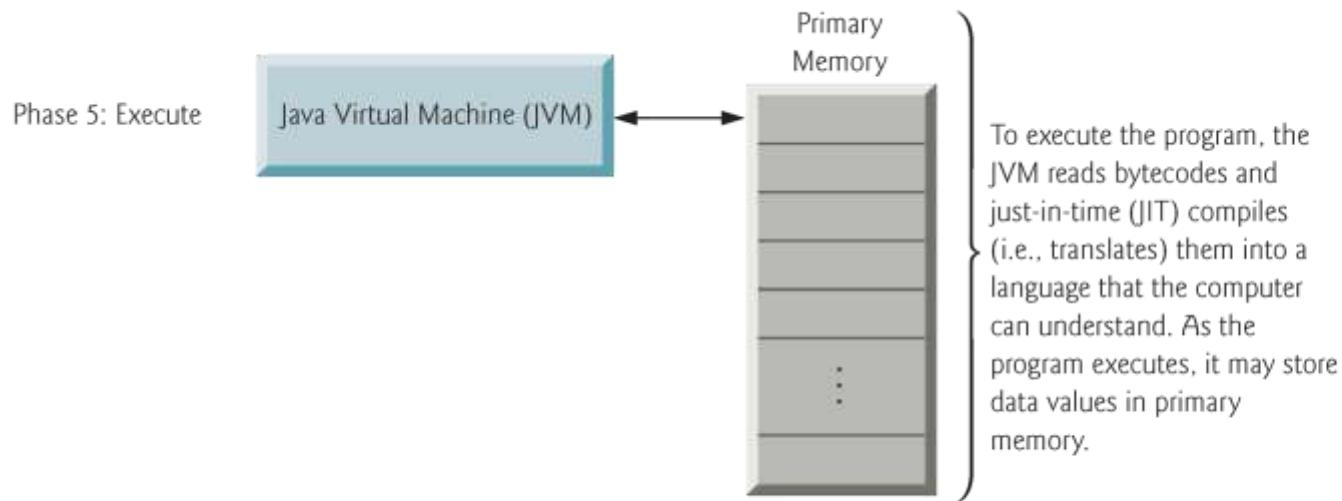


Fig. 1.10 | Typical Java development environment—execution phase.

In order to improve performance, **JIT compilers** interact with the Java Virtual Machine (JVM) at run time and compile suitable bytecode sequences into native machine code.

While using a JIT compiler, **the hardware is able to execute the native code**, as compared to having the JVM interpret the same sequence of bytecode repeatedly and incurring an overhead for the translation process.



Java Installation and Execution

- ▶ Demo
- ▶ <https://www.oracle.com/java/technologies/javase-jdk16-downloads.html>



Introduction to Java Applications



OBJECTIVES

In this chapter you'll learn:

- To write simple Java applications.
- To use input and output statements.
- Java's primitive types.
- Basic memory concepts.
- To use arithmetic operators.
- The precedence of arithmetic operators.
- To write decision-making statements.
- To use relational and equality operators.



-
- 2.1** Introduction
 - 2.2** Your First Program in Java: Printing a Line of Text
 - 2.3** Modifying Your First Java Program
 - 2.4** Displaying Text with `printf`
 - 2.5** Another Application: Adding Integers
 - 2.6** Memory Concepts
 - 2.7** Arithmetic
 - 2.8** Decision Making: Equality and Relational Operators
 - 2.9** Wrap-Up
-



2.1 Introduction

- ▶ Java application programming
- ▶ Use tools from the JDK to compile and run programs.



2.2 Our First Program in Java: Printing a Line of Text

- ▶ Java application
 - A computer program that executes when you use the `java command` to launch the Java Virtual Machine (JVM).
- ▶ Sample program in Fig. 2.1 displays a line of text.



```
1 // Fig. 2.1: Welcome1.java
2 // Text-printing program.
3
4 public class Welcome1
5 {
6     // main method begins execution of Java application
7     public static void main( String[] args )
8     {
9         System.out.println( "Welcome to Java Programming!" );
10    } // end method main
11 } // end class Welcome1
```

```
Welcome to Java Programming!
```

Fig. 2.1 | Text-printing program.



2.2 Our First Program in Java: Printing a Line of Text (Cont.)

► Comments

```
// Fig. 2.1: welcome1.java
```

- // indicates that the line is a **comment**.
 - Used to **document programs** and improve their readability.
 - Compiler ignores comments.
 - A comment that begins with // is an **end-of-line comment**—it terminates at the end of the line on which it appears.
- **Traditional comment**, can be spread over several lines as in

```
/* This is a traditional comment. It  
   can be split over multiple lines */
```

- This type of comment begins with /* and ends with */.
- All text between the delimiters is ignored by the compiler.



2.2 Our First Program in Java: Printing a Line of Text (Cont.)

▶ Javadoc comments

- Delimited by `/**` and `*/`.
- All text between the Javadoc comment delimiters is ignored by the compiler.
- Enable you to embed program documentation directly in your programs.
- The [javadoc utility program](#) (Appendix M) reads Javadoc comments and uses them to prepare your program's documentation in HTML format.



Common Programming Error 2.1

*Forgetting one of the delimiters of a traditional or Java-doc comment is a syntax error. A **syntax error** occurs when the compiler encounters code that violates Java's language rules (i.e., its syntax). These rules are similar to a natural language's grammar rules specifying sentence structure. Syntax errors are also called **compiler errors**, **compile-time errors** or **compilation errors**, because the compiler detects them during the compilation phase. The compiler responds by issuing an error message and preventing your program from compiling.*



Good Programming Practice 2.1

Some organizations require that every program begin with a comment that states the purpose of the program and the author, date and time when the program was last modified.



2.2 Our First Program in Java: Printing a Line of Text (Cont.)

- ▶ Blank lines and space characters
 - Make programs easier to read.
 - Blank lines, spaces and tabs are known as **white space** (or whitespace).
 - White space is ignored by the compiler.



Good Programming Practice 2.2

Use blank lines and spaces to enhance program readability.



2.2 Our First Program in Java: Printing a Line of Text (Cont.)

▶ Class declaration

```
public class Welcome1
```

- Every Java program consists of at least one class that you define.
- **class keyword** introduces a class declaration and is immediately followed by the **class name**.
- **Keywords** (Appendix C) are reserved for use by Java and are always spelled with all lowercase letters.



2.2 Our First Program in Java: Printing a Line of Text (Cont.)

▶ Class names

- By convention, begin with a capital letter and capitalize the first letter of each word they include (e.g., `SampleClassName`).
- A class name is an **identifier**—a series of characters consisting of letters, digits, underscores (_) and dollar signs (\$) that does not begin with a digit and does not contain spaces.
- Java is **case sensitive**—uppercase and lowercase letters are distinct—so `a1` and `A1` are different (but both valid) identifiers.



Common Programming Error 2.2

A public class must be placed in a file that has the same name as the class (in terms of both spelling and capitalization) plus the .java extension; otherwise, a compilation error occurs. For example, public class Welcome must be placed in a file named Welcome.java.



2.2 Our First Program in Java: Printing a Line of Text (Cont.)

► Braces

- A **left brace**, `{`, begins the **body** of every class declaration.
- A corresponding **right brace**, `}`, must end each class declaration.
- Code between braces should be indented.
- This indentation is one of the spacing conventions mentioned earlier.



Error-Prevention Tip 2.1

When you type an opening left brace, {, immediately type the closing right brace, }, then reposition the cursor between the braces and indent to begin typing the body. This practice helps prevent errors due to missing braces. Many IDEs insert the braces for you.



Common Programming Error 2.3

It's a syntax error if braces do not occur in matching pairs.



Good Programming Practice 2.3

Indent the entire body of each class declaration one “level” between the left brace and the right brace that delimit the body of the class. We recommend using three spaces to form a level of indent. This format emphasizes the class declaration’s structure and makes it easier to read.



Good Programming Practice 2.4

Many IDEs insert indentation for you in all the right places. The Tab key may also be used to indent code, but tab stops vary among text editors. Most IDEs allow you to configure tabs such that a specified number of spaces is inserted each time you press the Tab key.



2.2 Our First Program in Java: Printing a Line of Text (Cont.)

▶ Declaring the `main` Method

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

- Starting point of every Java application.
- Parentheses after the identifier `main` indicate that it's a program building block called a `method`.
- Java class declarations normally contain one or more methods.
- `main` must be defined as shown; otherwise, the JVM will not execute the application.
- Methods perform tasks and can return information when they complete their tasks.
- Keyword `void` indicates that this method will not return any information.



2.2 Our First Program in Java: Printing a Line of Text (Cont.)

- ▶ Body of the method declaration
 - Enclosed in left and right braces.
- ▶ Statement

```
System.out.println("Welcome to Java Programming!");
```

- Instructs the computer to perform an action
 - Print the **string** of characters contained between the double quotation marks.
- A string is sometimes called a **character string** or a **string literal**.
- White-space characters in strings are not ignored by the compiler.
- Strings cannot span multiple lines of code.



Good Programming Practice 2.5

Indent the entire body of each method declaration one “level” between the braces that define the body of the method. This makes the structure of the method stand out and makes the method declaration easier to read.



2.2 Our First Program in Java: Printing a Line of Text (Cont.)

- ▶ **System.out** object
 - Standard output object.
 - Allows Java applications to display strings in the **command window** from which the Java application executes.
- ▶ **System.out.println** method
 - Displays (or prints) a line of text in the command window.
 - The string in the parentheses the **argument** to the method.
 - Positions the output cursor at the beginning of the next line in the command window.
- ▶ Most statements end with a semicolon.



Error-Prevention Tip 2.2

When learning how to program, sometimes it's helpful to "break" a working program so you can familiarize yourself with the compiler's syntax-error messages. These messages do not always state the exact problem in the code.

When you encounter an error message, it will give you an idea of what caused the error. [Try removing a semicolon or brace from the program of Fig. 2.1, then recompile the program to see the error messages generated by the omission.]



Error-Prevention Tip 2.3

When the compiler reports a syntax error, it may not be on the line that the error message indicates. First, check the line for which the error was reported. If you don't find an error on that line, check several preceding lines.



2.2 Our First Program in Java: Printing a Line of Text (Cont.)

- ▶ Compiling and Executing Your First Java Application
 - Open a command window and change to the directory where the program is stored.
 - Many operating systems use the command **cd** to change directories.
 - To compile the program, type

```
javac welcome1.java
```
 - If the program contains no syntax errors, preceding command creates a **.class** file (known as the **class file**) containing the platform-independent Java bytecodes that represent the application.
 - When we use the **java** command to execute the application on a given platform, these bytecodes will be translated by the JVM into instructions that are understood by the underlying operating system.



Error-Prevention Tip 2.4

When attempting to compile a program, if you receive a message such as “bad command or filename,” “javac: command not found” or “'javac' is not recognized as an internal or external command, operable program or batch file,” then your Java software installation was not completed properly. If you’re using the JDK, this indicates that the system’s PATH environment variable was not set properly. Please carefully review the installation instructions in the Before You Begin section of this book. On some systems, after correcting the PATH, you may need to reboot your computer or open a new command window for these settings to take effect.



Error-Prevention Tip 2.5

Each syntax-error message contains the file name and line number where the error occurred. For example, Welcome1.java:6 indicates that an error occurred at line 6 in Welcome1.java. The rest of the message provides information about the syntax error.



Error-Prevention Tip 2.6

The compiler error message “class Welcome1 is public, should be declared in a file named Welcome1.java” indicates that the file name does not match the name of the public class in the file or that you typed the class name incorrectly when compiling the class.



2.2 Our First Program in Java: Printing a Line of Text (Cont.)

- ▶ To execute the program, type `java welcome1`.
- ▶ Launches the JVM, which loads the `.class` file for class `Welcome1`.
- ▶ Note that the `.class` file-name extension is omitted from the preceding command; otherwise, the JVM will not execute the program.
- ▶ The JVM calls method `main` to execute the program.



```
C:\examples\ch02\fig02_01>javac Welcome1.java
C:\examples\ch02\fig02_01>java Welcome1
Welcome to Java Programming!
C:\examples\ch02\fig02_01>
```

You type this command to execute the application

The program outputs to the screen
Welcome to Java Programming!

Fig. 2.2 | Executing `Welcome1` from the **Command Prompt**.



Error-Prevention Tip 2.7

When attempting to run a Java program, if you receive a message such as “Exception in thread "main"

java.lang.NoClassDefFoundError:

Welcome1,” your CLASSPATH environment variable has not been set properly. Please carefully review the installation instructions in the Before You Begin section of this book. On some systems, you may need to reboot your computer or open a new command window after configuring the CLASSPATH.



2.3 Modifying Your First Java Program

- ▶ Class `Welcome2`, shown in Fig. 2.3, uses two statements to produce the same output as that shown in Fig. 2.1.
- ▶ New and key features in each code listing are highlighted.
- ▶ `System.out`'s method `print` displays a string.
- ▶ Unlike `println`, `print` does not position the output cursor at the beginning of the next line in the command window.
 - The next character the program displays will appear immediately after the last character that `print` displays.



```
1 // Fig. 2.3: Welcome2.java
2 // Printing a line of text with multiple statements.
3
4 public class Welcome2
{
    // main method begins execution of Java application
7    public static void main( String[] args )
8    {
9        System.out.print( "Welcome to " );
10       System.out.println( "Java Programming!" );
11    } // end method main
12 } // end class Welcome2
```

Welcome to Java Programming!

Prints Welcome to and leaves cursor on same line.

Prints Java Programming! starting where the cursor was positioned previously, then outputs a newline character

Fig. 2.3 | Printing a line of text with multiple statements.



2.3 Modifying Your First Java Program (Cont.)

- ▶ Newline characters indicate to `System.out`'s `print` and `println` methods when to position the output cursor at the beginning of the next line in the command window.
- ▶ Newline characters are white-space characters.
- ▶ The **backslash (\)** is called an **escape character**.
 - Indicates a “special character”
- ▶ Backslash is combined with the next character to form an **escape sequence**.
- ▶ The escape sequence `\n` represents the newline character.
- ▶ Complete list of escape sequences

[java.sun.com/docs/books/jls/third_edition/html/
lexical.html#3.10.6](http://java.sun.com/docs/books/jls/third_edition/html/lexical.html#3.10.6).



```
1 // Fig. 2.4: Welcome3.java
2 // Printing multiple lines of text with a single statement.
3
4 public class Welcome3
{
    // main method begins execution of Java application
7    public static void main( String[] args )
8    {
9        System.out.println( "Welcome\n to\n Java\n Programming!" );
10    } // end method main
11 } // end class Welcome3
```

```
Welcome
to
Java
Programming!
```

Each \n moves the output cursor to the next line, where output continues

Fig. 2.4 | Printing multiple lines of text with a single statement.



Escape sequence	Description
\n	Newline. Position the screen cursor at the beginning of the next line.
\t	Horizontal tab. Move the screen cursor to the next tab stop.
\r	Carriage return. Position the screen cursor at the beginning of the current line—do <i>not</i> advance to the next line. Any characters output after the carriage return overwrite the characters previously output on that line.
\\"	Backslash. Used to print a backslash character.
\"	Double quote. Used to print a double-quote character. For example, <code>System.out.println("\"in quotes\"");</code> displays "in quotes".

Fig. 2.5 | Some common escape sequences.



2.4 Displaying Text with `printf`

- ▶ `System.out.printf` method
 - `f` means “formatted”
 - displays formatted data
- ▶ Multiple method arguments are placed in a `comma-separated list`.
- ▶ Java allows large statements to be split over many lines.
 - Cannot split a statement in the middle of an identifier or string.
- ▶ Method `printf`'s first argument is a `format string`
 - May consist of `fixed text` and `format specifiers`.
 - Fixed text is output as it would be by `print` or `println`.
 - Each format specifier is a placeholder for a value and specifies the type of data to output.
- ▶ Format specifiers begin with a percent sign (%) and are followed by a character that represents the data type.
- ▶ Format specifier `%s` is a placeholder for a string.



Good Programming Practice 2.6

Place a space after each comma (,) in an argument list to make programs more readable.



```
1 // Fig. 2.6: Welcome4.java
2 // Displaying multiple lines with method System.out.printf.
3
4 public class Welcome4
{
    // main method begins execution of Java application
7    public static void main( String[] args )
8    {
9        System.out.printf( "%s\n%s\n",
10            "Welcome to", "Java Programming!" );
11    } // end method main
12 } // end class Welcome4
```

```
Welcome to
Java Programming!
```

Each %s is a placeholder for a String
that comes later in the argument list

Statements can be split over multiple
lines.

Fig. 2.6 | Displaying multiple lines with method `System.out.printf`.



Common Programming Error 2.4

Splitting a statement in the middle of an identifier or a string is a syntax error.



2.5 Another Application: Adding Integers

- ▶ **Integers**
 - Whole numbers, like -22 , 7 , 0 and 1024)
- ▶ Programs remember numbers and other data in the computer's memory and access that data through program elements called **variables**.
- ▶ The program of Fig. 2.7 demonstrates these concepts.



```
1 // Fig. 2.7: Addition.java
2 // Addition program that displays the sum of two numbers.
3 import java.util.Scanner; // program uses class Scanner
4
5 public class Addition
6 {
7     // main method begins execution of Java application
8     public static void main( String[] args )
9     {
10         // create a Scanner to obtain input from the command window
11         Scanner input = new Scanner( System.in );
12
13         int number1; // first number to add
14         int number2; // second number to add
15         int sum; // sum of number1 and number2
16
17         System.out.print( "Enter first integer: " ); // prompt
18         number1 = input.nextInt(); // read first number from user
19
20         System.out.print( "Enter second integer: " ); // prompt
21         number2 = input.nextInt(); // read second number from user
22
23         sum = number1 + number2; // add numbers, then store total in sum
```

Imports class Scanner for use in this program

Creates Scanner for reading data from the user

Variables that are declared but not initialized

Reads an int value from the user

Reads another int value from the user

Sums the values of number1 and number2

Fig. 2.7 | Addition program that displays the sum of two numbers. (Part 1 of 2.)



```
24
25     System.out.printf( "Sum is %d\n", sum ); // display sum
26 } // end method main
27 } // end class Addition
```

```
Enter first integer: 45
Enter second integer: 72
Sum is 117
```

Fig. 2.7 | Addition program that displays the sum of two numbers. (Part 2 of 2.)



2.5 Another Application: Adding Integers (Cont.)

▶ import declaration

- Helps the compiler locate a class that is used in this program.
- Rich set of predefined classes that you can reuse rather than “reinventing the wheel.”
- Classes are grouped into **packages**—named groups of related classes—and are collectively referred to as the **Java class library**, or the **Java Application Programming Interface (Java API)**.
- You use **import** declarations to identify the predefined classes used in a Java program.



Common Programming Error 2.5

All `import` declarations must appear before the first class declaration in the file. Placing an `import` declaration inside or after a class declaration is a syntax error.



Error-Prevention Tip 2.8

Forgetting to include an import declaration for a class used in your program typically results in a compilation error containing a message such as “cannot find symbol.” When this occurs, check that you provided the proper import declarations and that the names in them are correct, including proper capitalization.



2.5 Another Application: Adding Integers (Cont.)

▶ Variable declaration statement

```
Scanner input = new Scanner( System.in );
```

- Specifies the name (**input**) and type (**Scanner**) of a variable that is used in this program.
- ### ▶ Variable

- A location in the computer's memory where a value can be stored for use later in a program.
- Must be declared with a **name** and a **type** before they can be used.
- A variable's name enables the program to access the value of the variable in memory.
- The name can be any valid identifier.
- A variable's type specifies what kind of information is stored at that location in memory.



2.5 Another Application: Adding Integers (Cont.)

- ▶ **Scanner**
 - Enables a program to read data for use in a program.
 - Data can come from many sources, such as the user at the keyboard or a file on disk.
 - Before using a **Scanner**, you must create it and specify the source of the data.
- ▶ The equals sign (=) in a declaration indicates that the variable should be **initialized** (i.e., prepared for use in the program) with the result of the expression to the right of the equals sign.
- ▶ The **new** keyword creates an object.
- ▶ **Standard input object, System.in**, enables applications to read bytes of information typed by the user.
- ▶ **Scanner** object translates these bytes into types that can be used in a program.



2.5 Another Application: Adding Integers (Cont.)

- ▶ Variable declaration statements

```
int number1; // first number to add
int number2; // second number to add
int sum; // sum of number1 and number2
```

declare that variables `number1`, `number2` and `sum` hold data of type `int`

- They can hold integer.
- Range of values for an `int` is $-2,147,483,648$ to $+2,147,483,647$.
- Actual `int` values may not contain commas.
- ▶ Several variables of the same type may be declared in one declaration with the variable names separated by commas.



Good Programming Practice 2.7

Declare each variable on a separate line. This format allows a descriptive comment to be inserted next to each declaration.



Good Programming Practice 2.8

Choosing meaningful variable names helps a program to be self-documenting (i.e., one can understand the program simply by reading it rather than by reading manuals or viewing an excessive number of comments).



Good Programming Practice 2.9

By convention, variable-name identifiers begin with a lowercase letter, and every word in the name after the first word begins with a capital letter. For example, variable-name identifier `firstNumber` starts its second word, `Number`, with a capital N.



2.5 Another Application: Adding Integers (Cont.)

- ▶ **Prompt**
 - Output statement that directs the user to take a specific action.
- ▶ **System** is a class.
 - Part of package `java.lang`.
 - Class **System** is not imported with an `import` declaration at the beginning of the program.



Software Engineering Observation 2.1

By default, package java.lang is imported in every Java program; thus, classes in java.lang are the only ones in the Java API that do not require an import declaration.



2.5 Another Application: Adding Integers (Cont.)

▶ Scanner method `nextInt`

```
number1 = input.nextInt(); // read first number from user
```

- Obtains an integer from the user at the keyboard.
 - Program waits for the user to type the number and press the Enter key to submit the number to the program.
- ▶ The result of the call to method `nextInt` is placed in variable `number1` by using the **assignment operator**, `=`.
- “`number1` gets the value of `input.nextInt()`.”
 - Operator `=` is called a **binary operator**—it has two **operands**.
 - Everything to the right of the assignment operator, `=`, is always evaluated before the assignment is performed.



Good Programming Practice 2.10

Placing spaces on either side of a binary operator makes the program more readable.



2.5 Another Application: Adding Integers (Cont.)

► Arithmetic

```
sum = number1 + number2; // add numbers
```

- Assignment statement that calculates the sum of the variables **number1** and **number2** then assigns the result to variable **sum** by using the assignment operator, **=**.
- “**sum** gets the value of **number1 + number2**.”
- In general, calculations are performed in assignment statements.
- Portions of statements that contain calculations are called **expressions**.
- An expression is any portion of a statement that has a value associated with it.



2.5 Another Application: Adding Integers (Cont.)

- ▶ Integer formatted output

```
System.out.printf( "Sum is %d\n", sum );
```

- Format specifier **%d** is a placeholder for an **int** value
- The letter **d** stands for “decimal integer.”



2.6 Memory Concepts

▶ Variables

- Every variable has a **name**, a **type**, a **size** (in bytes) and a **value**.
- When a new value is placed into a variable, the new value replaces the previous value (if any)
- The previous value is lost.



Fig. 2.8 | Memory location showing the name and value of variable `number1`.



Fig. 2.9 | Memory locations after storing values for `number1` and `number2`.

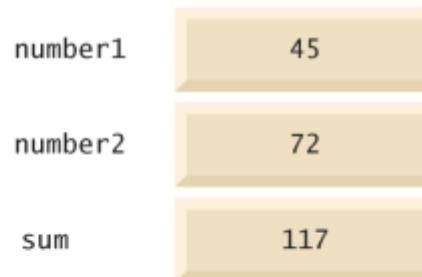


Fig. 2.10 | Memory locations after storing the sum of `number1` and `number2`.



2.7 Arithmetic

- ▶ Arithmetic operators are summarized in Fig. 2.11.
- ▶ The asterisk (*) indicates multiplication
- ▶ The percent sign (%) is the remainder operator
- ▶ The arithmetic operators are binary operators because they each operate on two operands.
- ▶ Integer division yields an integer quotient.
 - Any fractional part in integer division is simply discarded (i.e., truncated)—no rounding occurs.
- ▶ The remainder operator, %, yields the remainder after division.



Java operation	Operator	Algebraic expression	Java expression
Addition	+	$f + 7$	<code>f + 7</code>
Subtraction	-	$p - c$	<code>p - c</code>
Multiplication	*	bm	<code>b * m</code>
Division	/	x/y or $\frac{x}{y}$ or $x \div y$	<code>x / y</code>
Remainder	%	$r \bmod s$	<code>r % s</code>

Fig. 2.11 | Arithmetic operators.



2.7 Arithmetic (Cont.)

- ▶ Arithmetic expressions in Java must be written in **straight-line form** to facilitate entering programs into the computer.
- ▶ Expressions such as “ a divided by b ” must be written as a / b , so that all constants, variables and operators appear in a straight line.
- ▶ Parentheses are used to group terms in expressions in the same manner as in algebraic expressions.
- ▶ If an expression contains **nested parentheses**, the expression in the innermost set of parentheses is evaluated first.



2.7 Arithmetic (Cont.)

- ▶ Rules of operator precedence
 - Multiplication, division and remainder operations are applied first.
 - If an expression contains several such operations, they are applied from left to right.
 - Multiplication, division and remainder operators have the same level of precedence.
 - Addition and subtraction operations are applied next.
 - If an expression contains several such operations, the operators are applied from left to right.
 - Addition and subtraction operators have the same level of precedence.
- ▶ When we say that operators are applied from left to right, we are referring to their associativity.
- ▶ Some operators associate from right to left.
- ▶ Complete precedence chart is included in Appendix A.



Operator(s)	Operation(s)	Order of evaluation (precedence)
*	Multiplication	Evaluated first. If there are several operators of this type, they're evaluated from left to right.
/	Division	
%	Remainder	
+	Addition	Evaluated next. If there are several operators of this type, they're evaluated from left to right.
-	Subtraction	
=	Assignment	Evaluated last.

Fig. 2.12 | Precedence of arithmetic operators.

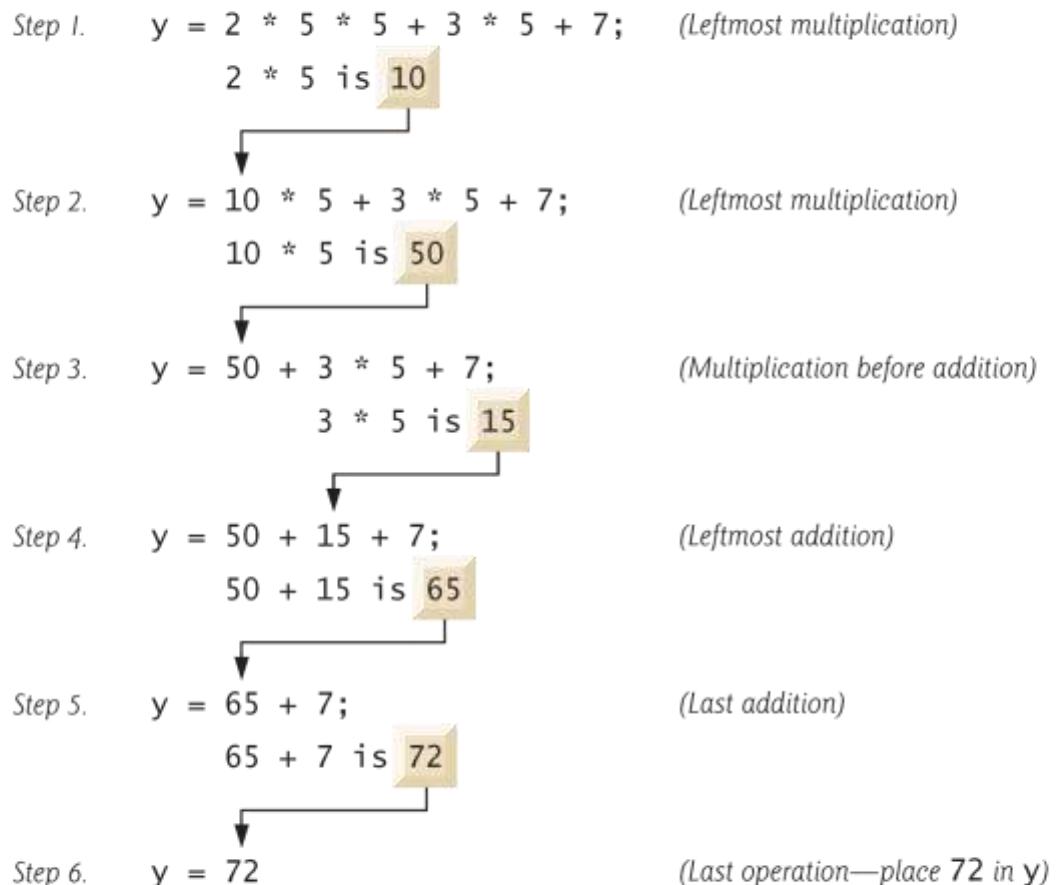


Fig. 2.13 | Order in which a second-degree polynomial is evaluated.



2.7 Arithmetic (Cont.)

- As in algebra, it's acceptable to place **redundant parentheses** (unnecessary parentheses) in an expression to make the expression clearer.



Primitive Datatypes

****Imp** No unsigned primitive types**

Data Type	Size	Description
byte	1 byte	Stores whole numbers from -128 to 127
short	2 bytes	Stores whole numbers from -32,768 to 32,767
int	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,647
long	8 bytes	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4 bytes	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits
double	8 bytes	Stores fractional numbers. Sufficient for storing 15 decimal digits
boolean	1 bit	Stores true or false values
char	2 bytes	Stores a single character/letter or ASCII values



Format Specifier	Conversion Applied
%%	Inserts a % sign
%x %X	Integer hexadecimal
%t %T	Time and Date
%s %S	String
%n	Inserts a newline character
%o	Octal integer
%f	Decimal floating-point
%e %E	Scientific notation
%g	Causes Formatter to use either %f or %e, whichever is shorter
%h %H	Hash code of the argument
%d	Decimal integer
%c	Character
%b %B	Boolean
%a %A	Floating-point hexadecimal



Type conversion

Widening or Automatic Type Conversion

- ▶ Widening conversion takes place when two data types are automatically converted. This happens when:
 - The two data types are compatible.
 - When we assign value of a smaller data type to a bigger data type.

Byte → Short → Int → Long –> Float → Double

Widening or Automatic Conversion

No automatic conversion is supported from numeric type to char or boolean



Narrowing or Explicit Conversion

- ▶ If we want to assign a value of larger data type to a smaller data type we perform explicit type casting or narrowing.
 - This is useful for incompatible data types where automatic conversion cannot be done.
 - Here, target-type specifies the desired type to convert the specified value to.

Double → Float → Long → Int → Short → Byte

Narrowing or Explicit Conversion



2.8 Decision Making: Equality and Relational Operators

- ▶ Condition
 - An expression that can be **true** or **false**. (**Boolean type**)
- ▶ if selection statement
 - Allows a program to make a **decision** based on a condition's value.
- ▶ Equality operators (**==** and **!=**)
- ▶ Relational operators (**>**, **<**, **>=** and **<=**)
- ▶ Both equality operators have the same level of precedence, which is lower than that of the relational operators.
- ▶ The equality operators associate from left to right.
- ▶ The relational operators all have the same level of precedence and also associate from left to right.



Standard algebraic equality or relational operator	Java equality or relational operator	Sample Java condition	Meaning of Java condition
<i>Equality operators</i>			
=	==	x == y	x is equal to y
≠	!=	x != y	x is not equal to y
<i>Relational operators</i>			
>	>	x > y	x is greater than y
<	<	x < y	x is less than y
≥	≥	x ≥ y	x is greater than or equal to y
≤	≤	x ≤ y	x is less than or equal to y

Fig. 2.14 | Equality and relational operators.



```
1 // Fig. 2.15: Comparison.java
2 // Compare integers using if statements, relational operators
3 // and equality operators.
4 import java.util.Scanner; // program uses class Scanner
5
6 public class Comparison
7 {
8     // main method begins execution of Java application
9     public static void main( String[] args )
10    {
11        // create Scanner to obtain input from command line
12        Scanner input = new Scanner( System.in );
13
14        int number1; // first number to compare
15        int number2; // second number to compare
16
17        System.out.print( "Enter first integer: " ); // prompt
18        number1 = input.nextInt(); // read first number from user
19
20        System.out.print( "Enter second integer: " ); // prompt
21        number2 = input.nextInt(); // read second number from user
22    }
```

Fig. 2.15 | Compare integers using if statements, relational operators and equality operators. (Part I of 3.)



```
23 if ( number1 == number2 )           ← Output statement executes only if the
24     System.out.printf( "%d == %d\n", number1, number2 );
25
26 if ( number1 != number2 )          ← Output statement executes only if the
27     System.out.printf( "%d != %d\n", number1, number2 );
28
29 if ( number1 < number2 )           ← Output statement executes only if
30     System.out.printf( "%d < %d\n", number1, number2 );
31
32 if ( number1 > number2 )           ← Output statement executes only if
33     System.out.printf( "%d > %d\n", number1, number2 );
34
35 if ( number1 <= number2 )          ← Output statement executes only if
36     System.out.printf( "%d <= %d\n", number1, number2 );
37
38 if ( number1 >= number2 )          ← Output statement executes only if
39     System.out.printf( "%d >= %d\n", number1, number2 );
40 } // end method main
41 } // end class Comparison
```

Fig. 2.15 | Compare integers using **if** statements, relational operators and equality operators. (Part 2 of 3.)



```
Enter first integer: 777
Enter second integer: 777
777 == 777
777 <= 777
777 >= 777
```

```
Enter first integer: 1000
Enter second integer: 2000
1000 != 2000
1000 < 2000
1000 <= 2000
```

```
Enter first integer: 2000
Enter second integer: 1000
2000 != 1000
2000 > 1000
2000 >= 1000
```

Fig. 2.15 | Compare integers using if statements, relational operators and equality operators. (Part 3 of 3.)



2.8 Decision Making: Equality and Relational Operators (Cont.)

- ▶ An **if** statement always begins with keyword **if**, followed by a condition in parentheses.
 - Expects one statement in its body, but may contain multiple statements if they are enclosed in a set of braces (**{ }**).
 - The indentation of the body statement is not required, but it improves the program's readability by emphasizing that statements are part of the body.
- ▶ Note that there is no semicolon (**;**) at the end of the first line of each **if** statement.
 - Such a semicolon would result in a logic error at execution time.
 - Treated as the **empty statement**—semicolon by itself.



Common Programming Error 2.6

Confusing the equality operator, ==, with the assignment operator, =, can cause a logic error or a syntax error. The equality operator should be read as “is equal to” and the assignment operator as “gets” or “gets the value of.” To avoid confusion, some people read the equality operator as “double equals” or “equals equals.”



Common Programming Error 2.7

Placing a semicolon immediately after the right parenthesis of the condition in an if statement is normally a logic error.



Error-Prevention Tip 2.9

A lengthy statement can be spread over several lines. If a single statement must be split across lines, choose breaking points that make sense, such as after a comma in a comma-separated list, or after an operator in a lengthy expression. If a statement is split across two or more lines, indent all subsequent lines until the end of the statement.



Good Programming Practice 2.12

When writing expressions containing many operators, refer to the operator precedence chart (Appendix A) . Confirm that the operations in the expression are performed in the order you expect. If, in a complex expression, you're uncertain about the order of evaluation, use parentheses to force the order, exactly as you'd do in algebraic expressions.

Operators	Associativity				Type
*	/	%		left to right	multiplicative
+	-			left to right	additive
<	\leq	>	\geq	left to right	relational
\equiv	\neq			left to right	equality
=				right to left	assignment

Fig. 2.16 | Precedence and associativity of operators discussed.



4.11 Compound Assignment Operators

- ▶ Compound assignment operators abbreviate assignment expressions.
- ▶ Statements like

variable = variable operator expression;

where operator is one of the binary operators +, -, *, / or % can be written in the form

variable operator= expression;

- ▶ Example:

`c = c + 3;`

can be written with the **addition compound assignment operator**, **`+=`**, as

`c += 3;`

- ▶ The `+=` operator adds the value of the expression on its right to the value of the variable on its left and stores the result in the variable on the left of the operator.



Assignment operator	Sample expression	Explanation	Assigns
<i>Assume: int c = 3, d = 5, e = 4, f = 6, g = 12;</i>			
<code>+=</code>	<code>c += 7</code>	<code>c = c + 7</code>	10 to c
<code>-=</code>	<code>d -= 4</code>	<code>d = d - 4</code>	1 to d
<code>*=</code>	<code>e *= 5</code>	<code>e = e * 5</code>	20 to e
<code>/=</code>	<code>f /= 3</code>	<code>f = f / 3</code>	2 to f
<code>%=</code>	<code>g %= 9</code>	<code>g = g % 9</code>	3 to g

Fig. 4.13 | Arithmetic compound assignment operators.



4.12 Increment and Decrement Operators

- ▶ Unary **increment operator**, `++`, adds one to its operand
- ▶ Unary **decrement operator**, `--`, subtracts one from its operand
- ▶ An increment or decrement operator that is prefixed to (placed before) a variable is referred to as the **prefix increment** or **prefix decrement operator**, respectively.
- ▶ An increment or decrement operator that is postfixified to (placed after) a variable is referred to as the **postfix increment** or **postfix decrement operator**, respectively.



Operator	Operator name	Sample expression	Explanation
<code>++</code>	prefix increment	<code>++a</code>	Increment <code>a</code> by 1, then use the new value of <code>a</code> in the expression in which <code>a</code> resides.
<code>++</code>	postfix increment	<code>a++</code>	Use the current value of <code>a</code> in the expression in which <code>a</code> resides, then increment <code>a</code> by 1.
<code>--</code>	prefix decrement	<code>--b</code>	Decrement <code>b</code> by 1, then use the new value of <code>b</code> in the expression in which <code>b</code> resides.
<code>--</code>	postfix decrement	<code>b--</code>	Use the current value of <code>b</code> in the expression in which <code>b</code> resides, then decrement <code>b</code> by 1.

Fig. 4.14 | Increment and decrement operators.



4.12 Increment and Decrement Operators (Cont.)

- ▶ Using the prefix increment (or decrement) operator to add (or subtract) 1 from a variable is known as **preincrementing** (or **predecrementing**) the variable.
- ▶ Preincrementing (or predecrementing) a variable causes the variable to be incremented (decremented) by 1; then the new value is used in the expression in which it appears.
- ▶ Using the postfix increment (or decrement) operator to add (or subtract) 1 from a variable is known as **postincrementing** (or **postdecrementing**) the variable.
- ▶ This causes the current value of the variable to be used in the expression in which it appears; then the variable's value is incremented (decremented) by 1.



Good Programming Practice 4.6

Unlike binary operators, the unary increment and decrement operators should be placed next to their operands, with no intervening spaces.



```
1 // Fig. 4.15: Increment.java
2 // Prefix increment and postfix increment operators.
3
4 public class Increment
{
5     public static void main( String[] args )
6     {
7         int c;
8
9         // demonstrate postfix increment operator
10        c = 5; // assign 5 to c
11        System.out.println( c ); // prints 5
12        System.out.println( c++ ); // prints 5 then postincrements
13        System.out.println( c ); // prints 6
14
15
16        System.out.println(); // skip a line
17
18        // demonstrate prefix increment operator
19        c = 5; // assign 5 to c
20        System.out.println( c ); // prints 5
21        System.out.println( ++c ); // preincrements then prints 6
22        System.out.println( c ); // prints 6
23    } // end main
24 } // end class Increment
```

Annotations:

- Line 13: A callout box with an arrow pointing to the code highlights the line `System.out.println(c++);`. The text inside the box reads: "Uses current value, then increments c".
- Line 21: A callout box with an arrow pointing to the code highlights the line `System.out.println(++c);`. The text inside the box reads: "Increments c then uses new value".

Fig. 4.15 | Preincrementing and postincrementing. (Part 1 of 2.)



5
5
6

5
6
6

Fig. 4.15 | Preincrementing and postincrementing. (Part 2 of 2.)



Common Programming Error 4.7

Attempting to use the increment or decrement operator on an expression other than one to which a value can be assigned is a syntax error. For example, writing `++(x + 1)` is a syntax error, because `(x + 1)` is not a variable.



Operators	Associativity				Type
<code>++</code>	<code>--</code>				right to left unary postfix
<code>++</code>	<code>--</code>	<code>+</code>	<code>-</code>	<code>(type)</code>	right to left unary prefix
<code>*</code>	<code>/</code>	<code>%</code>			left to right multiplicative
<code>+</code>	<code>-</code>				left to right additive
<code><</code>	<code><=</code>	<code>></code>	<code>>=</code>		left to right relational
<code>==</code>	<code>!=</code>				left to right equality
<code>?:</code>					right to left conditional
<code>=</code>	<code>+=</code>	<code>--</code>	<code>*=</code>	<code>/=</code>	<code>%=</code> right to left assignment

Fig. 4.16 | Precedence and associativity of the operators discussed so far.



Control Statements: Part I



OBJECTIVES

In this chapter you'll learn:

- Basic problem-solving techniques.
- To develop algorithms through the process of top-down, stepwise refinement.
- To use the `if` and `if...else` selection statements to choose among alternative actions.
- To use the `while` repetition statement to execute statements in a program repeatedly.
- To use counter-controlled repetition and sentinel-controlled repetition.
- To use the compound assignment, increment and decrement operators.
- The portability of primitive data types.



-
- 4.1** Introduction
 - 4.2** Algorithms
 - 4.3** Pseudocode
 - 4.4** Control Structures
 - 4.5** `if` Single-Selection Statement
 - 4.6** `if...else` Double-Selection Statement
 - 4.7** `while` Repetition Statement
 - 4.8** Formulating Algorithms: Counter-Controlled Repetition
 - 4.9** Formulating Algorithms: Sentinel-Controlled Repetition
 - 4.10** Formulating Algorithms: Nested Control Statements
 - 4.11** Compound Assignment Operators
 - 4.12** Increment and Decrement Operators
 - 4.13** Primitive Types
 - 4.14** (Optional) GUI and Graphics Case Study: Creating Simple Drawings
 - 4.15** Wrap-Up
-



4.1 Introduction

- ▶ Before writing a program to solve a problem, have a thorough understanding of the problem and a carefully planned approach to solving it.
- ▶ Understand the types of building blocks that are available and employ proven program-construction techniques.
- ▶ This chapter introduces
 - The `if`, `if...else` and `while` statements
 - Compound assignment, increment and decrement operators
 - Portability of Java's primitive types



4.2 Algorithms

- ▶ Any computing problem can be solved by executing a series of actions in a specific order.
- ▶ An **algorithm** is a procedure for solving a problem in terms of
 - the **actions** to execute and
 - the **order** in which these actions execute
- ▶ The “rise-and-shine algorithm” followed by one executive for getting out of bed and going to work:
 - (1) Get out of bed; (2) take off pajamas; (3) take a shower; (4) get dressed; (5) eat breakfast; (6) carpool to work.
- ▶ Suppose that the same steps are performed in a slightly different order:
 - (1) Get out of bed; (2) take off pajamas; (3) get dressed; (4) take a shower; (5) eat breakfast; (6) carpool to work.
- ▶ Specifying the order in which statements (actions) execute in a program is called **program control**.



4.3 Pseudocode

- ▶ **Pseudocode** is an informal language that helps you develop algorithms without having to worry about the strict details of Java language syntax.
- ▶ Particularly useful for developing algorithms that will be converted to structured portions of Java programs.
- ▶ Similar to everyday English.
- ▶ Helps you “think out” a program before attempting to write it in a programming language, such as Java.
- ▶ You can type pseudocode conveniently, using any text-editor program.
- ▶ Carefully prepared pseudocode can easily be converted to a corresponding Java program.
- ▶ Pseudocode normally describes only statements representing the actions that occur after you convert a program from pseudocode to Java and the program is run on a computer.
 - e.g., input, output or calculations.



4.4 Control Structures

- ▶ **Sequential execution:** Statements in a program execute one after the other in the order in which they are written.
- ▶ **Transfer of control:** Various Java statements, enable you to specify that the next statement to execute is not necessarily the next one in sequence.
- ▶ **Bohm and Jacopini**
 - Demonstrated that programs could be written *without any goto statements*.
 - All programs can be written in terms of only three control structures—the **sequence structure**, the **selection structure** and the **repetition structure**.
- ▶ When we introduce Java’s control structure implementations, we’ll refer to them in the terminology of the *Java Language Specification* as “*control statements*.”



4.4 Control Structures (Cont.)

▶ Sequence structure

- Built into Java.
- Unless directed otherwise, the computer executes Java statements one after the other in the order in which they're written.
- The [activity diagram](#) in Fig. 4.1 illustrates a typical sequence structure in which two calculations are performed in order.
- Java lets you have as many actions as you want in a sequence structure.
- Anywhere a single action may be placed, we may place several actions in sequence.

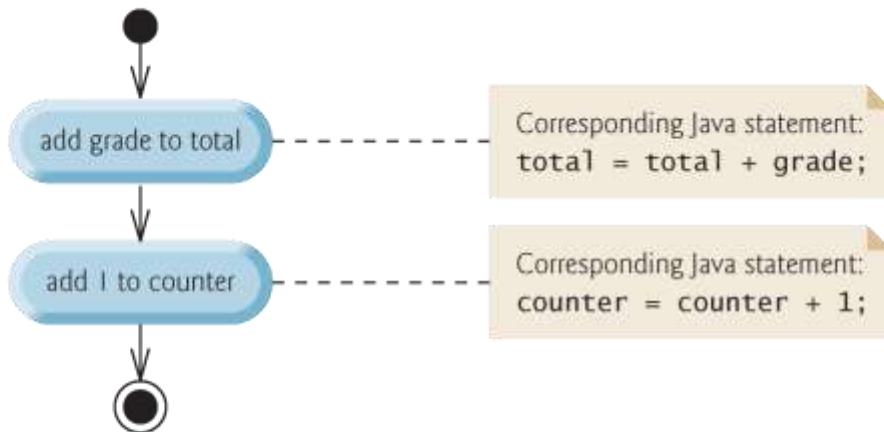


Fig. 4.1 | Sequence structure activity diagram.



4.4 Control Structures (Cont.)

- ▶ UML activity diagram
- ▶ Models the **workflow** (also called the **activity**) of a portion of a software system.
- ▶ May include a portion of an algorithm, like the sequence structure in Fig. 4.1.
- ▶ Composed of symbols
 - **action-state symbols** (rectangles with their left and right sides replaced with outward arcs)
 - **diamonds**
 - **small circles**
- ▶ Symbols connected by **transition arrows**, which represent the flow of the activity—the order in which the actions should occur.
- ▶ Help you develop and represent algorithms.
- ▶ Clearly show how control structures operate.



4.4 Control Structures (Cont.)

- ▶ Sequence structure activity diagram in Fig. 4.1.
- ▶ Two **action states** that represent actions to perform.
- ▶ Each contains an **action expression** that specifies a particular action to perform.
- ▶ Arrows represent **transitions** (order in which the actions represented by the action states occur).
- ▶ **Solid circle** at the top represents the **initial state**—the beginning of the workflow before the program performs the modeled actions.
- ▶ **Solid circle surrounded by a hollow circle** at the bottom represents the **final state**—the end of the workflow after the program performs its actions.



4.4 Control Structures (Cont.)

- ▶ **UML notes**
 - Like comments in Java.
 - Rectangles with the upper-right corners folded over.
 - **Dotted line** connects each note with the element it describes.
 - Activity diagrams normally do not show the Java code that implements the activity. We do this here to illustrate how the diagram relates to Java code.
- ▶ **More information on the UML**
 - see our optional case study (Chapters 12–13)
 - visit www.uml.org



4.4 Control Structures (Cont.)

- ▶ Three types of **selection statements**.
- ▶ **if** statement:
 - Performs an action, if a condition is true; skips it, if false.
 - **Single-selection statement**—selects or ignores a single action (or group of actions).
- ▶ **if...else** statement:
 - Performs an action if a condition is true and performs a different action if the condition is false.
 - **Double-selection statement**—selects between two different actions (or groups of actions).
- ▶ **switch** statement
 - Performs one of several actions, based on the value of an expression.
 - **Multiple-selection statement**—selects among many different actions (or groups of actions).



4.4 Control Structures (Cont.)

- ▶ Three **repetition statements** (also called **looping statements**)
 - Perform statements repeatedly while a **loop-continuation condition** remains true.
- ▶ **while** and **for** statements perform the action(s) in their bodies zero or more times
 - if the loop-continuation condition is initially false, the body will not execute.
- ▶ The **do...while** statement performs the action(s) in its body one or more times.
- ▶ **if, else, switch, while, do** and **for** are keywords.
 - Appendix C: Complete list of Java keywords.



4.4 Control Structures (Cont.)

- ▶ Every program is formed by combining the sequence statement, selection statements (three types) and repetition statements (three types) as appropriate for the algorithm the program implements.
- ▶ Can model each control statement as an activity diagram.
 - Initial state and a final state represent a control statement's entry point and exit point, respectively.
 - Single-entry/single-exit control statements
 - Control-statement stacking—connect the exit point of one to the entry point of the next.
 - Control-statement nesting—a control statement inside another.



4.5 if Single-Selection Statement

- ▶ Pseudocode

*If student's grade is greater than or equal to 60
Print "Passed"*

- ▶ If the condition is false, the Print statement is ignored, and the next pseudocode statement in order is performed.
- ▶ Indentation
 - Optional, but recommended
 - Emphasizes the inherent structure of structured programs
- ▶ The preceding pseudocode *If* in Java:

```
if ( studentGrade >= 60 )
    System.out.println( "Passed" );
```
- ▶ Corresponds closely to the pseudocode.



4.5 if Single-Selection Statement (Cont.)

- ▶ Figure 4.2 `if` statement UML activity diagram.
- ▶ Diamond, or **decision symbol**, indicates that a decision is to be made.
- ▶ Workflow continues along a path determined by the symbol's **guard conditions**, which can be true or false.
- ▶ Each transition arrow emerging from a decision symbol has a guard condition (in square brackets next to the arrow).
- ▶ If a guard condition is true, the workflow enters the action state to which the transition arrow points.

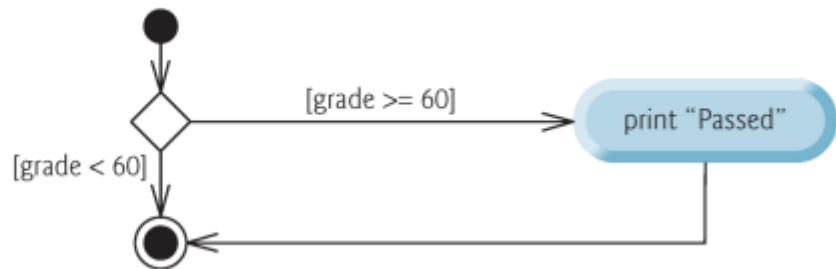


Fig. 4.2 | if single-selection statement UML activity diagram.



4.6 if...else Double-Selection Statement

- ▶ **if...else double-selection statement**—specify an action to perform when the condition is true and a different action when the condition is false.
- ▶ Pseudocode

```
If student's grade is greater than or equal to 60
    Print "Passed"
Else
    Print "Failed"
```

- ▶ The preceding *If...Else pseudocode statement in Java:*

```
if ( grade >= 60 )
    System.out.println( "Passed" );
else
    System.out.println( "Failed" );
```
- ▶ Note that the body of the **else** is also indented.



Good Programming Practice 4.1

*Indent both body statements of an `if...else` statement.
Many IDEs do this for you.*



Good Programming Practice 4.2

If there are several levels of indentation, each level should be indented the same additional amount of space.



4.6 if...else Double-Selection Statement (Cont.)

- ▶ Figure 4.3 illustrates the flow of control in the `if...else` statement.
- ▶ The symbols in the UML activity diagram (besides the initial state, transition arrows and final state) represent action states and decisions.

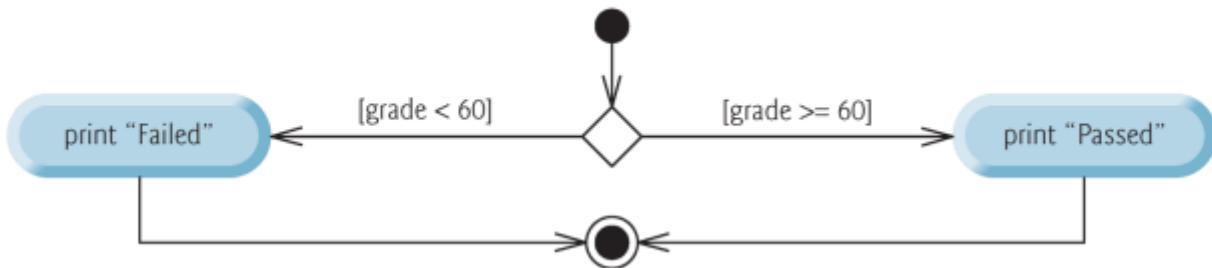


Fig. 4.3 | if...else double-selection statement UML activity diagram.



4.6 if...else Double-Selection Statement (Cont.)

- ▶ Conditional operator (`?:`)—shorthand `if...else`.
- ▶ Ternary operator (takes three operands)
- ▶ Operands and `? :` form a **conditional expression**
- ▶ Operand to the left of the `?` is a **boolean expression**—evaluates to a **boolean** value (**true** or **false**)
- ▶ Second operand (between the `?` and `:`) is the value if the **boolean** expression is **true**
- ▶ Third operand (to the right of the `:`) is the value if the **boolean** expression evaluates to **false**.



4.6 if...else Double-Selection Statement (Cont.)

- ▶ Example:

```
System.out.println(  
    studentGrade >= 60 ? "Passed" : "Failed" );
```

- ▶ Evaluates to the string "Passed" if the boolean expression `studentGrade >= 60` is true and to the string "Failed" if it is false.



Good Programming Practice 4.3

Conditional expressions are more difficult to read than if...else statements and should be used to replace only simple if...else statements that choose between two values.



4.6 if...else Double-Selection Statement (Cont.)

- ▶ Can test multiple cases by placing `if...else` statements inside other `if...else` statements to create *nested if...else statements*.
- ▶ Pseudocode:

```
If student's grade is greater than or equal to 90
    Print "A"
else
    If student's grade is greater than or equal to 80
        Print "B"
    else
        If student's grade is greater than or equal to 70
            Print "C"
        else
            If student's grade is greater than or equal to 60
                Print "D"
            else
                Print "F"
```



4.6 if...else Double-Selection Statement (Cont.)

- ▶ This pseudocode may be written in Java as

```
if ( studentGrade >= 90 )
    System.out.println( "A" );
else
    if ( studentGrade >= 80 )
        System.out.println( "B" );
    else
        if ( studentGrade >= 70 )
            System.out.println( "C" );
        else
            if ( studentGrade >= 60 )
                System.out.println( "D" );
            else
                System.out.println( "F" );
```

- ▶ If **studentGrade** ≥ 90 , the first four conditions will be true, but only the statement in the **if** part of the first **if...else** statement will execute. After that, the **else** part of the “outermost” **if...else** statement is skipped.



4.6 if...else Double-Selection Statement (Cont.)

- ▶ Most Java programmers prefer to write the preceding nested if...else statement as

```
if ( studentGrade >= 90 )
    System.out.println( "A" );
else if ( studentGrade >= 80 )
    System.out.println( "B" );
else if ( studentGrade >= 70 )
    System.out.println( "C" );
else if ( studentGrade >= 60 )
    System.out.println( "D" );
else
    System.out.println( "F" );
```

- ▶ The two forms are identical except for the spacing and indentation, which the compiler ignores.



4.6 if...else Double-Selection Statement (Cont.)

- ▶ The Java compiler always associates an `else` with the immediately preceding `if` unless told to do otherwise by the placement of braces (`{` and `}`).
- ▶ Referred to as the **dangling-else problem**.
- ▶ The following code is not what it appears:

```
if ( x > 5 )
    if ( y > 5 )
        System.out.println( "x and y are > 5" );
else
    System.out.println( "x is <= 5" );
```

- ▶ Beware! This nested `if...else` statement does not execute as it appears. The compiler actually interprets the statement as

```
if ( x > 5 )
    if ( y > 5 )
        System.out.println( "x and y are > 5" );
else
    System.out.println( "x is <= 5" );
```



4.6 if...else Double-Selection Statement (Cont.)

- To force the nested `if...else` statement to execute as it was originally intended, we must write it as follows:

```
if ( x > 5 )
{
    if ( y > 5 )
        System.out.println( "x and y are > 5" );
}
else
    System.out.println( "x is <= 5" );
```

- The braces indicate that the second `if` is in the body of the first and that the `else` is associated with the *first if*.
- Exercises 4.27–4.28 investigate the dangling-`else` problem further.



4.6 if...else Double-Selection Statement (Cont.)

- ▶ The **if** statement normally expects only one statement in its body.
- ▶ To include several statements in the body of an **if** (or the body of an **else** for an **if...else** statement), enclose the statements in braces.
- ▶ Statements contained in a pair of braces form a **block**.
- ▶ A block can be placed anywhere that a single statement can be placed.
- ▶ Example: A block in the **else** part of an **if...else** statement:

```
if ( grade >= 60 )
    System.out.println("Passed");
else
{
    System.out.println("Failed");
    System.out.println("You must take this course again.");
}
```



4.6 if...else Double-Selection Statement (Cont.)

- ▶ Syntax errors (e.g., when one brace in a block is left out of the program) are caught by the compiler.
- ▶ A **logic error** (e.g., when both braces in a block are left out of the program) has its effect at execution time.
- ▶ A **fatal logic error** causes a program to fail and terminate prematurely.
- ▶ A **nonfatal logic error** allows a program to continue executing but causes it to produce incorrect results.



4.6 if...else Double-Selection Statement (Cont.)

- ▶ Just as a block can be placed anywhere a single statement can be placed, it's also possible to have an empty statement.
- ▶ The empty statement is represented by placing a semicolon (;) where a statement would normally be.



Common Programming Error 4.1

Placing a semicolon after the condition in an `if` or `if...else` statement leads to a logic error in single-selection `if` statements and a syntax error in double-selection `if...else` statements (when the `if`-part contains an actual body statement).



4.7 while Repetition Statement

- ▶ Repetition statement—repeats an action while a condition remains true.
- ▶ Pseudocode

While there are more items on my shopping list

Purchase next item and cross it off my list

- ▶ The repetition statement's body may be a single statement or a block.
- ▶ Eventually, the condition will become false. At this point, the repetition terminates, and the first statement after the repetition statement executes.



4.7 while Repetition Statement (Cont.)

- ▶ Example of Java's **while** repetition statement: find the first power of 3 larger than 100. Assume **int** variable **product** is initialized to 3.

```
while ( product <= 100 )  
    product = 3 * product;
```

- ▶ Each iteration multiplies **product** by 3, so **product** takes on the values 9, 27, 81 and 243 successively.
- ▶ When variable **product** becomes 243, the **while**-statement condition—**product** \leq 100—becomes false.
- ▶ Repetition terminates. The final value of **product** is 243.
- ▶ Program execution continues with the next statement after the **while** statement.



Common Programming Error 4.2

*Not providing in the body of a `while` statement an action that eventually causes the condition in the `while` to become false normally results in a logic error called an **infinite loop** (the loop never terminates).*



4.7 while Repetition Statement (Cont.)

- ▶ The UML activity diagram in Fig. 4.4 illustrates the flow of control in the preceding `while` statement.
- ▶ The UML represents both the **merge symbol** and the decision symbol as diamonds.
- ▶ The merge symbol joins two flows of activity into one.



4.7 while Repetition Statement (Cont.)

- ▶ The decision and merge symbols can be distinguished by the number of “incoming” and “outgoing” transition arrows.
 - A decision symbol has one transition arrow pointing to the diamond and two or more pointing out from it to indicate possible transitions from that point. Each transition arrow pointing out of a decision symbol has a guard condition next to it.
 - A merge symbol has two or more transition arrows pointing to the diamond and only one pointing from the diamond, to indicate multiple activity flows merging to continue the activity. None of the transition arrows associated with a merge symbol has a guard condition.

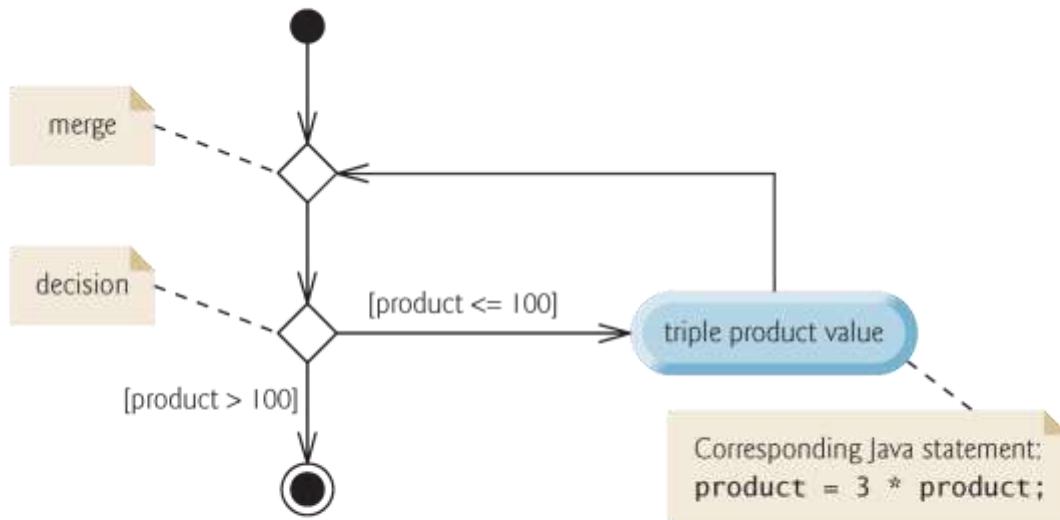


Fig. 4.4 | while repetition statement UML activity diagram.



4.8 Formulating Algorithms: Counter-Controlled Repetition

- ▶ A class of ten students took a quiz. The grades (integers in the range 0 to 100) for this quiz are available to you. Determine the class average on the quiz.
- ▶ The class average is equal to the sum of the grades divided by the number of students.
- ▶ The algorithm for solving this problem on a computer must input each grade, keep track of the total of all grades input, perform the averaging calculation and print the result.
- ▶ Use **counter-controlled repetition** to input the grades one at a time.
- ▶ A variable called a **counter** (or **control variable**) controls the number of times a set of statements will execute.
- ▶ Counter-controlled repetition is often called **definite repetition**, because the number of repetitions is known before the loop begins executing.



Software Engineering Observation 4.1

Experience has shown that the most difficult part of solving a problem on a computer is developing the algorithm for the solution. Once a correct algorithm has been specified, producing a working Java program from the algorithm is usually straightforward.



4.8 Formulating Algorithms: Counter-Controlled Repetition (Cont.)

- ▶ A **total** is a variable used to accumulate the sum of several values.
- ▶ A **counter** is a variable used to count.
- ▶ Variables used to store totals are normally initialized to zero before being used in a program.



4.8 Formulating Algorithms: Counter-Controlled Repetition (Cont.)

- ▶ Variables declared in a method body are local variables and can be used only from the line of their declaration to the closing right brace of the method declaration.
- ▶ A local variable's declaration must appear before the variable is used in that method.
- ▶ A local variable cannot be accessed outside the method in which it's declared.



4.9 Formulating Algorithms: Sentinel-Controlled Repetition

- ▶ *Develop a class-averaging program that processes grades for an arbitrary number of students each time it is run.*
- ▶ **Sentinel-controlled repetition** is often called **indefinite repetition** because the number of repetitions is not known before the loop begins executing.
- ▶ A special value called a **sentinel value** (also called a **signal value**, a **dummy value** or a **flag value**) can be used to indicate “end of data entry.”
- ▶ A sentinel value must be chosen that cannot be confused with an acceptable input value.



4.9 Formulating Algorithms: Sentinel-Controlled Repetition (Cont.)

- ▶ Program logic for sentinel-controlled repetition
 - Reads the first value before reaching the `while`.
 - This value determines whether the program's flow of control should enter the body of the `while`. If the condition of the `while` is false, the user entered the sentinel value, so the body of the `while` does not execute (i.e., no grades were entered).
 - If the condition is true, the body begins execution and processes the input.
 - Then the loop body inputs the next value from the user before the end of the loop.



Control Statements: Part 2



OBJECTIVES

In this Chapter you'll learn:

- The essentials of counter-controlled repetition.
- To use the **for** and **do...while** repetition statements to execute statements in a program repeatedly.
- To understand multiple selection using the **switch** selection statement.
- To use the **break** and **continue** program control statements to alter the flow of control.
- To use the logical operators to form complex conditional expressions in control statements.



-
- 5.1** Introduction
 - 5.2** Essentials of Counter-Controlled Repetition
 - 5.3** `for` Repetition Statement
 - 5.4** Examples Using the `for` Statement
 - 5.5** `do...while` Repetition Statement
 - 5.6** `switch` Multiple-Selection Statement
 - 5.7** `break` and `continue` Statements
 - 5.8** Logical Operators
 - 5.9** Structured Programming Summary
 - 5.10** (Optional) GUI and Graphics Case Study: Drawing Rectangles and Ovals
 - 5.11** Wrap-Up
-



5.1 Introduction

- ▶ **for** repetition statement
- ▶ **do...while** repetition statement
- ▶ **switch** multiple-selection statement
- ▶ **break** statement
- ▶ **continue** statement
- ▶ Logical operators
- ▶ Control statements summary.



5.2 Essentials of Counter-Controlled Repetition

- ▶ Counter-controlled repetition requires
 - a **control variable** (or loop counter)
 - the **initial value** of the control variable
 - the **increment** (or **decrement**) by which the control variable is modified each time through the loop (also known as **each iteration of the loop**)
 - the **loop-continuation condition** that determines if looping should continue.



```
1 // Fig. 5.1: WhileCounter.java
2 // Counter-controlled repetition with the while repetition statement.
3
4 public class WhileCounter
{
    public static void main( String[] args )
    {
        int counter = 1; // declare and initialize control variable
        ←
        Declares and initializes control variable
        counter to 1
        ←
        while ( counter <= 10 ) // loop-continuation condition
        {
            System.out.printf( "%d ", counter );
            ++counter; // increment control variable by 1
            ←
            Loop-continuation condition tests for
            count's final value
        } // end while
        ←
        initializes gradeCounter to 1;
        indicates first grade about to be input
        System.out.println(); // output a newline
    } // end main
} // end class WhileCounter
```

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

Fig. 5.1 | Counter-controlled repetition with the `while` repetition statement.



5.2 Essentials of Counter-Controlled Repetition (Cont.)

- ▶ In Fig. 5.1, the elements of counter-controlled repetition are defined in lines 8, 10 and 13.
- ▶ Line 8 declares the control variable (**counter**) as an **int**, reserves space for it in memory and sets its initial value to 1.
- ▶ The loop-continuation condition in the **while** (line 10) tests whether the value of the control variable is less than or equal to 10 (the final value for which the condition is **true**).
- ▶ Line 13 increments the control variable by 1 for each iteration of the loop.



Common Programming Error 5.1

Because floating-point values may be approximate, controlling loops with floating-point variables may result in imprecise counter values and inaccurate termination tests.



Error-Prevention Tip 5.1

Use integers to control counting loops.



Software Engineering Observation 5.1

“Keep it simple” is good advice for most of the code you’ll write.

5.3 for Repetition Statement

▶ **for repetition statement**

- Specifies the counter-controlled-repetition details in a single line of code.
- Figure 5.2 reimplements the application of Fig. 5.1 using **for**.



```
1 // Fig. 5.2: ForCounter.java
2 // Counter-controlled repetition with the for repetition statement.
3
4 public class ForCounter
{
5     public static void main( String[] args )
6     {
7         // for statement header includes initialization,
8         // loop-continuation condition and increment
9         for ( int counter = 1; counter <= 10; counter++ )
10            System.out.printf( "%d ", counter );
11
12            System.out.println(); // output a newline
13        } // end main
14    } // end class ForCounter
```

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

For statement's header contains everything you need for counter-controlled repetition

Fig. 5.2 | Counter-controlled repetition with the for repetition statement.



5.3 for Repetition Statement (Cont.)

- ▶ When the **for** statement begins executing, the control variable is declared and initialized.
- ▶ Next, the program checks the loop-continuation condition, which is between the two required semicolons.
- ▶ If the condition initially is true, the body statement executes.
- ▶ After executing the loop's body, the program increments the control variable in the increment expression, which appears to the right of the second semicolon.
- ▶ Then the loop-continuation test is performed again to determine whether the program should continue with the next iteration of the loop.
- ▶ A common logic error with counter-controlled repetition is an **off-by-one error**.



Common Programming Error 5.2

Using an incorrect relational operator or an incorrect final value of a loop counter in the loop-continuation condition of a repetition statement can cause an off-by-one error.



Error-Prevention Tip 5.2

Using the final value in the condition of a while or for statement and using the `<=` relational operator helps avoid off-by-one errors. For a loop that prints the values 1 to 10, the loop-continuation condition should be `counter <= 10` rather than `counter < 10` (which causes an off-by-one error) or `counter < 11` (which is correct). Many programmers prefer so-called zero-based counting, in which to count 10 times, `counter` would be initialized to zero and the loop-continuation test would be `counter < 10`.

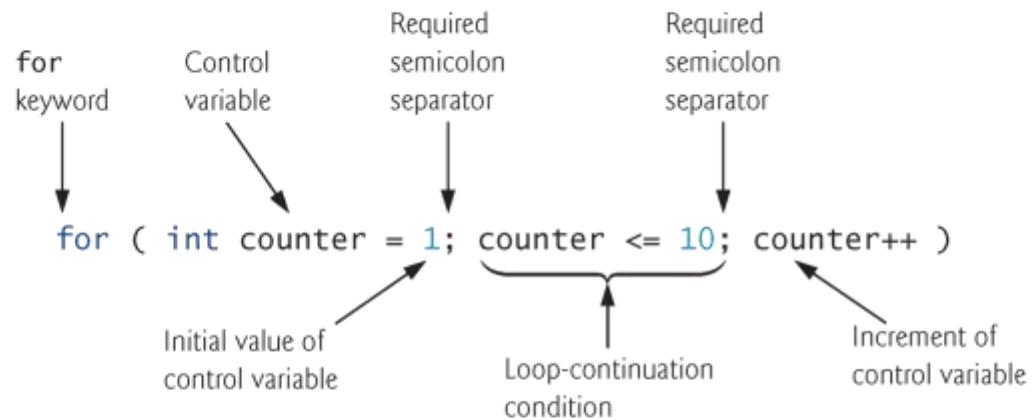


Fig. 5.3 | `for` statement header components.



5.3 for Repetition Statement (Cont.)

- ▶ The general format of the **for** statement is

```
for (initialization; loopContinuationCondition; increment  
      )  
      statement
```

- the *initialization* expression names the loop's control variable and optionally provides its initial value
- *loopContinuationCondition* determines whether the loop should continue executing
- *increment* modifies the control variable's value (possibly an increment or decrement), so that the loop-continuation condition eventually becomes false.
- ▶ The two semicolons in the **for** header are required.



5.3 for Repetition Statement (Cont.)

- ▶ In most cases, the **for** statement can be represented with an equivalent **while** statement as follows:

```
initialization;
while ( loopContinuationCondition )
{
    statement
    increment;
}
```

- ▶ Typically, **for** statements are used for counter-controlled repetition and **while** statements for sentinel-controlled repetition.
- ▶ If the *initialization* expression in the **for** header declares the control variable, the control variable can be used only in that **for** statement.



5.3 for Repetition Statement (Cont.)

- ▶ A variable's **scope** defines where it can be used in a program.
 - A local variable can be used only in the method that declares it and only from the point of declaration through the end of the method.



Common Programming Error 5.3

When a `for` statement's control variable is declared in the initialization section of the `for`'s header, using the control variable after the `for`'s body is a compilation error.



5.3 for Repetition Statement (Cont.)

- ▶ All three expressions in a **for** header are optional.
 - If the *loopContinuationCondition* is omitted, the condition is always true, thus creating an infinite loop.
 - You might omit the *initialization* expression if the program initializes the control variable before the loop.
 - You might omit the *increment* if the program calculates it with statements in the loop's body or if no increment is needed.
- ▶ The increment expression in a **for** acts as if it were a standalone statement at the end of the **for**'s body, so

counter = counter + 1

counter += 1

++counter

counter++

are equivalent increment expressions in a **for** statement.



Common Programming Error 5.4

*Placing a semicolon immediately to the right of the right parenthesis of a **for** header makes that **for**'s body an empty statement. This is normally a logic error.*



Error-Prevention Tip 5.3

Infinite loops occur when the loop-continuation condition in a repetition statement never becomes false. To prevent this situation in a counter-controlled loop, ensure that the control variable is incremented (or decremented) during each iteration of the loop. In a sentinel-controlled loop, ensure that the sentinel value is able to be input.



5.3 for Repetition Statement (Cont.)

- ▶ The initialization, loop-continuation condition and increment can contain arithmetic expressions.
- ▶ For example, assume that $x = 2$ and $y = 10$. If x and y are not modified in the body of the loop, the statement

```
for (int j = x; j <= 4 * x * y; j += y / x)
```
- ▶ is equivalent to the statement

```
for (int j = 2; j <= 80; j += 5)
```
- ▶ The increment of a **for** statement may be negative, in which case it's a decrement, and the loop counts downward.



Error-Prevention Tip 5.4

Although the value of the control variable can be changed in the body of a for loop, avoid doing so, because this practice can lead to subtle errors.

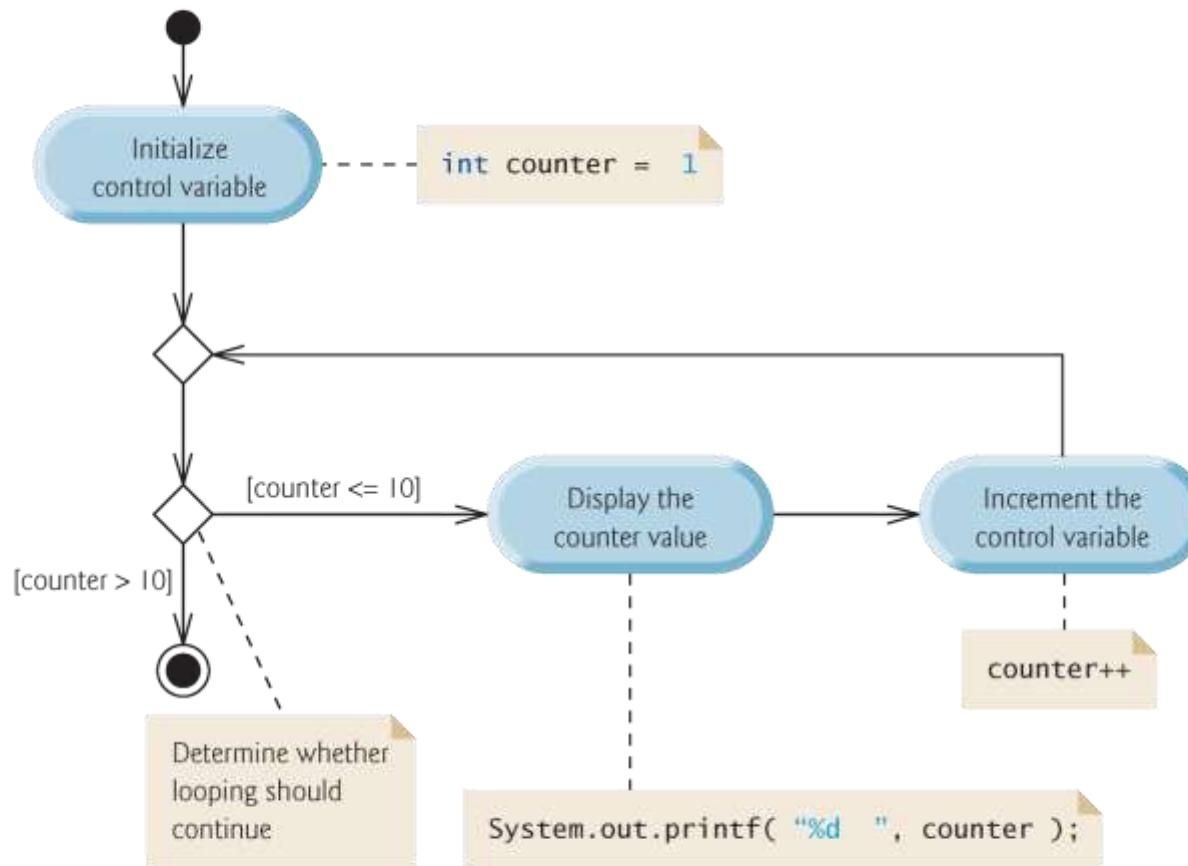


Fig. 5.4 | UML activity diagram for the `for` statement in Fig. 5.2.



5.4 Examples Using the for Statement

- ▶ a) Vary the control variable from 1 to 100 in increments of 1.

```
for ( int i = 1; i <= 100; i++ )
```

- ▶ b) Vary the control variable from 100 to 1 in decrements of 1.

```
for ( int i = 100; i >= 1; i-- )
```

- ▶ c) Vary the control variable from 7 to 77 in increments of 7.

```
for ( int i = 7; i <= 77; i += 7 )
```



5.4 Examples Using the for Statement (Cont.)

- ▶ d) Vary the control variable from 20 to 2 in decrements of 2.

```
for ( int i = 20; i >= 2; i -= 2 )
```

- ▶ e) Vary the control variable over the values 2, 5, 8, 11, 14, 17, 20.

```
for ( int i = 2; i <= 20; i += 3 )
```

- ▶ f) Vary the control variable over the values 99, 88, 77, 66, 55, 44, 33, 22, 11, 0.

```
for ( int i = 99; i >= 0; i -= 11 )
```



Common Programming Error 5.5

Using an incorrect relational operator in the loop-continuation condition of a loop that counts downward (e.g., using $i \leq 1$ instead of $i \geq 1$ in a loop counting down to 1) is usually a logic error.



```
1 // Fig. 5.5: Sum.java
2 // Summing integers with the for statement.
3
4 public class Sum
5 {
6     public static void main( String[] args )
7     {
8         int total = 0; // initialize total
9
10        // total even integers from 2 through 20
11        for ( int number = 2; number <= 20; number += 2 )
12            total += number;
13
14        System.out.printf( "Sum is %d\n", total ); // display results
15    } // end main
16 } // end class Sum
```

Sum is 110

Fig. 5.5 | Summing integers with the for statement.



5.4 Examples Using the `for` Statement (Cont.)

- ▶ The *initialization* and *increment* expressions can be comma-separated lists that enable you to use multiple initialization expressions or multiple increment expressions.
- ▶ Although this is discouraged, the body of the `for` statement in lines 11–12 of Fig. 5.5 could be merged into the increment portion of the `for` header by using a comma as follows:

```
for ( int number = 2;  
      number <= 20;  
      total += number, number += 2 )  
; // empty statement
```



Good Programming Practice 5.1

For readability limit the size of control-statement headers to a single line if possible.



5.4 Examples Using the for Statement (Cont.)

- ▶ Compound interest application
- ▶ A person invests \$1000 in a savings account yielding 5% interest. Assuming that all the interest is left on deposit, calculate and print the amount of money in the account at the end of each year for 10 years. Use the following formula to determine the amounts:

$$a = p (1 + r)^n$$

where

p is the original amount invested (i.e., the principal)

r is the annual interest rate (e.g., use 0.05 for 5%)

n is the number of years

a is the amount on deposit at the end of the *n*th year.



5.4 Examples Using the `for` Statement (Cont.)

- ▶ The solution to this problem (Fig. 5.6) involves a loop that performs the indicated calculation for each of the 10 years the money remains on deposit.
- ▶ Java treats floating-point constants like `1000.0` and `0.05` as type `double`.
- ▶ Java treats whole-number constants like `7` and `-22` as type `int`.



```
1 // Fig. 5.6: Interest.java
2 // Compound-interest calculations with for.
3
4 public class Interest
{
    public static void main( String[] args )
    {
        double amount; // amount on deposit at end of each year
        double principal = 1000.0; // initial amount before interest
        double rate = 0.05; // interest rate
        // display headers
        System.out.printf( "%s%20s\n", "Year", "Amount on deposit" );
        // calculate amount on deposit for each of ten years
        for ( int year = 1; year <= 10; year++ )
        {
            // calculate new amount for specified year
            amount = principal * Math.pow( 1.0 + rate, year );
    }
```

Java treats floating-point literals as double values

Uses static method Math.pow to help calculate the amount on deposit

Fig. 5.6 | Compound-interest calculations with for. (Part I of 2.)



```
21     // display the year and the amount
22     System.out.printf( "%4d%,20.2f\n", year, amount );
23 } // end for
24 } // end main
25 } // end class Interest
```

Comma in format specifier indicates
that large numbers should be displayed
with thousands separators

Year	Amount on deposit
1	1,050.00
2	1,102.50
3	1,157.63
4	1,215.51
5	1,276.28
6	1,340.10
7	1,407.10
8	1,477.46
9	1,551.33
10	1,628.89

Fig. 5.6 | Compound-interest calculations with `for`. (Part 2 of 2.)



5.4 Examples Using the for Statement (Cont.)

- ▶ In the format specifier `%20s`, the integer 20 between the % and the conversion character `s` indicates that the value output should be displayed with a **field width** of 20—that is, `printf` displays the value with at least 20 character positions.
- ▶ If the value to be output is less than 20 character positions wide, the value is **right justified** in the field by default.
- ▶ If the `year` value to be output were more than has more characters than the field width, the field width would be extended to the right to accommodate the entire value.
- ▶ To indicate that values should be output **left justified**, precede the field width with the **minus sign (-) formatting flag** (e.g., `%-20s`).



5.4 Examples Using the for Statement (Cont.)

- ▶ Classes provide methods that perform common tasks on objects.
- ▶ Most methods must be called on a specific object.
- ▶ Many classes also provide methods that perform common tasks and do not require objects. These are called **static** methods.
- ▶ Java does not include an exponentiation operator—**Math** class **static** method **pow** can be used for raising a value to a power.
- ▶ You can call a **static** method by specifying the class name followed by a dot (.) and the method name, as in
 - *ClassName.methodName(arguments)*
- ▶ **Math.pow(x, y)** calculates the value of x raised to the yth power. The method receives two **double** arguments and returns a **double** value.



Performance Tip 5.1

In loops, avoid calculations for which the result never changes—such calculations should typically be placed before the loop. Many of today's sophisticated optimizing compilers will place such calculations outside loops in the compiled code.



5.4 Examples Using the `for` Statement (Cont.)

- ▶ In the format specifier `%,20.2f`, the **comma (,)** **formatting flag** indicates that the floating-point value should be output with a **grouping separator**.
- ▶ Separator is specific to the user's locale (i.e., country).
- ▶ In the United States, the number will be output using commas to separate every three digits and a decimal point to separate the fractional part of the number, as in `1,234.45`.
- ▶ The number `20` in the format specification indicates that the value should be output right justified in a field width of `20` characters.
- ▶ The `.2` specifies the formatted number's precision—in this case, the number is rounded to the nearest hundredth and output with two digits to the right of the decimal point.



Error-Prevention Tip 5.5

Do not use variables of type `double` (or `float`) to perform precise monetary calculations. The imprecision of floating-point numbers can cause errors. In the exercises, you'll learn how to use integers to perform precise monetary calculations. Java also provides class `java.math.BigDecimal` to perform precise monetary calculations. For more information, see download.oracle.com/javase/6/docs/api/java/math//BigDecimal.html.



5.5 do...while Repetition Statement

- ▶ The **do...while repetition statement** is similar to the **while** statement.
- ▶ In the **while**, the program tests the loop-continuation condition at the beginning of the loop, before executing the loop's body; if the condition is false, the body never executes.
- ▶ The **do...while** statement tests the loop-continuation condition *after executing the loop's body*; therefore, the body always executes at least once.
- ▶ When a **do...while** statement terminates, execution continues with the next statement in sequence.



```
1 // Fig. 5.7: DoWhileTest.java
2 // do...while repetition statement.
3
4 public class DoWhileTest
{
5     public static void main( String[] args )
6     {
7         int counter = 1; // initialize counter
8
9         do
10        {
11            System.out.printf( "%d  ", counter );
12            ++counter;
13        } while ( counter <= 10 ); // end do...while
14
15        System.out.println(); // outputs a newline
16    } // end main
17 } // end class DoWhileTest
```

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

Condition tested at end of loop, so loop always executes at least once

Fig. 5.7 | do...while repetition statement.



5.5 do...while Repetition Statement (Cont.)

- ▶ Figure 5.8 contains the UML activity diagram for the `do...while` statement.
- ▶ The diagram makes it clear that the loop-continuation condition is not evaluated until after the loop performs the action state at least once.

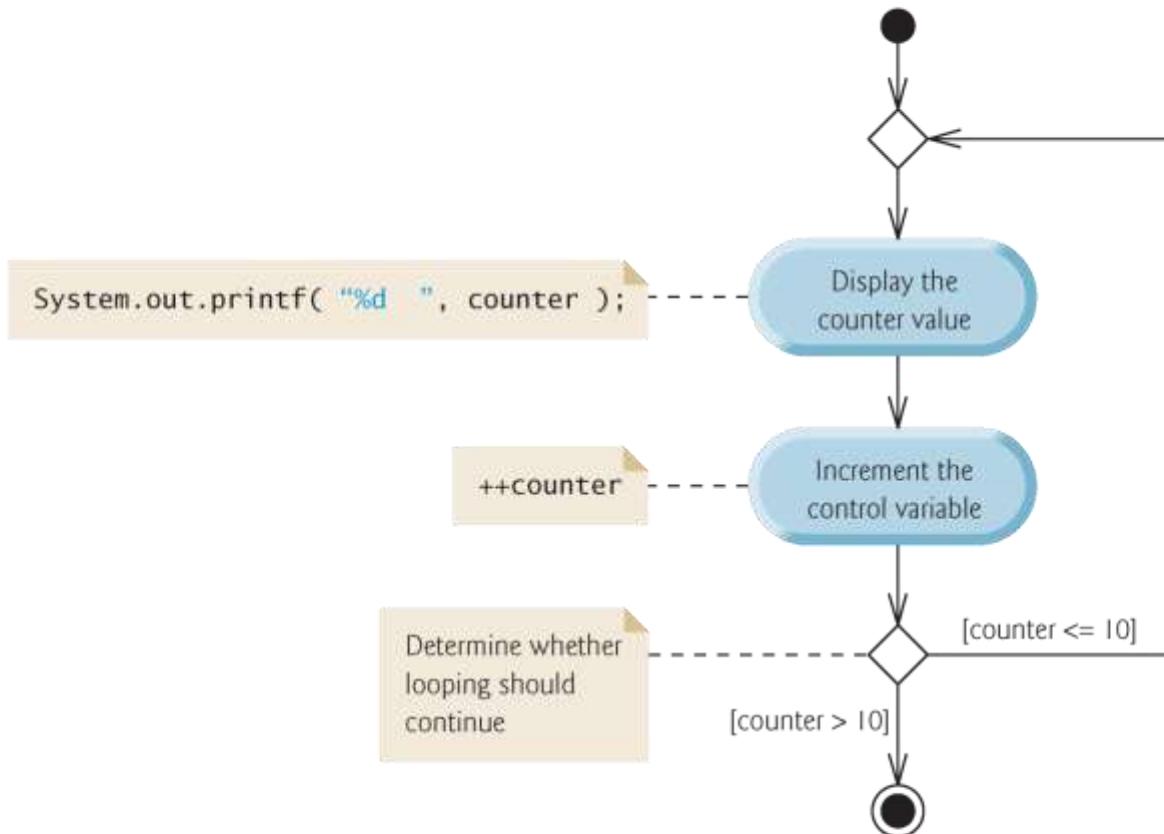


Fig. 5.8 | `do...while` repetition statement UML activity diagram.



5.5 do...while Repetition Statement (Cont.)

- ▶ Braces are not required in the `do...while` repetition statement if there's only one statement in the body.
- ▶ Most programmers include the braces, to avoid confusion between the `while` and `do...while` statements.
- ▶ Thus, the `do...while` statement with one body statement is usually written as follows:
 - `do`
 - `{`
 - statement*
 - `}` `while` (*condition*);



Good Programming Practice 5.2

Always include braces in a do...while statement. This helps eliminate ambiguity between the while statement and a do...while statement containing only one statement.



5.6 switch Multiple-Selection Statement

- ▶ **switch multiple-selection statement** performs different actions based on the possible values of a **constant integral expression** of type **byte**, **short**, **int** or **char**.



```
1 // Fig. 5.9: GradeBook.java
2 // GradeBook class uses switch statement to count letter grades.
3 import java.util.Scanner; // program uses class Scanner
4
5 public class GradeBook
{
6
7     private String courseName; // name of course this GradeBook represents
8     // int instance variables are initialized to 0 by default
9     private int total; // sum of grades
10    private int gradeCounter; // number of grades entered
11    private int aCount; // count of A grades
12    private int bCount; // count of B grades
13    private int cCount; // count of C grades
14    private int dCount; // count of D grades
15    private int fCount; // count of F grades
16
17    // constructor initializes courseName;
18    public GradeBook( String name )
19    {
20        courseName = name; // initializes courseName
21    } // end constructor
22
```

Fig. 5.9 | GradeBook class uses switch statement to count letter grades. (Part I
of 7.)



```
23 // method to set the course name
24 public void setCourseName( String name )
25 {
26     courseName = name; // store the course name
27 } // end method setCourseName
28
29 // method to retrieve the course name
30 public String getCourseName()
31 {
32     return courseName;
33 } // end method getCourseName
34
35 // display a welcome message to the GradeBook user
36 public void displayMessage()
37 {
38     // getCourseName gets the name of the course
39     System.out.printf( "Welcome to the grade book for\n%s!\n\n",
40         getCourseName() );
41 } // end method displayMessage
42
```

Fig. 5.9 | GradeBook class uses switch statement to count letter grades. (Part 2 of 7.)



```
43 // input arbitrary number of grades from user
44 public void inputGrades()
45 {
46     Scanner input = new Scanner( System.in );
47
48     int grade; // grade entered by user
49
50     System.out.printf( "%s\n%s\n    %s\n    %s\n",
51         "Enter the integer grades in the range 0-100.",
52         "Type the end-of-file indicator to terminate input:",
53         "On UNIX/Linux/Mac OS X type <Ctrl> d then press Enter",
54         "On Windows type <Ctrl> z then press Enter" );
55
```

Fig. 5.9 | GradeBook class uses switch statement to count letter grades. (Part 3 of 7.)



```
56     // loop until user enters the end-of-file indicator
57     while ( input.hasNext() )
58     {
59         grade = input.nextInt(); // read grade
60         total += grade; // add grade to total
61         ++gradeCounter; // increment number of grades
62
63         // call method to increment appropriate counter
64         incrementLetterGradeCounter( grade );
65     } // end while
66 } // end method inputGrades
67
```

Fig. 5.9 | GradeBook class uses switch statement to count letter grades. (Part 4 of 7.)



```
68 // add 1 to appropriate counter for specified grade
69 private void incrementLetterGradeCounter( int grade )
70 {
71     // determine which grade was entered
72     switch ( grade / 10 )
73     {
74         case 9: // grade was between 90
75             case 10: // and 100, inclusive
76                 ++aCount; // increment aCount
77                 break; // necessary to exit switch
78
79         case 8: // grade was between 80 and 89
80             ++bCount; // increment bCount
81             break; // exit switch
82
83         case 7: // grade was between 70 and 79
84             ++cCount; // increment cCount
85             break; // exit switch
86
87         case 6: // grade was between 60 and 69
88             ++dCount; // increment dCount
89             break; // exit switch
```

Fig. 5.9 | GradeBook class uses switch statement to count letter grades. (Part 5 of 7.)



```
90
91     default: // grade was less than 60
92         ++fCount; // increment fCount
93         break; // optional; will exit switch anyway
94     } // end switch
95 } // end method incrementLetterGradeCounter
96
97 // display a report based on the grades entered by the user
98 public void displayGradeReport()
99 {
100    System.out.println( "\nGrade Report:" );
101
102    // if user entered at least one grade...
103    if ( gradeCounter != 0 )
104    {
105        // calculate average of all grades entered
106        double average = (double) total / gradeCounter;
107
108        // output summary of results
109        System.out.printf( "Total of the %d grades entered is %d\n",
110                           gradeCounter, total );
111        System.out.printf( "Class average is %.2f\n", average );
```

Fig. 5.9 | GradeBook class uses switch statement to count letter grades. (Part 6 of 7.)



```
112
113     "Number of students who received each grade:",
114     "A: ", aCount,    // display number of A grades
115     "B: ", bCount,    // display number of B grades
116     "C: ", cCount,    // display number of C grades
117     "D: ", dCount,    // display number of D grades
118     "F: ", fCount ); // display number of F grades
119 } // end if
120 else // no grades were entered, so output appropriate message
121     System.out.println( "No grades were entered" );
122 } // end method displayGradeReport
123 } // end class GradeBook
```

Fig. 5.9 | GradeBook class uses switch statement to count letter grades. (Part 7 of 7.)



Portability Tip 5.1

The keystroke combinations for entering end-of-file are system dependent.



Common Programming Error 5.6

Forgetting a break statement when one is needed in a switch is a logic error.



```
1 // Fig. 5.10: GradeBookTest.java
2 // Create GradeBook object, input grades and display grade report.
3
4 public class GradeBookTest
5 {
6     public static void main( String[] args )
7     {
8         // create GradeBook object myGradeBook and
9         // pass course name to constructor
10        GradeBook myGradeBook = new GradeBook(
11            "CS101 Introduction to Java Programming" );
12
13        myGradeBook.displayMessage(); // display welcome message
14        myGradeBook.inputGrades(); // read grades from user
15        myGradeBook.displayGradeReport(); // display report based on grades
16    } // end main
17 } // end class GradeBookTest
```

Fig. 5.10 | Create GradeBook object, input grades and display grade report. (Part I
of 3.)



Welcome to the grade book for
CS101 Introduction to Java Programming!

Enter the integer grades in the range 0-100.

Type the end-of-file indicator to terminate input:

On UNIX/Linux/Mac OS X type <Ctrl> d then press Enter

On Windows type <Ctrl> z then press Enter

```
99
92
45
57
63
71
76
85
90
100
^Z
```

Grade Report:

Total of the 10 grades entered is 778

Class average is 77.80

Fig. 5.10 | Create GradeBook object, input grades and display grade report. (Part 2 of 3.)



Number of students who received each grade:

A: 4
B: 1
C: 2
D: 1
F: 2

Fig. 5.10 | Create GradeBook object, input grades and display grade report. (Part 3 of 3.)



5.6 switch Multiple-Selection Statement (Cont.)

- ▶ The **end-of-file indicator** is a system-dependent keystroke combination which the user enters to indicate that there is no more data to input.
- ▶ On UNIX/Linux/Mac OS X systems, end-of-file is entered by typing the sequence
 - $<Ctrl> d$
- ▶ on a line by itself. This notation means to simultaneously press both the *Ctrl* key and the *d* key.
- ▶ On Windows systems, end-of-file can be entered by typing
 - $<Ctrl> z$
- ▶ On some systems, you must press *Enter* after typing the end-of-file key sequence.
- ▶ Windows typically displays the characters $\wedge Z$ on the screen when the end-of-file indicator is typed.



5.6 switch Multiple-Selection Statement (Cont.)

- ▶ Scanner method `hasNext` determine whether there is more data to input. This method returns the boolean value `true` if there is more data; otherwise, it returns `false`.
- ▶ As long as the end-of-file indicator has not been typed, method `hasNext` will return `true`.



5.6 switch Multiple-Selection Statement (Cont.)

- ▶ The **switch** statement consists of a block that contains a sequence of **case labels** and an optional **default case**.
- ▶ The program evaluates the **controlling expression** in the parentheses following keyword **switch**.
- ▶ The program compares the controlling expression's value (which must evaluate to an integral value of type **byte**, **char**, **short** or **int**) with each **case** label.
- ▶ If a match occurs, the program executes that **case**'s statements.
- ▶ The **break statement** causes program control to proceed with the first statement after the **switch**.



5.6 switch Multiple-Selection Statement (Cont.)

- ▶ **switch** does not provide a mechanism for testing ranges of values—every value must be listed in a separate **case** label.
- ▶ Note that each **case** can have multiple statements.
- ▶ **switch** differs from other control statements in that it does not require braces around multiple statements in a **case**.
- ▶ Without **break**, the statements for a matching case and subsequent cases execute until a **break** or the end of the **switch** is encountered. This is called “falling through.”
- ▶ If no match occurs between the controlling expression’s value and a **case** label, the **default** case executes.
- ▶ If no match occurs and there is no **default** case, program control simply continues with the first statement after the **switch**.



Software Engineering Observation 5.2

*Recall from Chapter 3 that methods declared with access modifier **private** can be called only by other methods of the class in which the **private** methods are declared. Such methods are commonly referred to as **utility methods** or **helper methods** because they're typically used to support the operation of the class's other methods.*



5.6 switch Multiple-Selection Statement (Cont.)

- ▶ Figure 5.11 shows the UML activity diagram for the general **switch** statement.
- ▶ Most **switch** statements use a **break** in each **case** to terminate the **switch** statement after processing the **case**.
- ▶ The **break** statement is not required for the **switch**'s last **case** (or the optional **default** case, when it appears last), because execution continues with the next statement after the **switch**.

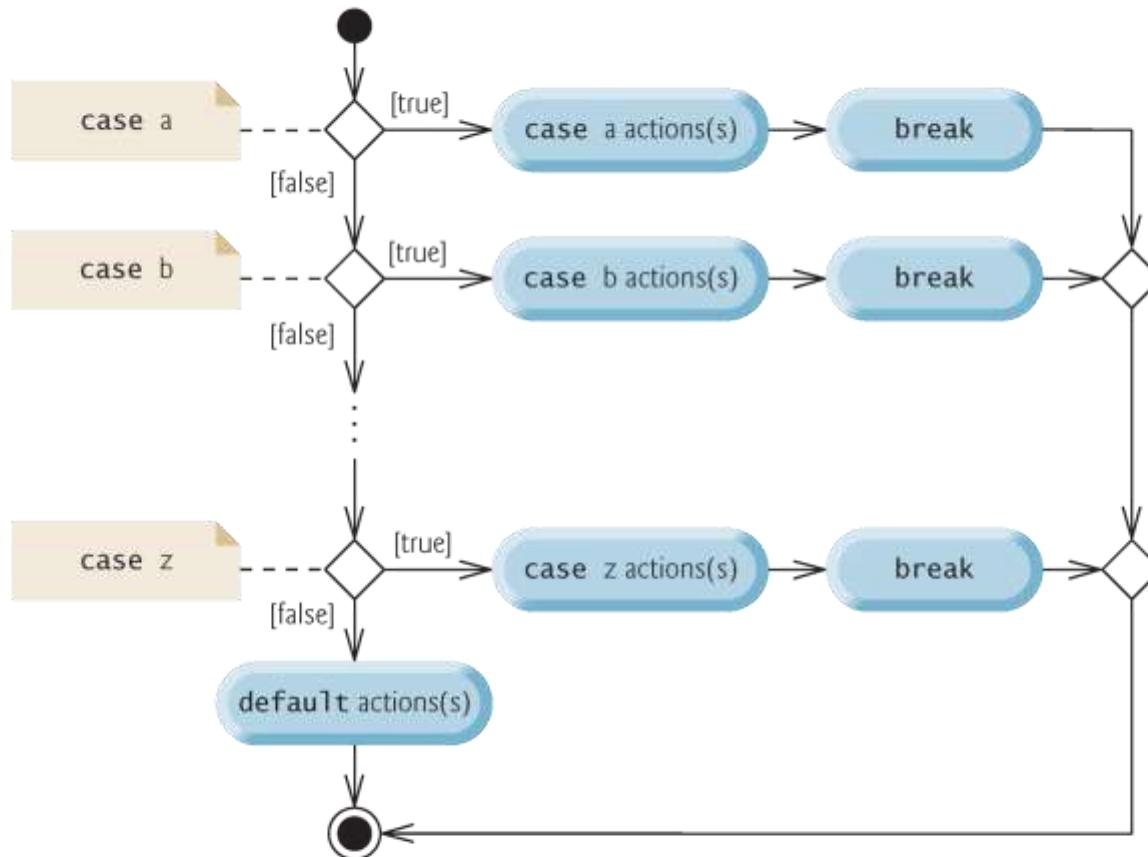


Fig. 5.11 | switch multiple-selection statement UML activity diagram with break statements.



Software Engineering Observation 5.3

Provide a default case in switch statements.

Including a default case focuses you on the need to process exceptional conditions.



Good Programming Practice 5.3

Although each case and the default case in a switch can occur in any order, place the default case last. When the default case is listed last, the break for that case is not required.



5.6 switch Multiple-Selection Statement (Cont.)

- ▶ When using the **switch** statement, remember that each **case** must contain a constant integral expression.
- ▶ An integer constant is simply an integer value.
- ▶ In addition, you can use **character constants**—specific characters in single quotes, such as 'A', '7' or '\$'—which represent the integer values of characters.
- ▶ The expression in each **case** can also be a **constant variable**—a variable that contains a value which does not change for the entire program. Such a variable is declared with keyword **final**.
- ▶ Java has a feature called enumerations. Enumeration constants can also be used in **case** labels.



5.6 switch Multiple-Selection Statement (Cont.)

- ▶ As of Java SE 7, you can use Strings in a switch statement's controlling expression and in case labels as in:

- ▶

```
switch( city )
{
    case "Maynard":
        zipCode = "01754";
        break;
    case "Marlborough":
        zipCode = "01752";
        break;
    case "Framingham":
        zipCode = "01701";
        break;
} // end switch
```



5.7 break and continue Statements

- ▶ The **break** statement, when executed in a **while**, **for**, **do...while** or **switch**, causes immediate exit from that statement.
- ▶ Execution continues with the first statement after the control statement.
- ▶ Common uses of the **break** statement are to escape early from a loop or to skip the remainder of a **switch**.



```
1 // Fig. 5.12: BreakTest.java
2 // break statement exiting a for statement.
3 public class BreakTest
4 {
5     public static void main( String[] args )
6     {
7         int count; // control variable also used after loop terminates
8
9         for ( count = 1; count <= 10; count++ ) // loop 10 times
10        {
11            if ( count == 5 ) // if count is 5,
12                break;           // terminate loop
13
14            System.out.printf( "%d ", count );
15        } // end for
16
17        System.out.printf( "\nBroke out of loop at count = %d\n", count );
18    } // end main
19 } // end class BreakTest
```

```
1 2 3 4
Broke out of loop at count = 5
```

Terminates the loop immediately and program control continues at line 17

Fig. 5.12 | break statement exiting a for statement.



5.7 break and continue Statements (Cont.)

- ▶ The **continue** statement, when executed in a **while**, **for** or **do...while**, skips the remaining statements in the loop body and proceeds with the next iteration of the loop.
- ▶ In **while** and **do...while** statements, the program evaluates the loop-continuation test immediately after the **continue** statement executes.
- ▶ In a **for** statement, the increment expression executes, then the program evaluates the loop-continuation test.



```
1 // Fig. 5.13: ContinueTest.java
2 // continue statement terminating an iteration of a for statement.
3 public class ContinueTest
4 {
5     public static void main( String[] args )
6     {
7         for ( int count = 1; count <= 10; count++ ) // loop 10 times
8         {
9             if ( count == 5 ) // if count is 5,
10                 continue; // skip remaining code in loop
11
12             System.out.printf( "%d ", count );
13         } // end for
14
15         System.out.println( "\nUsed continue to skip printing 5" );
16     } // end main
17 } // end class ContinueTest
```

```
1 2 3 4 6 7 8 9 10
Used continue to skip printing 5
```

Terminates current iteration of loop
and proceeds to increment

Fig. 5.13 | continue statement terminating an iteration of a for statement.



Software Engineering Observation 5.4

Some programmers feel that break and continue violate structured programming. Since the same effects are achievable with structured programming techniques, these programmers do not use break or continue.



Software Engineering Observation 5.5

There's a tension between achieving quality software engineering and achieving the best-performing software. Sometimes one of these goals is achieved at the expense of the other. For all but the most performance-intensive situations, apply the following rule of thumb: First, make your code simple and correct; then make it fast and small, but only if necessary.



5.8 Logical Operators

- ▶ Java's **logical operators** enable you to form more complex conditions by combining simple conditions.
- ▶ The logical operators are
 - `&&` (conditional AND)
 - `||` (conditional OR)
 - `&` (boolean logical AND)
 - `|` (boolean logical inclusive OR)
 - `^` (boolean logical exclusive OR)
 - `!` (logical NOT).
- ▶ [Note: The `&`, `|` and `^` operators are also bitwise operators when they are applied to integral operands.]



5.8 Logical Operators (Cont.)

- ▶ The **& (conditional AND)** operator ensures that two conditions are *both true* before choosing a certain path of execution.
- ▶ The table in Fig. 5.14 summarizes the **&&** operator. The table shows all four possible combinations of **false** and **true** values for *expression1* and *expression2*.
- ▶ Such tables are called **truth tables**. Java evaluates to **false** or **true** all expressions that include relational operators, equality operators or logical operators.

expression1	expression2	expression1 && expression2
false	false	false
false	true	false
true	false	false
true	true	true

Fig. 5.14 | && (conditional AND) operator truth table.



5.8 Logical Operators (Cont.)

- ▶ The **|| (conditional OR)** operator ensures that *either or both* of two conditions are true before choosing a certain path of execution.
- ▶ Figure 5.15 is a truth table for operator conditional OR (||).
- ▶ Operator **&&** has a higher precedence than operator **||**.
- ▶ Both operators associate from left to right.



expression1	expression2	expression1 expression2
false	false	false
false	true	true
true	false	true
true	true	true

Fig. 5.15 | || (conditional OR) operator truth table.



5.8 Logical Operators (Cont.)

- ▶ The parts of an expression containing `&&` or `||` operators are evaluated only until it's known whether the condition is true or false. T
- ▶ This feature of conditional AND and conditional OR expressions is called **short-circuit evaluation**.



Common Programming Error 5.7

In expressions using operator `&&`, a condition—we'll call this the dependent condition—may require another condition to be true for the evaluation of the dependent condition to be meaningful. In this case, the dependent condition should be placed after the other condition, or an error might occur. For example, in the expression `(i != 0) && (10 /i == 2)`, the second condition must appear after the first condition, or a divide-by-zero error might occur.



5.8 Logical Operators (Cont.)

- ▶ The **boolean logical AND (&)** and **boolean logical inclusive OR (|)** operators are identical to the `&&` and `||` operators, except that the `&` and `|` operators *always evaluate both of their operands* (i.e., they do not perform short-circuit evaluation).
- ▶ This is useful if the right operand of the boolean logical AND or boolean logical inclusive OR operator has a required **side effect**—a modification of a variable's value.



Error-Prevention Tip 5.6

For clarity, avoid expressions with side effects in conditions. The side effects may look clever, but they can make it harder to understand code and can lead to subtle logic errors.



5.8 Logical Operators (Cont.)

- ▶ A simple condition containing the **boolean logical exclusive OR** (\wedge) operator is **true if and only if** one of its operands is **true** and the other is **false**.
- ▶ If both are **true** or both are **false**, the entire condition is **false**.
- ▶ Figure 5.16 is a truth table for the boolean logical exclusive OR operator (\wedge).
- ▶ This operator is guaranteed to evaluate both of its operands.



expression1	expression2	expression1 ^ expression2
false	false	false
false	true	true
true	false	true
true	true	false

Fig. 5.16 | \wedge (boolean logical exclusive OR) operator truth table.



5.8 Logical Operators (Cont.)

- ▶ The **!** (**logical NOT**, also called **logical negation** or **logical complement**) operator “reverses” the meaning of a condition.
- ▶ The logical negation operator is a unary operator that has only a single condition as an operand.
- ▶ The logical negation operator is placed before a condition to choose a path of execution if the original condition (without the logical negation operator) is **false**.
- ▶ In most cases, you can avoid using logical negation by expressing the condition differently with an appropriate relational or equality operator.
- ▶ Figure 5.17 is a truth table for the logical negation operator.



expression	!expression
false	true
true	false

Fig. 5.17 | ! (logical negation, or logical NOT) operator truth table.



5.8 Logical Operators (Cont.)

- ▶ Figure 5.18 produces the truth tables discussed in this section.
- ▶ The **%b format specifier** displays the word “true” or the word “false” based on a **boolean** expression’s value.



```
1 // Fig. 5.18: LogicalOperators.java
2 // Logical operators.
3
4 public class LogicalOperators
{
5     public static void main( String[] args )
6     {
7         // create truth table for && (conditional AND) operator
8         System.out.printf( "%s\n%s: %b\n%s: %b\n%s: %b\n%s: %b\n\n",
9             "Conditional AND (&&)", "false && false", ( false && false ),
10            "false && true", ( false && true ),
11            "true && false", ( true && false ),
12            "true && true", ( true && true ) );
13
14
15         // create truth table for || (conditional OR) operator
16         System.out.printf( "%s\n%s: %b\n%s: %b\n%s: %b\n%s: %b\n\n",
17             "Conditional OR (||)", "false || false", ( false || false ),
18            "false || true", ( false || true ),
19            "true || false", ( true || false ),
20            "true || true", ( true || true ) );
21 }
```

Value of each condition like this is displayed using format specifier %b

Fig. 5.18 | Logical operators. (Part I of 4.)



```
22 // create truth table for & (boolean logical AND) operator
23 System.out.printf( "%s\n%s: %b\n%s: %b\n%s: %b\n%n",
24     "Boolean logical AND (&)", "false & false", ( false & false ),
25     "false & true", ( false & true ),
26     "true & false", ( true & false ),
27     "true & true", ( true & true ) );
28
29 // create truth table for | (boolean logical inclusive OR) operator
30 System.out.printf( "%s\n%s: %b\n%s: %b\n%s: %b\n%n",
31     "Boolean logical inclusive OR (|)", 
32     "false | false", ( false | false ),
33     "false | true", ( false | true ),
34     "true | false", ( true | false ),
35     "true | true", ( true | true ) );
36
37 // create truth table for ^ (boolean logical exclusive OR) operator
38 System.out.printf( "%s\n%s: %b\n%s: %b\n%s: %b\n%n",
39     "Boolean logical exclusive OR (^)",
40     "false ^ false", ( false ^ false ),
41     "false ^ true", ( false ^ true ),
42     "true ^ false", ( true ^ false ),
43     "true ^ true", ( true ^ true ) );
44
```

Fig. 5.18 | Logical operators. (Part 2 of 4.)



```
45     // create truth table for ! (logical negation) operator
46     System.out.printf( "%s\n%s: %b\n%s: %b\n", "Logical NOT (!)",
47             "false", ( !false ), "true", ( !true ) );
48 } // end main
49 } // end class LogicalOperators
```

```
Conditional AND (&&)
false && false: false
false && true: false
true && false: false
true && true: true
```

```
Conditional OR (||)
false || false: false
false || true: true
true || false: true
true || true: true
```

```
Boolean logical AND (&)
false & false: false
false & true: false
true & false: false
true & true: true
```

Fig. 5.18 | Logical operators. (Part 3 of 4.)



```
Boolean logical inclusive OR (|)  
false | false: false  
false | true: true  
true | false: true  
true | true: true
```

```
Boolean logical exclusive OR (^)  
false ^ false: false  
false ^ true: true  
true ^ false: true  
true ^ true: false
```

```
Logical NOT (!)  
!false: true  
!true: false
```

Fig. 5.18 | Logical operators. (Part 4 of 4.)



Operators	Associativity	Type
<code>++ --</code>	right to left	unary postfix
<code>++ -- + - ! (type)</code>	right to left	unary prefix
<code>* / %</code>	left to right	multiplicative
<code>+ -</code>	left to right	additive
<code>< <= > >=</code>	left to right	relational
<code>== !=</code>	left to right	equality
<code>&</code>	left to right	boolean logical AND
<code>^</code>	left to right	boolean logical exclusive OR
<code> </code>	left to right	boolean logical inclusive OR
<code>&&</code>	left to right	conditional AND
<code> </code>	left to right	conditional OR
<code>?:</code>	right to left	conditional
<code>= += -= *= /= %=</code>	right to left	assignment

Fig. 5.19 | Precedence/associativity of the operators discussed so far.



5.9 Structured Programming Summary

- ▶ Figure 5.20 uses UML activity diagrams to summarize Java's control statements.
- ▶ Java includes only single-entry/single-exit control statements—there is only one way to enter and only one way to exit each control statement.
- ▶ Connecting control statements in sequence to form structured programs is simple. The final state of one control statement is connected to the initial state of the next—that is, the control statements are placed one after another in a program in sequence. We call this control-statement stacking.
- ▶ The rules for forming structured programs also allow for control statements to be nested.

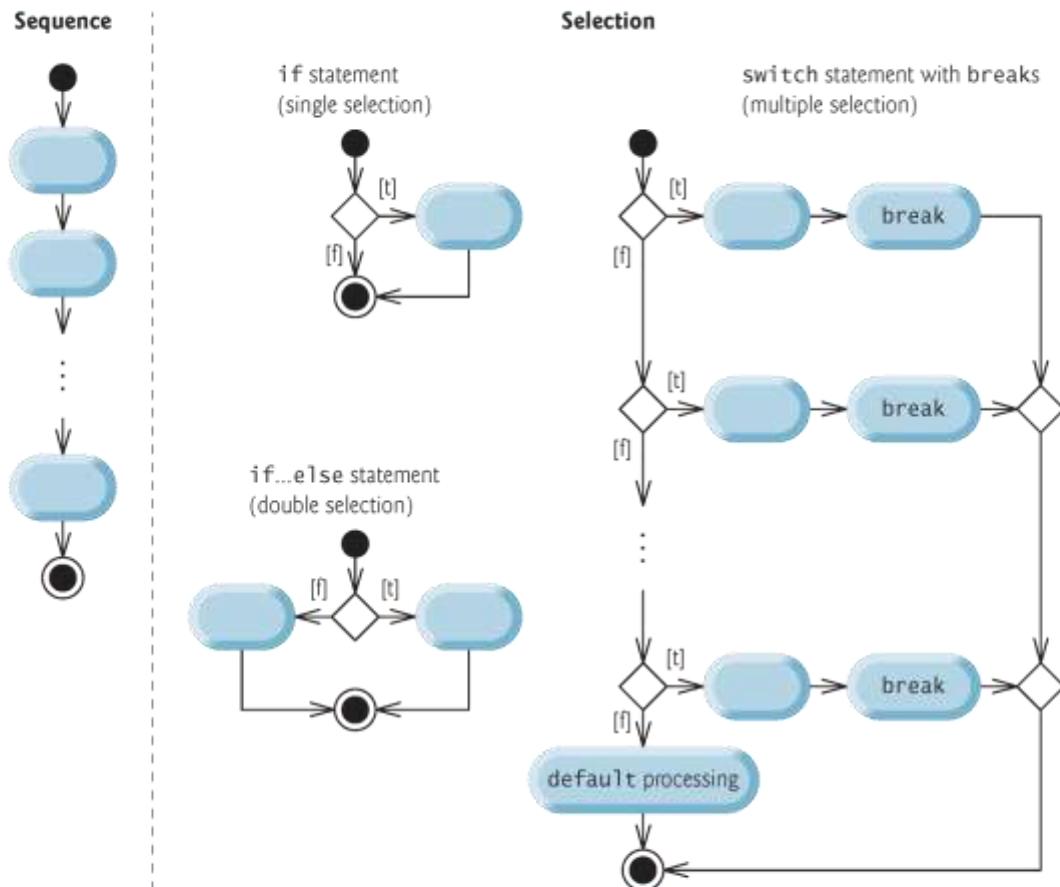
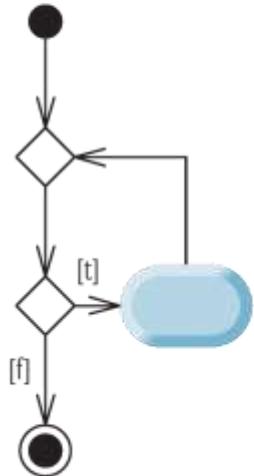


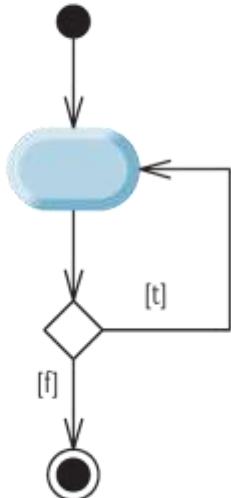
Fig. 5.20 | Java's single-entry/single-exit sequence, selection and repetition statements. (Part I of 2.)

Repetition

while statement



do...while statement



for statement

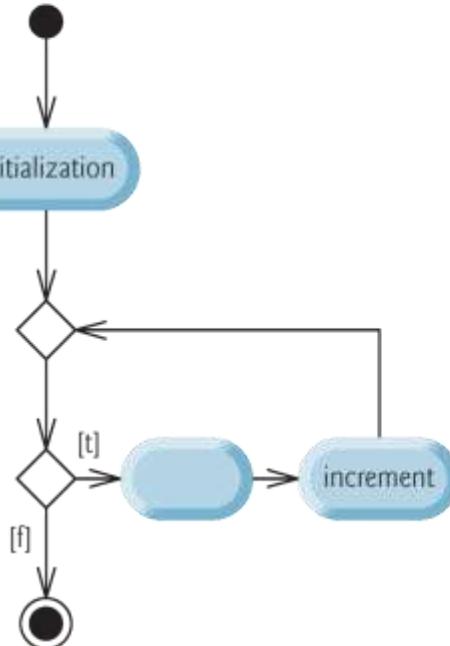


Fig. 5.20 | Java's single-entry/single-exit sequence, selection and repetition statements. (Part 2 of 2.)



5.9 Structured Programming Summary (Cont.)

- ▶ Structured programming promotes simplicity.
- ▶ Bohm and Jacopini: Only three forms of control are needed to implement an algorithm:
 - Sequence
 - Selection
 - Repetition
- ▶ The sequence structure is trivial. Simply list the statements to execute in the order in which they should execute.



5.9 Structured Programming Summary (Cont.)

- ▶ Selection is implemented in one of three ways:
 - `if` statement (single selection)
 - `if...else` statement (double selection)
 - `switch` statement (multiple selection)
- ▶ The simple `if` statement is sufficient to provide any form of selection—everything that can be done with the `if...else` statement and the `switch` statement can be implemented by combining `if` statements.



5.9 Structured Programming Summary (Cont.)

- ▶ Repetition is implemented in one of three ways:
 - **while** statement
 - **do...while** statement
 - **for** statement
- ▶ The **while** statement is sufficient to provide any form of repetition. Everything that can be done with **do...while** and **for** can be done with the **while** statement.



5.9 Structured Programming Summary (Cont.)

- ▶ Combining these results illustrates that any form of control ever needed in a Java program can be expressed in terms of
 - sequence
 - `if` statement (selection)
 - `while` statement (repetition)and that these can be combined in only two ways—stacking and nesting.



Introduction to Classes, Objects Methods and Strings



OBJECTIVES

In this chapter you'll learn:

- How to declare a class and use it to create an object.
- How to implement a class's behaviors as methods.
- How to implement a class's attributes as instance variables and properties.
- How to call an object's methods to make them perform their tasks.
- What instance variables of a class and local variables of a method are.
- How to use a constructor to initialize an object's data.
- The differences between primitive and reference types.



3.1 Introduction

3.2 Declaring a Class with a Method and Instantiating an Object of a Class

3.3 Declaring a Method with a Parameter

3.4 Instance Variables, *set* Methods and *get* Methods

3.5 Primitive Types vs. Reference Types

3.6 Initializing Objects with Constructors

3.7 Floating-Point Numbers and Type `double`

3.8 (Optional) GUI and Graphics Case Study: Using Dialog Boxes

3.9 Wrap-Up



3.1 Introduction

- ▶ Covered in this chapter
 - Classes
 - Objects
 - Methods
 - Parameters
 - `double` primitive type



Quick Discussion – Arrays

- ▶ Array declaration

```
int arr []; or int[] arr;
```

This just declares the variable

- ▶ To initialize and allocate memory

```
int[] arr = new int[100];
```

- ▶ Shorthand initialization

```
int[] arr = {2,7,9};
```



Quick Discussion – Arrays

- ▶ To get length of array, use `array.length`

```
for (int i=0; i<arr.length. i++)  
    System.out.println(arr[i]);
```



3.2 Declaring a Class with a Method and Instantiating an Object of a Class

- ▶ Create a new class (**GradeBook**)
- ▶ Use it to create an object.
- ▶ Each class declaration that begins with keyword **public** must be stored in a file that has the same name as the class and ends with the **.java** file-name extension.
- ▶ Keyword **public** is an **access modifier**.
 - Indicates that the class is “available to the public”



3.2 Declaring a Class with a Method and Instantiating an Object of a Class (Cont.)

- ▶ The `main` method is called automatically by the Java Virtual Machine (JVM) when you execute an application.
- ▶ Normally, you must call methods explicitly to tell them to perform their tasks.
- ▶ A `public` is “available to the public”
 - It can be called from methods of other classes.
- ▶ The `return type` specifies the type of data the method returns after performing its task.
- ▶ Return type `void` indicates that a method will perform a task but will *not* return (i.e., give back) any information to its `calling method` when it completes its task.



3.2 Declaring a Class with a Method and Instantiating an Object of a Class (Cont.)

- ▶ Method name follows the return type.
- ▶ By convention, method names begin with a lowercase first letter and subsequent words in the name begin with a capital letter.
- ▶ Empty parentheses after the method name indicate that the method does not require additional information to perform its task.
- ▶ Together, everything in the first line of the method is typically called the **Method header**
- ▶ Every method's body is delimited by left and right braces.
- ▶ The method body contains one or more statements that perform the method's task.



```
1 // Fig. 3.1: GradeBook.java
2 // Class declaration with one method.
3
4 public class GradeBook
{
    // display a welcome message to the GradeBook user
7    public void displayMessage()
8    {
9        System.out.println( "Welcome to the Grade Book!" );
10   } // end method displayMessage
11 } // end class GradeBook
```

Performs the task of displaying a message on the screen; method `displayMessage` must be called to perform this task

Fig. 3.1 | Class declaration with one method.



Eclipse Installation

<https://www.eclipse.org/downloads/packages/>



3.2 Declaring a Class with a Method and Instantiating an Object of a Class (Cont.)

- ▶ Use class **GradeBook** in an application.
- ▶ Class **GradeBook** is not an application because it does not contain **main**.
- ▶ Can't execute **GradeBook**; will receive an error message like:
 - Exception in thread "main"
java.lang.NoSuchMethodError: main
- ▶ Must either declare a separate class that contains a **main** method or place a **main** method in class **GradeBook**.
- ▶ To help you prepare for the larger programs, use a separate class containing method **main** to test each new class.
- ▶ Some programmers refer to such a class as a driver class.



```
1 // Fig. 3.2: GradeBookTest.java
2 // Creating a GradeBook object and calling its displayMessage method.
3
4 public class GradeBookTest
{
5
6     // main method begins program execution
7     public static void main( String[] args )
8     {
9         // create a GradeBook object and assign it to myGradeBook
10        GradeBook myGradeBook = new GradeBook(); ← Creates a GradeBook object and
11
12        // call myGradeBook's displayMessage method
13        myGradeBook.displayMessage(); ← assigns it to variable myGradeBook
14    } // end main
15 } // end class GradeBookTest
```

Welcome to the Grade Book!

Fig. 3.2 | Creating a GradeBook object and calling its displayMessage method.



3.2 Declaring a Class with a Method and Instantiating an Object of a Class (Cont.)

- ▶ A `static` method (such as `main`) is special
 - It can be called without first creating an object of the class in which the method is declared.
- ▶ Typically, you cannot call a method that belongs to another class until you create an object of that class.
- ▶ Declare a variable of the class type.
 - Each new class you create becomes a new type that can be used to declare variables and create objects.
 - You can declare new class types as needed; this is one reason why Java is known as an **extensible language**.



3.2 Declaring a Class with a Method and Instantiating an Object of a Class (Cont.)

▶ Class instance creation expression

- Keyword **new** creates a new object of the class specified to the right of the keyword.
- Used to initialize a variable of a class type.
- The parentheses to the right of the class name are required.
- Parentheses in combination with a class name represent a call to a **constructor**, which is similar to a method but is used only at the time an object is created to initialize the object's data.



3.2 Declaring a Class with a Method and Instantiating an Object of a Class (Cont.)

- ▶ Call a method via the class-type variable
 - Variable name followed by a **dot separator** (.), the method name and parentheses.
 - Call causes the method to perform its task.
- ▶ Any class can contain a **main** method.
 - The JVM invokes the **main** method only in the class used to execute the application.
 - If multiple classes that contain **main**, then one that is invoked is the one in the class named in the **java** command.



3.2 Declaring a Class with a Method and Instantiating an Object of a Class (Cont.)

- ▶ Compiling an Application with Multiple Classes
 - Compile the classes in Fig. 3.1 and Fig. 3.2 before executing.
 - Type the command

```
javac GradeBook.java GradeBookTest.java
```
 - If the directory containing the application includes only this application's files, you can compile all the classes in the directory with the command

```
javac *.java
```



Multiple classes

- ▶ When two classes are present in a single source file. Eg. Employee class into a file Employee.java and the EmployeeTest class into EmployeeTest.java.
- ▶ Invoke the Java compiler with a wildcard:
 - javac Employee*.java
- ▶ Or, you can simply type
- ▶ javac EmployeeTest.java

If the Java compiler sees the Employee class being used inside EmployeeTest.java, it will look for a file named Employee.class. If it does not find that file, it automatically searches for Employee.java and compiles it.



3.3 Declaring a Method with a Parameter

- ▶ Car analogy
 - Pressing a car's gas pedal sends a message to the car to perform a task—make the car go faster.
 - The farther down you press the pedal, the faster the car accelerates.
 - Message to the car includes the task to perform and additional information that helps the car perform the task.
- ▶ **Parameter:** Additional information a method needs to perform its task.
 - **Implicit Parameter** (later)
 - **Explicit Parameter**



3.3 Declaring a Method with a Parameter (Cont.)

- ▶ A method can require one or more parameters that represent additional information it needs to perform its task.
 - Defined in a comma-separated parameter list
 - Located in the parentheses that follow the method name
 - Each parameter must specify a type and an identifier.
- ▶ A method call supplies values—called arguments—for each of the method's parameters.



```
1 // Fig. 3.4: GradeBook.java
2 // Class declaration with a method that has a parameter.
3
4 public class GradeBook
{
    // display a welcome message to the GradeBook user
7    public void displayMessage( String courseName ) ←
8    {
9        System.out.printf( "Welcome to the grade book for\n%s!\n",
10                      courseName ); ←
11
12 } // end class GradeBook
```

Parameter `courseName` provides the additional information that the method requires to perform its task

Parameter `courseName`'s value is displayed as part of the output

Fig. 3.4 | Class declaration with one method that has a parameter.



```
1 // Fig. 3.5: GradeBookTest.java
2 // Create GradeBook object and pass a String to
3 // its displayMessage method.
4 import java.util.Scanner; // program uses Scanner
5
6 public class GradeBookTest
7 {
8     // main method begins program execution
9     public static void main( String[] args )
10    {
11        // create Scanner to obtain input from command window
12        Scanner input = new Scanner( System.in );
13
14        // create a GradeBook object and assign it to myGradeBook
15        GradeBook myGradeBook = new GradeBook();
16
17        // prompt for and input course name
18        System.out.println( "Please enter the course name:" );
19        String nameOfCourse = input.nextLine(); // read a line of text
20        System.out.println(); // outputs a blank line
21
```

Reads a String from
the user

Fig. 3.5 | Creating a GradeBook object and passing a String to its displayMessage method. (Part I of 2.)



```
22     // call myGradeBook's displayMessage method  
23     // and pass nameOfCourse as an argument  
24     myGradeBook.displayMessage( nameOfCourse ); ←  
25 } // end main  
26 } // end class GradeBookTest
```

Passes the value of `nameOfCourse` as
the argument to method
`displayMessage`

```
Please enter the course name:  
CS101 Introduction to Java Programming
```

```
Welcome to the grade book for  
CS101 Introduction to Java Programming!
```

Fig. 3.5 | Creating a `GradeBook` object and passing a `String` to its `displayMessage` method. (Part 2 of 2.)



3.3 Declaring a Method with a Parameter (Cont.)

▶ Scanner method `nextLine`

- Reads characters typed by the user until the newline character is encountered
- Returns a `String` containing the characters up to, but not including, the newline
- Press *Enter* to submit the string to the program.
- Pressing *Enter* inserts a newline character at the end of the characters the user typed.
- The newline character is discarded by `nextLine`.

▶ Scanner method `next`

- Reads individual words
- Reads characters until a white-space character is encountered, then returns a `String` (the white-space character is discarded).
- Information after the first white-space character can be read by other statements that call the `Scanner`'s methods later in the program-.



3.3 Declaring a Method with a Parameter (Cont.)

- ▶ More on Arguments and Parameters
 - The number of arguments in a method call must match the number of parameters in the parameter list of the method's declaration.
 - The argument types in the method call must be “consistent with” the types of the corresponding parameters in the method's declaration.



3.3 Declaring a Method with a Parameter (Cont.)

► Notes on **import** Declarations

- Classes **System** and **String** are in package **java.lang**
 - Implicitly imported into every Java program
 - Can use the **java.lang** classes without explicitly importing them
 - Most classes you'll use in Java programs must be imported explicitly.
- Classes that are compiled in the same directory on disk are in the same package—known as the **default package**.
- Classes in the same package are implicitly imported into the source-code files of other classes in the same package.
- An **import** declaration is not required if you always refer to a class via its **fully qualified class name**
 - Package name followed by a dot (.) and the class name.



Software Engineering Observation 3.1

The Java compiler does not require `import` declarations in a Java source-code file if the fully qualified class name is specified every time a class name is used in the source code. Most Java programmers prefer to use `import` declarations.



3.4 Instance Variables, Accessor (get) and Mutators (set)

▶ Local variables

- Variables declared in the body of a particular method.
- When a method terminates, the values of its local variables are lost.
- Recall from Section 3.2 that an object has attributes that are carried with the object as it's used in a program. Such attributes exist before a method is called on an object and after the method completes execution.



3.4 Instance Variables, *set Methods* and *get Methods* (Cont.)

- ▶ A class normally consists of one or more methods that manipulate the attributes that belong to a particular object of the class.
 - Attributes are represented as variables in a class declaration.
 - Called **fields**.
 - Declared inside a class declaration but outside the bodies of the class's method declarations.
- ▶ **Instance variable**
 - When each object of a class maintains its own copy of an attribute, the field is an instance variable
 - Each object (instance) of the class has a separate instance of the variable in memory.



```
1 // Fig. 3.7: GradeBook.java
2 // GradeBook class that contains a courseName instance variable
3 // and methods to set and get its value.
4
5 public class GradeBook
6 {
7     private String courseName; // course name for this GradeBook
8
9     // method to set the course name
10    public void setCourseName( String name )
11    {
12        courseName = name; // store the course name
13    } // end method setCourseName
14
15    // method to retrieve the course name
16    public String getCourseName()
17    {
18        return courseName;
19    } // end method getCourseName
20}
```

Each GradeBook object maintains its own copy of instance variable courseName

Method allows client code to change the courseName

Method allows client code to obtain the courseName

Fig. 3.7 | GradeBook class that contains a courseName instance variable and methods to set and get its value. (Part 1 of 2.)



```
23
24     // calls getCourseName to get the name of
25     // the course this GradeBook represents
26     System.out.printf( "Welcome to the grade book for\n%s!\n",
27         getCourseName() );
28 } // end method displayMessage
29 } // end class GradeBook
```

variable `courseName` and the class's other methods

Good practice to access your instance variables via set or get methods

Fig. 3.7 | GradeBook class that contains a `courseName` instance variable and methods to set and get its value. (Part 2 of 2.)

Be careful not to write accessor methods that return references to mutable objects.

```
class Employee
{
    private Date hireDay;
    . . .
    public Date getHireDay()
    {
        return hireDay; // BAD
    }
    . . .
}
```

This is allowed syntactically, but now the calling method can change the `Date` value.



3.4 Instance Variables, *set Methods* and *get Methods* (Cont.)

- ▶ Every instance (i.e., object) of a class contains one copy of each instance variable.
- ▶ Instance variables typically declared **private**.
 - **private** is an access modifier.
 - **private** variables and methods are accessible only to methods of the class in which they are declared.
- ▶ Declaring instance **private** is known as **data hiding** or information hiding.
- ▶ **private** variables are encapsulated (hidden) in the object and can be accessed only by methods of the object's class.
 - Prevents instance variables from being modified accidentally by a class in another part of the program.
 - *Set* and *get* methods used to access instance variables.



Software Engineering Observation 3.2

*Precede each field and method declaration with an access modifier. Generally, instance variables should be declared **private** and methods **public**. (It's appropriate to declare certain methods **private**, if they'll be accessed only by other methods of the class.)*



Good Programming Practice 3.1

We prefer to list a class's fields first, so that, as you read the code, you see the names and types of the variables before they're used in the class's methods. You can list the class's fields anywhere in the class outside its method declarations, but scattering them can lead to hard-to-read code.



3.4 Instance Variables, *set Methods* and *get Methods* (Cont.)

- ▶ When a method that specifies a return type other than `void` completes its task, the method returns a result to its calling method.
- ▶ Method `setCourseName` and `getCourseName` each use variable `courseName` even though it was not declared in any of the methods.
 - Can use an instance variable of the class in each of the classes methods.
 - Exception to this is `static` methods (Chapter 8)
- ▶ The order in which methods are declared in a class does not determine when they are called at execution time.
- ▶ One method of a class can call another method of the same class by using just the method name.



3.4 Instance Variables, *set Methods* and *get Methods* (Cont.)

- ▶ Unlike local variables, which are not automatically initialized, every field has a **default initial value**—a value provided by Java when you do not specify the field's initial value.
- ▶ Fields are not required to be explicitly initialized before they are used in a program—unless they must be initialized to values other than their default values.
- ▶ The default value for a field of type **String** is **null**



```
1 // Fig. 3.8: GradeBookTest.java
2 // Creating and manipulating a GradeBook object.
3 import java.util.Scanner; // program uses Scanner
4
5 public class GradeBookTest
6 {
7     // main method begins program execution
8     public static void main( String[] args )
9     {
10         // create Scanner to obtain input from command window
11         Scanner input = new Scanner( System.in );
12
13         // create a GradeBook object and assign it to myGradeBook
14         GradeBook myGradeBook = new GradeBook();
15
16         // display initial value of courseName
17         System.out.printf( "Initial course name is: %s\n\n",
18             myGradeBook.getCourseName() ); ← Gets the value of the myGradeBook
19                                     object's courseName instance variable
20
21         // prompt for and read course name
22         System.out.println( "Please enter the course name:" );
23         String theName = input.nextLine(); // read a line of text
24         myGradeBook.setCourseName( theName ); // set the course name ← Sets the value of the
25                                         courseName instance variable
```

Fig. 3.8 | Creating and manipulating a GradeBook object. (Part I of 2.)



```
24     System.out.println(); // outputs a blank line
25
26     // display welcome message after specifying course name
27     myGradeBook.displayMessage(); ←
28 } // end main
29 } // end class GradeBookTest
```

Displays the GradeBook's message,
including the value of the `courseName`
instance variable

Initial course name is: null

Please enter the course name:

CS101 Introduction to Java Programming

Welcome to the grade book for

CS101 Introduction to Java Programming!

Fig. 3.8 | Creating and manipulating a `GradeBook` object. (Part 2 of 2.)



3.4 Instance Variables, *set Methods* and *get Methods* (Cont.)

- ▶ *set* and *get* methods
 - A class's **private** fields can be manipulated only by the class's methods.
 - A **client of an object** calls the class's **public** methods to manipulate the **private** fields of an object of the class.
 - Classes often provide **public** methods to allow clients to *set* (i.e., assign values to) or *get* (i.e., obtain the values of) **private** instance variables.
 - The names of these methods need not begin with *set* or *get*, but this naming convention is recommended.



3.4 Instance Variables, *set Methods* and *get Methods* (Cont.)

- ▶ Figure 3.9 contains an updated UML class diagram for the version of class **GradeBook** in Fig. 3.7.
 - Models instance variable **courseName** as an attribute in the middle compartment of the class.
 - The UML represents instance variables as attributes by listing the attribute name, followed by a colon and the attribute type.
 - A minus sign (–) access modifier corresponds to access modifier **private**.



3.5 Primitive Types vs. Reference Types

- ▶ Types are divided into primitive types and **reference types**.
- ▶ The primitive types are **boolean**, **byte**, **char**, **short**, **int**, **long**, **float** and **double**.
- ▶ All nonprimitive types are reference types.
- ▶ A primitive-type variable can store exactly one value of its declared type at a time.
- ▶ Primitive-type instance variables are initialized by default—variables of types **byte**, **char**, **short**, **int**, **long**, **float** and **double** are initialized to 0, and variables of type **boolean** are initialized to **false**.
- ▶ You can specify your own initial value for a primitive-type variable by assigning the variable a value in its declaration.



3.5 Primitive Types vs. Reference Types

- ▶ Programs use variables of reference types (normally called **references**) to store the locations of objects in the computer's memory.
 - Such a variable is said to **refer to an object** in the program.
- ▶ Objects that are referenced may each contain many instance variables and methods.
- ▶ Reference-type instance variables are initialized by default to the value **null**
 - A reserved word that represents a “reference to nothing.”
- ▶ When using an object of another class, a reference to the object is required to **invoke** (i.e., call) its methods.



3.6 Initializing Objects with Constructors

- ▶ When an object of a class is created, its instance variables are initialized by default.
- ▶ Each class can provide a constructor that initializes an object of a class when the object is created.
- ▶ Java requires a constructor call for *every* object that is created.
- ▶ Keyword **new** requests memory from the system to store an object, then calls the corresponding class's constructor to initialize the object.
- ▶ A constructor *must* have the same name as the class.



3.6 Initializing Objects with Constructors (Cont.)

- ▶ By default, the compiler provides a **default constructor** with no parameters in any class that does not explicitly include a constructor.
 - Instance variables are initialized to their default values.
- ▶ Can provide your own constructor to specify custom initialization for objects of your class.
- ▶ A constructor's parameter list specifies the data it requires to perform its task.
- ▶ Constructors cannot return values, so they cannot specify a return type.
- ▶ Normally, constructors are declared **public**.
- ▶ *If you declare any constructors for a class, the Java compiler will not create a default constructor for that class.*



```
1 // Fig. 3.10: GradeBook.java
2 // GradeBook class with a constructor to initialize the course name.
3
4 public class GradeBook
{
5     private String courseName; // course name for this GradeBook
6
7     // constructor initializes courseName with String argument
8     public GradeBook( String name )
9     {
10         courseName = name; // initializes courseName
11     } // end constructor
12
13
14     // method to set the course name
15     public void setCourseName( String name )
16     {
17         courseName = name; // store the course name
18     } // end method setCourseName
19 }
```

Constructor that initializes
courseName to the specified value

Fig. 3.10 | GradeBook class with a constructor to initialize the course name. (Part I
of 2.)



```
20 // method to retrieve the course name
21 public String getCourseName()
22 {
23     return courseName;
24 } // end method getCourseName
25
26 // display a welcome message to the GradeBook user
27 public void displayMessage()
28 {
29     // this statement calls getCourseName to get the
30     // name of the course this GradeBook represents
31     System.out.printf( "Welcome to the grade book for\n%s!\n",
32                       getCourseName() );
33 } // end method displayMessage
34 } // end class GradeBook
```

Fig. 3.10 | GradeBook class with a constructor to initialize the course name. (Part 2 of 2.)



```
1 // Fig. 3.11: GradeBookTest.java
2 // GradeBook constructor used to specify the course name at the
3 // time each GradeBook object is created.
4
5 public class GradeBookTest
6 {
7     // main method begins program execution
8     public static void main( String[] args )
9     {
10         // create GradeBook object
11         GradeBook gradeBook1 = new GradeBook(
12             "CS101 Introduction to Java Programming" );
13         GradeBook gradeBook2 = new GradeBook(
14             "CS102 Data Structures in Java" );
15
16         // display initial value of courseName for each GradeBook
17         System.out.printf( "gradeBook1 course name is: %s\n",
18             gradeBook1.getCourseName() );
19         System.out.printf( "gradeBook2 course name is: %s\n",
20             gradeBook2.getCourseName() );
21     } // end main
22 } // end class GradeBookTest
```

Class instance creation expression initializes the GradeBook and returns a reference that is assigned to variable gradeBook1

Class instance creation expression initializes the GradeBook and returns a reference that is assigned to variable gradeBook1

Fig. 3.11 | GradeBook constructor used to specify the course name at the time each GradeBook object is created. (Part I of 2.)



```
gradeBook1 course name is: CS101 Introduction to Java Programming  
gradeBook2 course name is: CS102 Data Structures in Java
```

Fig. 3.11 | GradeBook constructor used to specify the course name at the time each GradeBook object is created. (Part 2 of 2.)



Methods: A Deeper Look



OBJECTIVES

In this chapter you'll learn:

- How `static` methods and fields are associated with classes rather than objects.
- How the method call/return mechanism is supported by the method-call stack.
- How packages group related classes.
- How to use random-number generation to implement game-playing applications.
- How the visibility of declarations is limited to specific regions of programs.
- What method overloading is and how to create overloaded methods.



- 6.1** Introduction
- 6.2** Program Modules in Java
- 6.3** `static` Methods, `static` Fields and Class Math
- 6.4** Declaring Methods with Multiple Parameters
- 6.5** Notes on Declaring and Using Methods
- 6.6** Method-Call Stack and Activation Records
- 6.7** Argument Promotion and Casting
- 6.8** Java API Packages
- 6.9** Case Study: Random-Number Generation
 - 6.9.1 Generalized Scaling and Shifting of Random Numbers
 - 6.9.2 Random-Number Repeatability for Testing and Debugging
- 6.10** Case Study: A Game of Chance; Introducing Enumerations
- 6.11** Scope of Declarations
- 6.12** Method Overloading
- 6.13** (Optional) GUI and Graphics Case Study: Colors and Filled Shapes
- 6.14** Wrap-Up



6.1 Introduction

- ▶ Best way to develop and maintain a large program is to construct it from small, simple pieces, or **modules**.
 - **divide and conquer.**
- ▶ Topics in this chapter
 - **static** methods
 - Declare a method with more than one parameter
 - Method-call stack
 - Simulation techniques with random-number generation.
 - How to declare values that cannot change (i.e., constants) in your programs.
 - Method overloading.



6.2 Program Modules in Java

- ▶ Java programs combine new methods and classes that you write with predefined methods and classes available in the [Java Application Programming Interface](#) and in other class libraries.
- ▶ Related classes are typically grouped into packages so that they can be imported into programs and reused.
 - You'll learn how to group your own classes into packages in Chapter 8.



6.2 Program Modules in Java (Cont.)

- ▶ Methods help you modularize a program by separating its tasks into self-contained units.
- ▶ Statements in method bodies
 - Written only once
 - Hidden from other methods
 - Can be reused from several locations in a program
- ▶ Divide-and-conquer approach
 - Constructing programs from small, simple pieces
- ▶ Software reusability
 - Use existing methods as building blocks to create new programs.
- ▶ Dividing a program into meaningful methods makes the program easier to debug and maintain.



Software Engineering Observation 6.2

To promote software reusability, every method should be limited to performing a single, well-defined task, and the name of the method should express that task effectively.



Error-Prevention Tip 6.1

A method that performs one task is easier to test and debug than one that performs many tasks.



6.2 Program Modules in Java (Cont.)

- ▶ Hierarchical form of management (Fig. 6.1).
 - A boss (the caller) asks a worker (the called method) to perform a task and report back (return) the results after completing the task.
 - The boss method does not know how the worker method performs its designated tasks.
 - The worker may also call other worker methods, unbeknown to the boss.
- ▶ “Hiding” of implementation details promotes good software engineering.

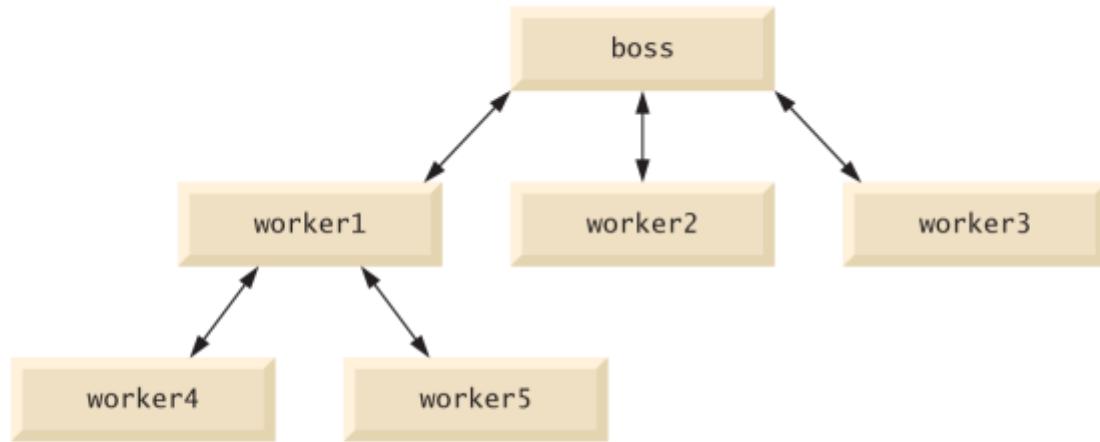


Fig. 6.1 | Hierarchical boss-method/worker-method relationship.



6.3 static Methods, static Fields and Class Math

- ▶ Sometimes a method performs a task that does not depend on the contents of any object.
 - Applies to the class in which it's declared as a whole
 - Known as a **static** method or a **class method**
- ▶ It's common for classes to contain convenient **static** methods to perform common tasks.
- ▶ To declare a method as **static**, place the keyword **static** before the return type in the method's declaration.
- ▶ Calling a **static** method
 - *ClassName . methodName(arguments)*
- ▶ **Class Math** provides a collection of **static** methods that enable you to perform common mathematical calculations.
- ▶ Method arguments may be constants, variables or expressions.



Software Engineering Observation 6.4

Class Math is part of the java.lang package, which is implicitly imported by the compiler, so it's not necessary to import class Math to use its methods.



Method	Description	Example
<code>abs(x)</code>	absolute value of x	<code>abs(23.7)</code> is 23.7 <code>abs(0.0)</code> is 0.0 <code>abs(-23.7)</code> is 23.7
<code>ceil(x)</code>	rounds x to the smallest integer not less than x	<code>ceil(9.2)</code> is 10.0 <code>ceil(-9.8)</code> is -9.0
<code>cos(x)</code>	trigonometric cosine of x (x in radians)	<code>cos(0.0)</code> is 1.0
<code>exp(x)</code>	exponential method e^x	<code>exp(1.0)</code> is 2.71828 <code>exp(2.0)</code> is 7.38906
<code>floor(x)</code>	rounds x to the largest integer not greater than x	<code>floor(9.2)</code> is 9.0 <code>floor(-9.8)</code> is -10.0
<code>log(x)</code>	natural logarithm of x (base e)	<code>log(Math.E)</code> is 1.0 <code>log(Math.E * Math.E)</code> is 2.0
<code>max(x, y)</code>	larger value of x and y	<code>max(2.3, 12.7)</code> is 12.7 <code>max(-2.3, -12.7)</code> is -2.3
<code>min(x, y)</code>	smaller value of x and y	<code>min(2.3, 12.7)</code> is 2.3 <code>min(-2.3, -12.7)</code> is -12.7

Fig. 6.2 | Math class methods. (Part I of 2.)



Method	Description	Example
<code>pow(x, y)</code>	x raised to the power y (i.e., x^y)	<code>pow(2.0, 7.0)</code> is 128.0 <code>pow(9.0, 0.5)</code> is 3.0
<code>sin(x)</code>	trigonometric sine of x (x in radians)	<code>sin(0.0)</code> is 0.0
<code>sqrt(x)</code>	square root of x	<code>sqrt(900.0)</code> is 30.0
<code>tan(x)</code>	trigonometric tangent of x (x in radians)	<code>tan(0.0)</code> is 0.0

Fig. 6.2 | Math class methods. (Part 2 of 2.)



6.3 static Methods, static Fields and Class Math (Cont.)

- ▶ Math fields for common mathematical constants
 - `Math.PI` (3.141592653589793)
 - `Math.E` (2.718281828459045)
- ▶ Declared in class `Math` with the modifiers `public`, `final` and `static`
 - `public` allows you to use these fields in your own classes.
 - A field declared with keyword `final` is constant—its value cannot change after the field is initialized.
 - `PI` and `E` are declared `final` because their values never change.



6.3 static Methods, static Fields and Class Math (Cont.)

- ▶ A field that represents an attribute is also known as an instance variable—each object (instance) of the class has a separate instance of the variable in memory.
- ▶ Fields for which each object of a class does not have a separate instance of the field are declared **static** and are also known as **class variables**.
- ▶ All objects of a class containing **static** fields share one copy of those fields.
- ▶ Together the class variables (i.e., **static** variables) and instance variables represent the fields of a class.



6.5 Notes on Declaring and Using Methods

- ▶ Three ways to call a method:
 - Using a method name by itself to call another method of the same class
 - Using a variable that contains a reference to an object, followed by a dot (.) and the method name to call a method of the referenced object
 - Using the class name and a dot (.) to call a **static** method of a class



6.3 static Methods, static Fields and Class Math (Cont.)

- ▶ Why is method `main` declared `static`?
 - The JVM attempts to invoke the `main` method of the class you specify—when no objects of the class have been created.
 - Declaring `main` as `static` allows the JVM to invoke `main` without creating an instance of the class.



Why static methods cannot call non-static methods?

Can we allow non-static methods to call static methods?



6.5 Notes on Declaring and Using Methods (Cont.)

- ▶ A non-**static** method can call any method of the same class directly and can manipulate any of the class's fields directly.
- ▶ A **static** method can call *only other static methods* of the same class directly and can manipulate *only static fields* in the same class directly.
 - To access the class's non-**static** members, a **static** method must use a reference to an object of the class.



6.5 Notes on Declaring and Using Methods (Cont.)

- ▶ Three ways to return control to the statement that calls a method:
 - When the program flow reaches the method-ending right brace
 - When the following statement executes
`return;`
 - When the method returns a result with a statement like
`return expression;`



6.6 Method-Call Stack and Activation Records

- ▶ **Stack** data structure
 - Analogous to a pile of dishes
 - A dish is placed on the pile at the top (referred to as **pushing** the dish onto the stack).
 - A dish is removed from the pile from the top (referred to as **popping** the dish off the stack).
- ▶ **Last-in, first-out (LIFO)** data structures
 - The last item pushed (inserted) on the stack is the first item popped (removed) from the stack.



6.6 Method-Call Stack and Activation Records (Cont.)

- ▶ When a program calls a method, the called method must know how to return to its caller
 - The return address of the calling method is pushed onto the **program-execution** (or **method-call**) **stack**.
- ▶ If a series of method calls occurs, the successive return addresses are pushed onto the stack in last-in, first-out order.
- ▶ The program-execution stack also contains the memory for the local variables used in each invocation of a method during a program's execution.
 - Stored as a portion of the program-execution stack known as the **activation record** or **stack frame** of the method call.



6.6 Method-Call Stack and Activation Records (Cont.)

- ▶ When a method call is made, the activation record for that method call is pushed onto the program-execution stack.
- ▶ When the method returns to its caller, the method's activation record is popped off the stack and those local variables are no longer known to the program.
- ▶ If more method calls occur than can have their activation records stored on the program-execution stack, an error known as a **stack overflow** occurs.



6.7 Argument Promotion and Casting

- ▶ **Argument promotion**
 - Converting an argument's value, if possible, to the type that the method expects to receive in its corresponding parameter.
- ▶ Conversions may lead to compilation errors if Java's **promotion rules** are not satisfied.
- ▶ **Promotion rules**
 - specify which conversions are allowed.
 - apply to expressions containing values of two or more primitive types and to primitive-type values passed as arguments to methods.
- ▶ Each value is promoted to the “highest” type in the expression.
- ▶ Figure 6.4 lists the primitive types and the types to which each can be promoted.



Type	Valid promotions
<code>double</code>	None
<code>float</code>	<code>double</code>
<code>long</code>	<code>float or double</code>
<code>int</code>	<code>long, float or double</code>
<code>char</code>	<code>int, long, float or double</code>
<code>short</code>	<code>int, long, float or double (but not char)</code>
<code>byte</code>	<code>short, int, long, float or double (but not char)</code>
<code>boolean</code>	None (<code>boolean</code> values are not considered to be numbers in Java)

Fig. 6.4 | Promotions allowed for primitive types.



6.7 Argument Promotion and Casting (Cont.)

- ▶ Converting values to types lower in the table of Fig. 6.4 will result in different values if the lower type cannot represent the value of the higher type
- ▶ In cases where information may be lost due to conversion, the Java compiler requires you to use a cast operator to explicitly force the conversion to occur—otherwise a compilation error occurs.



6.9 Case Study: Random-Number Generation

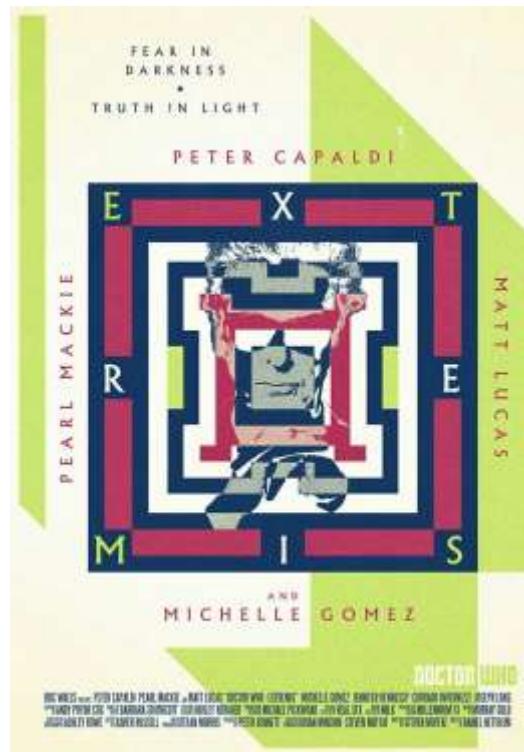
- ▶ Simulation and game playing
 - element of chance
 - Class `Random` (package `java.util`)
 - static method `random` of class `Math`.
- ▶ Objects of class `Random` can produce random `boolean`, `byte`, `float`, `double`, `int`, `long` and Gaussian values
- ▶ `Math` method `random` can produce only `double` values in the range $0.0 \leq x < 1.0$.

`static double random()`

Returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0.

6.9 Case Study: Random-Number Generation (Cont.)

- ▶ Class `Random` produces pseudorandom numbers
 - A sequence of values produced by a complex mathematical calculation.
 - The calculation uses the current time of day to `seed` the random-number generator.





Secure Random

Java's Random class produced *deterministic* values that could be by malicious programmers.

Secure-Random objects produce **nondeterministic random numbers** that *cannot* be predicted.



Creating a SecureRandom Object

A new secure random-number generator object can be created as follows:

```
SecureRandom randomNumbers = new SecureRandom();
```

1



Obtaining a Random int Value

Consider the following statement:

```
int randomValue = randomNumbers.nextInt();
```

All 2^{32} possible int values are produced with (approximately) equal probability

SecureRandom method `nextInt` generates a random int value. If it truly produces values *at random*, then every value in the range should have an *equal chance* (or probability) of being chosen each time `nextInt` is called.



Changing the Range of Values Produced By nextInt

- ▶ Class SecureRandom provides another version of method nextInt that receives an int argument and returns a value from 0 up to, but not including, the argument's value.
- ▶ For example, for coin tossing, the following statement returns 0 or 1.

```
int randomValue = randomNumbers.nextInt(2);
```



SecureRandom Performance

Using `SecureRandom` instead of `Random` to achieve higher levels of security incurs a significant performance penalty. For “casual” applications, you might want to use class `Random` from package **java.util**—simply replace `SecureRandom` with `Random`.



Generalized Scaling and Shifting of Random Numbers

```
int face = 1 + randomNumbers.nextInt(6);
```

The width of the range is determined by the number 6 that's passed as an argument to SecureRandom method nextInt, and the starting number of the range is the number 1 that's added to randomNumbers.nextInt(6). We can generalize this result as

```
int number = shiftingValue + randomNumbers.nextInt(scalingFactor)
```



It's also possible to choose integers at random from sets of values other than ranges of consecutive integers. For example, to obtain a random value from the sequence 2, 5, 8, 11 and 14, you could use the statement

```
int number = 2 + 3 * randomNumbers.nextInt(5);
```

In general,

```
int number = shiftingValue +  
differenceBetweenValues * randomNumbers.nextInt(scalingFactor)
```



6.9.2 Random-Number Repeatability for Testing and Debugging

- ▶ When debugging an application, it's sometimes useful to repeat the exact same sequence of pseudorandom numbers.
- ▶ To do so, create a **Random** object as follows:
 - `Random randomNumbers =
 new Random(seedvalue);`
 - **seedvalue** (of type **long**) seeds the random-number calculation.
- ▶ You can set a **Random** object's seed at any time during program execution by calling the object's **set** method.

`Void setSeed(long seed)`



6.10 Case Study: A Game of Chance; Introducing Enumerations

- ▶ Basic rules for the dice game Craps:
 - *You roll two dice. Each die has six faces, which contain one, two, three, four, five and six spots, respectively. After the dice have come to rest, the sum of the spots on the two upward faces is calculated. If the sum is 7 or 11 on the first throw, you win. If the sum is 2, 3 or 12 on the first throw (called “craps”), you lose (i.e., the “house” wins). If the sum is 4, 5, 6, 8, 9 or 10 on the first throw, that sum becomes your “point.” To win, you must continue rolling the dice until you “make your point” (i.e., roll that same point value). You lose by rolling a 7 before making your point.*



```
1 // Fig. 6.8: Craps.java
2 // Craps class simulates the dice game craps.
3 import java.util.Random;
4
5 public class Craps
6 {
7     // create random number generator for use in method rollDice
8     private static final Random randomNumbers = new Random();
9
10    // enumeration with constants that represent the game status
11    private enum Status { CONTINUE, WON, LOST };
12
13    // constants that represent common rolls of the dice
14    private static final int SNAKE_EYES = 2;
15    private static final int TREY = 3;
16    private static final int SEVEN = 7;
17    private static final int YO_LEVEN = 11;
18    private static final int BOX_CARS = 12;
19
```

Fig. 6.8 | Craps class simulates the dice game craps. (Part I of 5.)



```
20 // plays one game of craps
21 public static void main( String[] args )
22 {
23     int myPoint = 0; // point if no win or loss on first roll
24     Status gameStatus; // can contain CONTINUE, WON or LOST
25
26     int sumOfDice = rollDice(); // first roll of the dice
27
28     // determine game status and point based on first roll
29     switch ( sumOfDice )
30     {
31         case SEVEN: // win with 7 on first roll
32         case YO_LEVEN: // win with 11 on first roll
33             gameStatus = Status.WON;
34             break;
35         case SNAKE_EYES: // lose with 2 on first roll
36         case TREY: // Lose with 3 on first roll
37         case BOX_CARS: // lose with 12 on first roll
38             gameStatus = Status.LOST;
39             break;

```

Fig. 6.8 | Craps class simulates the dice game craps. (Part 2 of 5.)



```
40     default: // did not win or lose, so remember point
41         gameStatus = Status.CONTINUE; // game is not over
42         myPoint = sumOfDice; // remember the point
43         System.out.printf( "Point is %d\n", myPoint );
44         break; // optional at end of switch
45     } // end switch
46
47     // while game is not complete
48     while ( gameStatus == Status.CONTINUE ) // not WON or LOST
49     {
50         sumOfDice = rollDice(); // roll dice again
51
52         // determine game status
53         if ( sumOfDice == myPoint ) // win by making point
54             gameStatus = Status.WON;
55         else
56             if ( sumOfDice == SEVEN ) // lose by rolling 7 before point
57                 gameStatus = Status.LOST;
58     } // end while
59
```

Fig. 6.8 | Craps class simulates the dice game craps. (Part 3 of 5.)



```
60     // display won or lost message
61     if ( gameStatus == Status.WON )
62         System.out.println( "Player wins" );
63     else
64         System.out.println( "Player loses" );
65 } // end main
66
67 // roll dice, calculate sum and display results
68 public static int rollDice()
69 {
70     // pick random die values
71     int die1 = 1 + randomNumbers.nextInt( 6 ); // first die roll
72     int die2 = 1 + randomNumbers.nextInt( 6 ); // second die roll
73
74     int sum = die1 + die2; // sum of die values
75
76     // display results of this roll
77     System.out.printf( "Player rolled %d + %d = %d\n",
78                         die1, die2, sum );
79
80     return sum; // return sum of dice
81 } // end method rollDice
82 } // end class Craps
```

Fig. 6.8 | Craps class simulates the dice game craps. (Part 4 of 5.)



Player rolled 5 + 6 = 11
Player wins

Player rolled 5 + 4 = 9
Point is 9
Player rolled 4 + 2 = 6
Player rolled 3 + 6 = 9
Player wins

Player rolled 1 + 2 = 3
Player loses

Player rolled 2 + 6 = 8
Point is 8
Player rolled 5 + 1 = 6
Player rolled 2 + 1 = 3
Player rolled 1 + 6 = 7
Player loses

Fig. 6.8 | Craps class simulates the dice game craps. (Part 5 of 5.)



6.10 Case Study: A Game of Chance; Introducing Enumerations (Cont.)

- ▶ Notes:
 - **myPoint** is initialized to 0 to ensure that the application will compile.
 - If you do not initialize **myPoint**, the compiler issues an error, because **myPoint** is not assigned a value in every **case** of the **switch** statement, and thus the program could try to use **myPoint** before it is assigned a value.
 - **gameStatus** is assigned a value in every **case** of the **switch** statement—thus, it's guaranteed to be initialized before it's used and does not need to be initialized.



6.10 Case Study: A Game of Chance; Introducing Enumerations (Cont.)

► **enum** type **Status**

- An **enumeration** in its simplest form declares a set of constants represented by identifiers.
- Special kind of class that is introduced by the keyword **enum** and a type name.
- Braces delimit an **enum** declaration's body.
- Inside the braces is a comma-separated list of **enumeration constants**, each representing a unique value.
- The identifiers in an **enum** must be unique.
- Variables of an **enum** type can be assigned only the constants declared in the enumeration.



Good Programming Practice 6.2

Using enumeration constants (like Status.WON, Status.LOST and Status.CONTINUE) rather than literal values (such as 0, 1 and 2) makes programs easier to read and maintain.



6.10 Case Study: A Game of Chance; Introducing Enumerations (Cont.)

- ▶ Why Some Constants Are Not Defined as `enum` Constants
 - Doing so would prevent us from using `sumOfDice` as the `switch` statement's controlling expression, because Java does not allow you to compare an `int` to an `enum` constant.
 - Java does not provide an easy way to convert an `int` value to a particular `enum` constant.
 - Translating an `int` into an `enum` constant could be done with a separate `switch` statement.
 - This would be cumbersome and not improve the readability of the program (thus defeating the purpose of using an `enum`).



6.11 Scope of Declarations

- ▶ Declarations introduce names that can be used to refer to such Java entities.
- ▶ The **scope** of a declaration is the portion of the program that can refer to the declared entity by its name.
 - Such an entity is said to be “in scope” for that portion of the program.



6.11 Scope of Declarations (Cont.)

- ▶ Basic scope rules:
 - The scope of a parameter declaration is the body of the method in which the declaration appears.
 - The scope of a local-variable declaration is from the point at which the declaration appears to the end of that block.
 - The scope of a local-variable declaration that appears in the initialization section of a **for** statement's header is the body of the **for** statement and the other expressions in the header.
 - A method or field's scope is the entire body of the class.
- ▶ Any block may contain variable declarations.
- ▶ If a local variable or parameter in a method has the same name as a field of the class, the field is “hidden” until the block terminates execution—this is called **shadowing**.



Error-Prevention Tip 6.3

Use different names for fields and local variables to help prevent subtle logic errors that occur when a method is called and a local variable of the method shadows a field in the class.



```
1 // Fig. 6.9: Scope.java
2 // Scope class demonstrates field and local variable scopes.
3
4 public class Scope
5 {
6     // field that is accessible to all methods of this class
7     private static int x = 1;
8
9     // method main creates and initializes local variable x
10    // and calls methods useLocalVariable and useField
11    public static void main( String[] args )
12    {
13        int x = 5; // method's local variable x shadows field x
14
15        System.out.printf( "local x in main is %d\n", x );
16
17        useLocalVariable(); // useLocalVariable has local x
18        useField(); // useField uses class Scope's field x
19        useLocalVariable(); // useLocalVariable reinitializes local x
20        useField(); // class Scope's field x retains its value
21
22        System.out.printf( "\nlocal x in main is %d\n", x );
23    } // end main
```

Fig. 6.9 | Scope class demonstrates field and local variable scopes. (Part I of 3.)



```
24
25 // create and initialize local variable x during each call
26 public static void useLocalVariable()
27 {
28     int x = 25; // initialized each time useLocalVariable is called
29
30     System.out.printf(
31         "\nlocal x on entering method useLocalVariable is %d\n", x );
32     ++x; // modifies this method's local variable x
33     System.out.printf(
34         "local x before exiting method useLocalVariable is %d\n", x );
35 } // end method useLocalVariable
36
37 // modify class Scope's field x during each call
38 public static void useField()
39 {
40     System.out.printf(
41         "\nfield x on entering method useField is %d\n", x );
42     x *= 10; // modifies class Scope's field x
43     System.out.printf(
44         "field x before exiting method useField is %d\n", x );
45 } // end method useField
46 } // end class Scope
```

Fig. 6.9 | Scope class demonstrates field and local variable scopes. (Part 2 of 3.)



```
local x in main is 5  
  
local x on entering method useLocalVariable is 25  
local x before exiting method useLocalVariable is 26  
  
field x on entering method useField is 1  
field x before exiting method useField is 10  
  
local x on entering method useLocalVariable is 25  
local x before exiting method useLocalVariable is 26  
  
field x on entering method useField is 10  
field x before exiting method useField is 100  
  
local x in main is 5
```

Fig. 6.9 | Scope class demonstrates field and local variable scopes. (Part 3 of 3.)



6.12 Method Overloading

- ▶ **Method overloading**
 - Methods of the same name declared in the same class
 - Must have different sets of parameters
- ▶ Compiler selects the appropriate method to call by examining the number, types and order of the arguments in the call.
- ▶ Used to create several methods with the same name that perform the same or similar tasks, but on different types or different numbers of arguments.



```
1 // Fig. 6.10: MethodOverload.java
2 // Overloaded method declarations.
3
4 public class MethodOverload
{
    // test overloaded square methods
    public static void main( String[] args )
    {
        System.out.printf( "Square of integer 7 is %d\n", square( 7 ) );
        System.out.printf( "Square of double 7.5 is %f\n", square( 7.5 ) );
    } // end main
12
13 // square method with int argument
14 public static int square( int intValue )
15 {
    System.out.printf( "\nCalled square with int argument: %d\n",
17     intValue );
    return intValue * intValue;
19 } // end method square with int argument
20
```

Fig. 6.10 | Overloaded method declarations. (Part I of 2.)

- Literal integer values are treated as type **int**, so the method call in line 9 invokes the version of **square** that specifies an **int** parameter.
- Literal floating-point values are treated as type **double**, so the method call in line 10 invokes the version of **square** that specifies a **double** parameter.



```
21 // square method with double argument
22 public static double square( double doubleValue )
23 {
24     System.out.printf( "\nCalled square with double argument: %f\n",
25                         doubleValue );
26     return doubleValue * doubleValue;
27 } // end method square with double argument
28 } // end class MethodOverload
```

```
Called square with int argument: 7
Square of integer 7 is 49
```

```
Called square with double argument: 7.500000
Square of double 7.5 is 56.250000
```

Fig. 6.10 | Overloaded method declarations. (Part 2 of 2.)



6.12 Method Overloading (cont.)

- ▶ Distinguishing Between Overloaded Methods
 - The compiler distinguishes overloaded methods by their **signatures**—the methods' names and the **number, types and order** of their parameters.
- ▶ Return types of overloaded methods
 - *Method calls cannot be distinguished by return type.*
- ▶ Overloaded methods can have different return types if the methods have different parameter lists.
- ▶ Overloaded methods need not have the same number of parameters.



Common Programming Error 6.9

Declaring overloaded methods with identical parameter lists is a compilation error regardless of whether the return types are different.



Chapter 9

Object-Oriented Programming:

Inheritance

Java™ How to Program, 9/e



OBJECTIVES

In this chapter you'll learn:

- How inheritance promotes software reusability.
- The notions of superclasses and subclasses and the relationship between them.
- To use keyword **extends** to create a class that inherits attributes and behaviors from another class.
- To use access modifier **protected** to give subclass methods access to superclass members.
- To access superclass members with **super**.
- How constructors are used in inheritance hierarchies.
- The methods of class **Object**, the direct or indirect superclass of all classes.



9.1 Introduction

9.2 Superclasses and Subclasses

9.3 **protected** Members

9.4 Relationship between Superclasses and Subclasses

9.4.1 Creating and Using a **CommissionEmployee** Class

9.4.2 Creating and Using a **BasePlusCommissionEmployee** Class

9.4.3 Creating a **CommissionEmployee–BasePlusCommissionEmployee** Inheritance Hierarchy

9.4.4 **CommissionEmployee–BasePlusCommissionEmployee** Inheritance Hierarchy Using
protected Instance Variables

9.4.5 **CommissionEmployee–BasePlusCommissionEmployee** Inheritance Hierarchy Using
private Instance Variables

9.5 Constructors in Subclasses

9.6 Software Engineering with Inheritance

9.7 **Object** Class

9.8 (Optional) GUI and Graphics Case Study: Displaying Text and Images Using **Labels**

9.9 Wrap-Up



```
21     System.out.printf(
22         "Date object constructor for date %s\n", this );
23 } // end Date constructor
24
25 // utility method to confirm proper month value
26 private int checkMonth( int testMonth )
27 {
28     if ( testMonth > 0 && testMonth <= 12 ) // validate month
29         return testMonth;
30     else // month is invalid
31         throw new IllegalArgumentException( "month must be 1-12" );
32 } // end method checkMonth
33
34 // utility method to confirm proper day value based on month and year
35 private int checkDay( int testDay )
36 {
37     // check if day in range for month
38     if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
39         return testDay;
40 }
```

Fig. 8.7 | Date class declaration. (Part 2 of 3.)



```
41     // check for leap year
42     if ( month == 2 && testDay == 29 && ( year % 400 == 0 ||
43         ( year % 4 == 0 && year % 100 != 0 ) ) )
44         return testDay;
45
46     throw new IllegalArgumentException(
47         "day out-of-range for the specified month and year" );
48 } // end method checkDay
49
50 // return a String of the form month/day/year
51 public String toString()
52 {
53     return String.format( "%d/%d/%d", month, day, year );
54 } // end method toString
55 } // end class Date
```

Fig. 8.7 | Date class declaration. (Part 3 of 3.)



```
1 // Fig. 8.8: Employee.java
2 // Employee class with references to other objects.
3
4 public class Employee
5 {
6     private String firstName;
7     private String lastName;
8     private Date birthDate;
9     private Date hireDate;
10
11    // constructor to initialize name, birth date and hire date
12    public Employee( String first, String last, Date dateOfBirth,
13                      Date dateOfHire )
14    {
15        firstName = first;
16        lastName = last;
17        birthDate = dateOfBirth;
18        hireDate = dateOfHire;
19    } // end Employee constructor
20
```

Fig. 8.8 | Employee class with references to other objects. (Part 1 of 2.)



```
21 // convert Employee to String format
22 public String toString()
23 {
24     return String.format( "%s, %s Hired: %s Birthday: %s",
25             lastName, firstName, hireDate, birthDate );
26 } // end method toString
27 } // end class Employee
```

Fig. 8.8 | Employee class with references to other objects. (Part 2 of 2.)



```
1 // Fig. 8.9: EmployeeTest.java
2 // Composition demonstration.
3
4 public class EmployeeTest
5 {
6     public static void main( String[] args )
7     {
8         Date birth = new Date( 7, 24, 1949 );
9         Date hire = new Date( 3, 12, 1988 );
10        Employee employee = new Employee( "Bob", "Blue", birth, hire );
11
12        System.out.println( employee );
13    } // end main
14 } // end class EmployeeTest
```

```
Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988
Blue, Bob Hired: 3/12/1988 Birthday: 7/24/1949
```

Fig. 8.9 | Composition demonstration.



9.1 Introduction

► Inheritance

- A form of software reuse in which a new class is created by absorbing an existing class's members and embellishing them with new or modified capabilities.
- Can save time during program development by basing new classes on existing proven and debugged high-quality software.
- Increases the likelihood that a system will be implemented and maintained effectively.



9.1 Introduction (Cont.)

- ▶ When creating a class, rather than declaring completely new members, you can designate that the new class should inherit the members of an existing class.
 - Existing class is the **superclass**
 - New class is the **subclass**
- ▶ Each subclass can be a superclass of future subclasses.
- ▶ A subclass can add its own fields and methods.
- ▶ A subclass is more specific than its superclass and represents a more specialized group of objects.
- ▶ The subclass exhibits the behaviors of its superclass and can add behaviors that are specific to the subclass.
 - This is why inheritance is sometimes referred to as **specialization**.



9.1 Introduction (Cont.)

- ▶ The **direct superclass** is the superclass from which the subclass explicitly inherits.
- ▶ An **indirect superclass** is any class above the direct superclass in the **class hierarchy**.
- ▶ The Java class hierarchy begins with class **Object** (in package **java.lang**)
 - *Every* class in Java directly or indirectly **extends** (or “inherits from”) **Object**.
- ▶ Java supports only **single inheritance**, in which each class is derived from exactly one direct superclass.



9.1 Introduction (Cont.)

- ▶ We distinguish between the **is-a relationship** and the **has-a relationship**
- ▶ *Is-a* represents inheritance
 - In an *is-a* relationship, an object of a subclass can also be treated as an object of its superclass
- ▶ *Has-a* represents composition
 - In a *has-a* relationship, an object contains as members references to other objects



9.2 Superclasses and Subclasses

- ▶ Figure 9.1 lists several simple examples of superclasses and subclasses
 - Superclasses tend to be “more general” and subclasses “more specific.”
- ▶ Because every subclass object *is an* object of its superclass, and one superclass can have many subclasses, the set of objects represented by a superclass is typically larger than the set of objects represented by any of its subclasses.



Superclass	Subclasses
Student	GraduateStudent, UndergraduateStudent
Shape	Circle, Triangle, Rectangle, Sphere, Cube
Loan	CarLoan, HomeImprovementLoan, MortgageLoan
Employee	Faculty, Staff
BankAccount	CheckingAccount, SavingsAccount

Fig. 9.1 | Inheritance examples.



9.2 Superclasses and Subclasses (Cont.)

- ▶ A superclass exists in a hierarchical relationship with its subclasses.
- ▶ Fig. 9.2 shows a sample university community class hierarchy
 - Also called an **inheritance hierarchy**.
- ▶ Each arrow in the hierarchy represents an *is-a relationship*.
- ▶ Follow the arrows upward in the class hierarchy
 - an **Employee** *is a* **CommunityMember**”
 - “**a Teacher** *is a* **Faculty** member.”
- ▶ **CommunityMember** is the direct superclass of **Employee**, **Student** and **Alumnus** and is an indirect superclass of all the other classes in the diagram.
- ▶ Starting from the bottom, you can follow the arrows and apply the *is-a* relationship up to the topmost superclass.

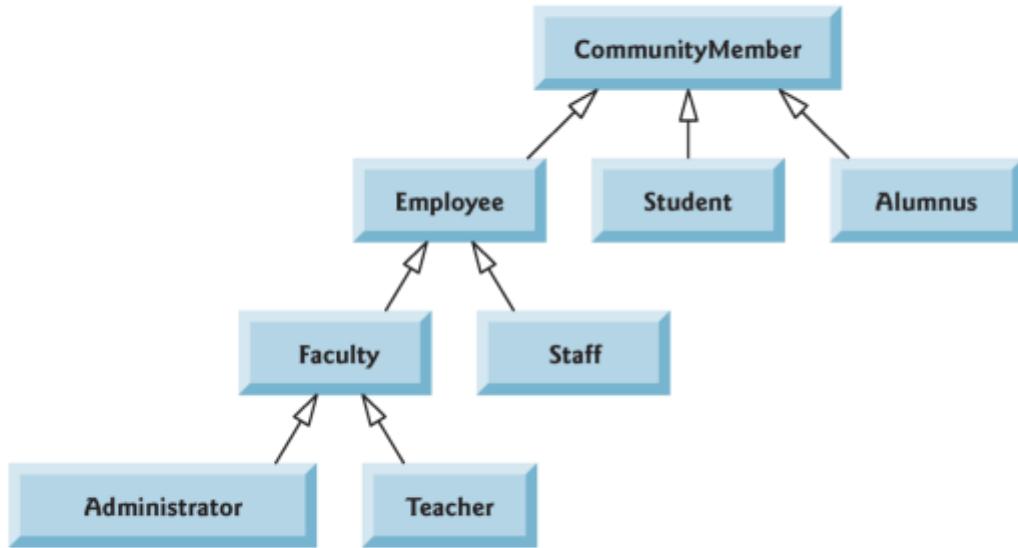


Fig. 9.2 | Inheritance hierarchy for university **CommunityMembers**.



9.2 Superclasses and Subclasses (Cont.)

- ▶ Fig. 9.3 shows a **Shape** inheritance hierarchy.
- ▶ You can follow the arrows from the bottom of the diagram to the topmost superclass in this class hierarchy to identify several *is-a* relationships.
 - A **Triangle** *is a* **TwoDimensionalShape** and *is a* **Shape**
 - A **Sphere** *is a* **ThreeDimensionalShape** and *is a* **Shape**.

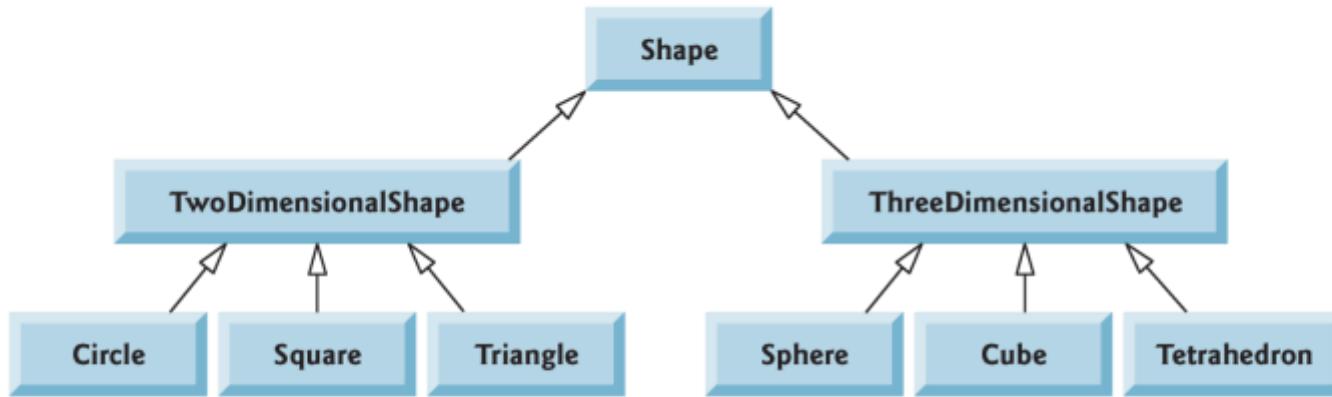


Fig. 9.3 | Inheritance hierarchy for Shapes.



9.2 Superclasses and Subclasses (Cont.)

- ▶ Not every class relationship is an inheritance relationship.
- ▶ *Has-a* relationship
 - Create classes by composition of existing classes.
 - Example: Given the classes **Employee**, **BirthDate** and **PhoneNumber**, it's improper to say that an **Employee** *is a* **BirthDate** or that an **Employee** *is a* **PhoneNumber**.
 - However, an **Employee** *has a* **BirthDate**, and an **Employee** *has a* **PhoneNumber**.



9.2 Superclasses and Subclasses (Cont.)

- ▶ Objects of all classes that extend a common superclass can be treated as objects of that superclass.
 - Commonality expressed in the members of the superclass.
- ▶ Inheritance issue
 - A subclass can inherit methods that it does not need or should not have.
 - Even when a superclass method is appropriate for a subclass, that subclass often needs a customized version of the method.
 - The subclass can **override** (redefine) the superclass method with an appropriate implementation.



9.3 protected Members

- ▶ A class's **public** members are accessible wherever the program has a reference to an object of that class or one of its subclasses.
- ▶ A class's **private** members are accessible only within the class itself.
- ▶ **protected** access is an intermediate level of access between **public** and **private**.
 - A superclass's **protected** members can be accessed by members of that superclass, by members of its subclasses and by members of other classes in the same package
 - **protected** members also have package access.
 - All **public** and **protected** superclass members retain their original access modifier when they become members of the subclass.



9.3 protected Members (Cont.)

- ▶ A superclass's **private** members are hidden in its subclasses
 - They can be accessed only through the **public** or **protected** methods inherited from the superclass
- ▶ Subclass methods can refer to **public** and **protected** members inherited from the superclass simply by using the member names.
- ▶ When a subclass method overrides an inherited superclass method, the superclass method can be accessed from the subclass by preceding the superclass method name with keyword **super** and a dot (.) separator.



Software Engineering Observation 9.1

*Methods of a subclass cannot directly access **private** members of their superclass. A subclass can change the state of **private** superclass instance variables only through non-**private** methods provided in the superclass and inherited by the subclass.*



Software Engineering Observation 9.2

Declaring private instance variables helps you test, debug and correctly modify systems. If a subclass could access its superclass's private instance variables, classes that inherit from that subclass could access the instance variables as well. This would propagate access to what should be private instance variables, and the benefits of information hiding would be lost.

105



9.4 Relationship between Superclasses and Subclasses

- ▶ Inheritance hierarchy containing types of employees in a company's payroll application
- ▶ Commission employees are paid a percentage of their sales
- ▶ Base-salaried commission employees receive a base salary plus a percentage of their sales.



9.4.1 Creating and Using a CommissionEmployee Class

- ▶ Class **CommissionEmployee** (Fig. 9.4) **extends** class **Object** (from package **java.lang**).
 - **CommissionEmployee** inherits **Object**'s methods.
 - If you don't explicitly specify which class a new class extends, the class extends **Object** implicitly.



```
1 // Fig. 9.4: CommissionEmployee.java
2 // CommissionEmployee class represents an employee paid a
3 // percentage of gross sales.
4 public class CommissionEmployee extends Object
5 {
6     private String firstName;
7     private String lastName;
8     private String socialSecurityNumber;
9     private double grossSales; // gross weekly sales
10    private double commissionRate; // commission percentage
11
12    // five-argument constructor
13    public CommissionEmployee( String first, String last, String ssn,
14        double sales, double rate )
15    {
16        // implicit call to Object constructor occurs here
17        firstName = first;
18        lastName = last;
19        socialSecurityNumber = ssn;
20        setGrossSales( sales ); // validate and store gross sales
21        setCommissionRate( rate ); // validate and store commission rate
22    } // end five-argument CommissionEmployee constructor
```

Fig. 9.4 | CommissionEmployee class represents an employee paid a percentage of gross sales. (Part I of 6.)



```
23
24     // set first name
25     public void setFirstName( String first )
26     {
27         firstName = first; // should validate
28     } // end method setFirstName
29
30     // return first name
31     public String getFirstName()
32     {
33         return firstName;
34     } // end method getFirstName
35
36     // set last name
37     public void setLastName( String last )
38     {
39         lastName = last; // should validate
40     } // end method setLastName
41
```

Fig. 9.4 | CommissionEmployee class represents an employee paid a percentage of gross sales. (Part 2 of 6.)



```
42     // return last name
43     public String getLastname()
44     {
45         return lastName;
46     } // end method getLastname
47
48     // set social security number
49     public void setSocialSecurityNumber( String ssn )
50     {
51         socialSecurityNumber = ssn; // should validate
52     } // end method setSocialSecurityNumber
53
54     // return social security number
55     public String getSocialSecurityNumber()
56     {
57         return socialSecurityNumber;
58     } // end method getSocialSecurityNumber
59
```

Fig. 9.4 | CommissionEmployee class represents an employee paid a percentage of gross sales. (Part 3 of 6.)



```
60 // set gross sales amount
61 public void setGrossSales( double sales )
62 {
63     if ( sales >= 0.0 )
64         grossSales = sales;
65     else
66         throw new IllegalArgumentException(
67             "Gross sales must be >= 0.0" );
68 } // end method setGrossSales
69
70 // return gross sales amount
71 public double getGrossSales()
72 {
73     return grossSales;
74 } // end method getGrossSales
75
```

Fig. 9.4 | CommissionEmployee class represents an employee paid a percentage of gross sales. (Part 4 of 6.)



```
76 // set commission rate
77 public void setCommissionRate( double rate )
78 {
79     if ( rate > 0.0 && rate < 1.0 )
80         commissionRate = rate;
81     else
82         throw new IllegalArgumentException(
83             "Commission rate must be > 0.0 and < 1.0" );
84 } // end method setCommissionRate
85
86 // return commission rate
87 public double getCommissionRate()
88 {
89     return commissionRate;
90 } // end method getCommissionRate
91
92 // calculate earnings
93 public double earnings()
94 {
95     return commissionRate * grossSales;
96 } // end method earnings
97
```

Fig. 9.4 | CommissionEmployee class represents an employee paid a percentage of gross sales. (Part 5 of 6.)



```
98 // return String representation of CommissionEmployee object
99 @Override // indicates that this method overrides a superclass method
100 public String toString()
101 {
102     return String.format( "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f",
103                         "commission employee", firstName, lastName,
104                         "social security number", socialSecurityNumber,
105                         "gross sales", grossSales,
106                         "commission rate", commissionRate );
107 } // end method toString
108 } // end class CommissionEmployee
```

Fig. 9.4 | CommissionEmployee class represents an employee paid a percentage of gross sales. (Part 6 of 6.)



9.4.1 Creating and Using a CommissionEmployee Class (Cont.)

- ▶ Constructors are not inherited.
- ▶ The first task of a subclass constructor is to call its direct superclass's constructor explicitly or implicitly
 - Ensures that the instance variables inherited from the superclass are initialized properly.
- ▶ If the code does not include an explicit call to the superclass constructor, Java implicitly calls the superclass's default or no-argument constructor.
- ▶ A class's default constructor calls the superclass's default or no-argument constructor.



9.4.1 Creating and Using a CommissionEmployee Class (Cont.)

- ▶ **toString** is one of the methods that every class inherits directly or indirectly from class **Object**.
 - Returns a **String** representing an object.
 - Called implicitly whenever an object must be converted to a **String** representation.
- ▶ Class **Object**'s **toString** method returns a **String** that includes the name of the object's class.
 - This is primarily a placeholder that can be overridden by a subclass to specify an appropriate **String** representation.



9.4.1 Creating and Using a CommissionEmployee Class (Cont.)

- ▶ To override a superclass method, a subclass must declare a method with the same signature as the superclass method
- ▶ **@Override annotation**
 - Indicates that a method should override a superclass method with the same signature.
 - If it does not, a compilation error occurs.



Common Programming Error 9.1

Using an incorrect method signature when attempting to override a superclass method causes an unintentional method overload that can lead to subtle logic errors.



Error-Prevention Tip 9.1

Declare overridden methods with the @Override annotation to ensure at compilation time that you defined their signatures correctly. It's always better to find errors at compile time rather than at runtime.



Common Programming Error 9.2

It's a syntax error to override a method with a more restricted access modifier—a `public` method of the superclass cannot become a `protected` or `private` method in the subclass; a `protected` method of the superclass cannot become a `private` method in the subclass. Doing so would break the is-a relationship in which it's required that all subclass objects be able to respond to method calls that are made to `public` methods declared in the superclass. If a `public` method, for example, could be overridden as a `protected` or `private` method, the subclass objects would not be able to respond to the same method calls as superclass objects. Once a method is declared `public` in a superclass, the method remains `public` for all that class's direct and indirect subclasses.



```
1 // Fig. 9.5: CommissionEmployeeTest.java
2 // CommissionEmployee class test program.
3
4 public class CommissionEmployeeTest
{
    6     public static void main( String[] args )
    7     {
        8         // instantiate CommissionEmployee object
        9         CommissionEmployee employee = new CommissionEmployee(
10             "Sue", "Jones", "222-22-2222", 10000, .06 );
11
12         // get commission employee data
13         System.out.println(
14             "Employee information obtained by get methods: \n" );
15         System.out.printf( "%s %s\n", "First name is",
16             employee.getFirstName() );
17         System.out.printf( "%s %s\n", "Last name is",
18             employee.getLastName() );
19         System.out.printf( "%s %s\n", "Social security number is",
20             employee.getSocialSecurityNumber() );
21         System.out.printf( "%s %.2f\n", "Gross sales is",
22             employee.getGrossSales() );
23         System.out.printf( "%s %.2f\n", "Commission rate is",
24             employee.getCommissionRate() );
```

Fig. 9.5 | CommissionEmployee class test program. (Part I of 2.)



```
25  
26     employee.setGrossSales( 500 ); // set gross sales  
27     employee.setCommissionRate( .1 ); // set commission rate  
28  
29     System.out.printf( "\n%s:\n\n%s\n",  
30                         "Updated employee information obtained by toString", employee );  
31 } // end main  
32 } // end class CommissionEmployeeTest
```

Implicit `toString` call occurs here

Employee information obtained by get methods:

First name is Sue
Last name is Jones
Social security number is 222-22-2222
Gross sales is 10000.00
Commission rate is 0.06

Updated employee information obtained by `toString`:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 500.00
commission rate: 0.10

Fig. 9.5 | `CommissionEmployee` class test program. (Part 2 of 2.)



9.4.2 Creating and Using a BasePlusCommissionEmployee Class

- ▶ Class `BasePlusCommissionEmployee` (Fig. 9.6) contains a first name, last name, social security number, gross sales amount, commission rate and base salary.
 - All but the base salary are in common with class `CommissionEmployee`.
- ▶ Class `BasePlusCommissionEmployee`'s `public` services include a constructor, and methods `earnings`, `toString` and `get` and `set` for each instance variable
 - Most of these are in common with class `CommissionEmployee`.



```
1 // Fig. 9.6: BasePlusCommissionEmployee.java
2 // BasePlusCommissionEmployee class represents an employee who receives
3 // a base salary in addition to commission.
4
5 public class BasePlusCommissionEmployee
6 {
7     private String firstName;
8     private String lastName;
9     private String socialSecurityNumber;
10    private double grossSales; // gross weekly sales
11    private double commissionRate; // commission percentage
12    private double baseSalary; // base salary per week
13
14    // six-argument constructor
15    public BasePlusCommissionEmployee( String first, String last,
16        String ssn, double sales, double rate, double salary )
17    {
18        // implicit call to Object constructor occurs here
19        firstName = first;
20        lastName = last;
21        socialSecurityNumber = ssn;
22        setGrossSales( sales ); // validate and store gross sales
```

Fig. 9.6 | BasePlusCommissionEmployee class represents an employee who receives a base salary in addition to a commission. (Part I of 7.)



```
23     setCommissionRate( rate ); // validate and store commission rate
24     setBaseSalary( salary ); // validate and store base salary
25 } // end six-argument BasePlusCommissionEmployee constructor
26
27 // set first name
28 public void setFirstName( String first )
29 {
30     firstName = first; // should validate
31 } // end method setFirstName
32
33 // return first name
34 public String getFirstName()
35 {
36     return firstName;
37 } // end method getFirstName
38
39 // set last name
40 public void setLastName( String last )
41 {
42     lastName = last; // should validate
43 } // end method setLastName
44
```

Fig. 9.6 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 2 of 7.)



```
45 // return last name
46 public String getLastname()
47 {
48     return lastName;
49 } // end method getLastname
50
51 // set social security number
52 public void setSocialSecurityNumber( String ssn )
53 {
54     socialSecurityNumber = ssn; // should validate
55 } // end method setSocialSecurityNumber
56
57 // return social security number
58 public String getSocialSecurityNumber()
59 {
60     return socialSecurityNumber;
61 } // end method getSocialSecurityNumber
62
```

Fig. 9.6 | BasePlusCommissionEmployee class represents an employee who receives a base salary in addition to a commission. (Part 3 of 7.)



```
63 // set gross sales amount
64 public void setGrossSales( double sales )
65 {
66     if ( sales >= 0.0 )
67         grossSales = sales;
68     else
69         throw new IllegalArgumentException(
70             "Gross sales must be >= 0.0" );
71 } // end method setGrossSales
72
73 // return gross sales amount
74 public double getGrossSales()
75 {
76     return grossSales;
77 } // end method getGrossSales
78
```

Fig. 9.6 | BasePlusCommissionEmployee class represents an employee who receives a base salary in addition to a commission. (Part 4 of 7.)



```
79 // set commission rate
80 public void setCommissionRate( double rate )
81 {
82     if ( rate > 0.0 && rate < 1.0 )
83         commissionRate = rate;
84     else
85         throw new IllegalArgumentException(
86             "Commission rate must be > 0.0 and < 1.0" );
87 } // end method setCommissionRate
88
89 // return commission rate
90 public double getCommissionRate()
91 {
92     return commissionRate;
93 } // end method getCommissionRate
94
```

Fig. 9.6 | BasePlusCommissionEmployee class represents an employee who receives a base salary in addition to a commission. (Part 5 of 7.)



```
95 // set base salary
96 public void setBaseSalary( double salary )
97 {
98     if ( salary >= 0.0 )
99         baseSalary = salary;
100    else
101        throw new IllegalArgumentException(
102            "Base salary must be >= 0.0" );
103    } // end method setBaseSalary
104
105 // return base salary
106 public double getBaseSalary()
107 {
108     return baseSalary;
109 } // end method getBaseSalary
110
111 // calculate earnings
112 public double earnings()
113 {
114     return baseSalary + ( commissionRate * grossSales );
115 } // end method earnings
116
```

Fig. 9.6 | BasePlusCommissionEmployee class represents an employee who receives a base salary in addition to a commission. (Part 6 of 7.)



```
117 // return String representation of BasePlusCommissionEmployee
118 @Override // indicates that this method overrides a superclass method
119 public String toString()
120 {
121     return String.format(
122         "%s: %s %s\n%s: %.2f\n%s: %.2f\n%s: %.2f",
123         "base-salaried commission employee", firstName, lastName,
124         "social security number", socialSecurityNumber,
125         "gross sales", grossSales, "commission rate", commissionRate,
126         "base salary", baseSalary );
127 } // end method toString
128 } // end class BasePlusCommissionEmployee
```

Fig. 9.6 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 7 of 7.)



9.4.2 Creating and Using a BasePlusCommissionEmployee Class (Cont.)

- ▶ Class `BasePlusCommissionEmployee` does not specify “extends Object”
 - Implicitly extends `Object`.
- ▶ `BasePlusCommissionEmployee`’s constructor invokes class `Object`’s default constructor implicitly.



```
1 // Fig. 9.7: BasePlusCommissionEmployeeTest.java
2 // BasePlusCommissionEmployee test program.
3
4 public class BasePlusCommissionEmployeeTest
5 {
6     public static void main( String[] args )
7     {
8         // instantiate BasePlusCommissionEmployee object
9         BasePlusCommissionEmployee employee =
10            new BasePlusCommissionEmployee(
11                "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
12
13        // get base-salaried commission employee data
14        System.out.println(
15            "Employee information obtained by get methods: \n" );
16        System.out.printf( "%s %s\n", "First name is",
17            employee.getFirstName() );
18        System.out.printf( "%s %s\n", "Last name is",
19            employee.getLastName() );
20        System.out.printf( "%s %s\n", "Social security number is",
21            employee.getSocialSecurityNumber() );
22        System.out.printf( "%s %.2f\n", "Gross sales is",
23            employee.getGrossSales() );
```

Fig. 9.7 | BasePlusCommissionEmployee test program. (Part I of 3.)



```
24     System.out.printf( "%s %.2f\n", "Commission rate is",
25         employee.getCommissionRate() );
26     System.out.printf( "%s %.2f\n", "Base salary is",
27         employee.getBaseSalary() );
28
29     employee.setBaseSalary( 1000 ); // set base salary
30
31     System.out.printf( "\n%s:\n\n%s\n",
32         "Updated employee information obtained by toString",
33         employee.toString() );
34 } // end main
35 } // end class BasePlusCommissionEmployeeTest
```

Fig. 9.7 | BasePlusCommissionEmployee test program. (Part 2 of 3.)



Employee information obtained by get methods:

```
First name is Bob  
Last name is Lewis  
Social security number is 333-33-3333  
Gross sales is 5000.00  
Commission rate is 0.04  
Base salary is 300.00
```

Updated employee information obtained by toString:

```
base-salaried commission employee: Bob Lewis  
social security number: 333-33-3333  
gross sales: 5000.00  
commission rate: 0.04  
base salary: 1000.00
```

Fig. 9.7 | BasePlusCommissionEmployee test program. (Part 3 of 3.)



9.4.2 Creating and Using a BasePlusCommissionEmployee Class (Cont.)

- ▶ Much of `BasePlusCommissionEmployee`'s code is similar, or identical, to that of `CommissionEmployee`.
- ▶ `private` instance variables `firstName` and `lastName` and methods `setFirstName`, `getFirstName`, `setLastName` and `getLastName` are identical.
 - Both classes also contain corresponding `get` and `set` methods.
- ▶ The constructors are almost identical
 - `BasePlusCommissionEmployee`'s constructor also sets the base-Salary.
- ▶ The `toString` methods are nearly identical
 - `BasePlusCommissionEmployee`'s `toString` also outputs instance variable `baseSalary`



9.4.2 Creating and Using a BasePlusCommissionEmployee Class (Cont.)

- ▶ We literally *copied* CommissionEmployee's code, pasted it into **BasePlusCommissionEmployee**, then modified the new class to include a base salary and methods that manipulate the base salary.
 - This “copy-and-paste” approach is often error prone and time consuming.
 - It spreads copies of the same code throughout a system, creating a code-maintenance nightmare.



Software Engineering Observation 9.3

With inheritance, the common instance variables and methods of all the classes in the hierarchy are declared in a superclass. When changes are made for these common features in the superclass—subclasses then inherit the changes. Without inheritance, changes would need to be made to all the source-code files that contain a copy of the code in question.

9.4.3 Creating a CommissionEmployee- BasePlusCommissionEmployee Inheritance Hierarchy



- ▶ Class **BasePlusCommissionEmployee** class extends class **CommissionEmployee**
- ▶ A **BasePlusCommissionEmployee** object *is a* **CommissionEmployee**
 - Inheritance passes on class **CommissionEmployee**'s capabilities.
- ▶ Class **BasePlusCommissionEmployee** also has instance variable **baseSalary**.
- ▶ Subclass **BasePlusCommissionEmployee** inherits **CommissionEmployee**'s instance variables and methods
 - Only the superclass's **public** and **protected** members are directly accessible in the subclass.



```
1 // Fig. 9.8: BasePlusCommissionEmployee.java
2 // private superclass members cannot be accessed in a subclass.
3
4 public class BasePlusCommissionEmployee extends CommissionEmployee
5 {
6     private double baseSalary; // base salary per week
7
8     // six-argument constructor
9     public BasePlusCommissionEmployee( String first, String last,
10         String ssn, double sales, double rate, double salary )
11    {
12        // explicit call to superclass CommissionEmployee constructor
13        super( first, last, ssn, sales, rate );
14
15        setBaseSalary( salary ); // validate and store base salary
16    } // end six-argument BasePlusCommissionEmployee constructor
17
```

Fig. 9.8 | private superclass members cannot be accessed in a subclass. (Part I of 5.)



```
18 // set base salary
19 public void setBaseSalary( double salary )
20 {
21     if ( salary >= 0.0 )
22         baseSalary = salary;
23     else
24         throw new IllegalArgumentException(
25             "Base salary must be >= 0.0" );
26 } // end method setBaseSalary
27
28 // return base salary
29 public double getBaseSalary()
30 {
31     return baseSalary;
32 } // end method getBaseSalary
33
34 // calculate earnings
35 @Override // indicates that this method overrides a superclass method
36 public double earnings()
37 {
38     // not allowed: commissionRate and grossSales private in superclass
39     return baseSalary + ( commissionRate * grossSales );
40 } // end method earnings
```

Fig. 9.8 | private superclass members cannot be accessed in a subclass. (Part 2 of 5.)



```
41
42     // return String representation of BasePlusCommissionEmployee
43     @Override // indicates that this method overrides a superclass method
44     public String toString()
45     {
46         // not allowed: attempts to access private superclass members
47         return String.format(
48             "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f\n%s: %.2f",
49             "base-salaried commission employee", firstName, lastName,
50             "social security number", socialSecurityNumber,
51             "gross sales", grossSales, "commission rate", commissionRate,
52             "base salary", baseSalary );
53     } // end method toString
54 } // end class BasePlusCommissionEmployee
```

Fig. 9.8 | private superclass members cannot be accessed in a subclass. (Part 3 of 5.)



```
BasePlusCommissionEmployee.java:39: commissionRate has private access in
CommissionEmployee
    return baseSalary + ( commissionRate * grossSales );
               ^
BasePlusCommissionEmployee.java:39: grossSales has private access in
CommissionEmployee
    return baseSalary + ( commissionRate * grossSales );
               ^
BasePlusCommissionEmployee.java:49: firstName has private access in
CommissionEmployee
    "base-salaried commission employee", firstName, lastName,
               ^
BasePlusCommissionEmployee.java:49: lastName has private access in
CommissionEmployee
    "base-salaried commission employee", firstName, lastName,
               ^
BasePlusCommissionEmployee.java:50: socialSecurityNumber has private access
in CommissionEmployee
    "social security number", socialSecurityNumber,
               ^
BasePlusCommissionEmployee.java:51: grossSales has private access in
CommissionEmployee
    "gross sales", grossSales, "commission rate", commissionRate,
               ^
```

Fig. 9.8 | private superclass members cannot be accessed in a subclass. (Part 4 of 5.)



```
BasePlusCommissionEmployee.java:51: commissionRate has private access in  
CommissionEmployee  
    "gross sales", grossSales, "commission rate", commissionRate,  
                           ^  
7 errors
```

Fig. 9.8 | private superclass members cannot be accessed in a subclass. (Part 5 of 5.)

9.4.3 Creating a CommissionEmployee- BasePlusCommissionEmployee Inheritance Hierarchy (Cont.)



- ▶ Each subclass constructor must implicitly or explicitly call its superclass constructor to initialize the instance variables inherited from the superclass.
 - **Superclass constructor call syntax**—keyword `super`, followed by a set of parentheses containing the superclass constructor arguments.
 - Must be the first statement in the subclass constructor's body.
- ▶ If the subclass constructor did not invoke the superclass's constructor explicitly, Java would attempt to invoke the superclass's no-argument or default constructor.
 - Class `CommissionEmployee` does not have such a constructor, so the compiler would issue an error.
- ▶ You can explicitly use `super()` to call the superclass's no-argument or default constructor, but this is rarely done.

9.4.3 Creating a CommissionEmployee- BasePlusCommissionEmployee Inheritance Hierarchy (Cont.)



- ▶ Compilation errors occur when the subclass attempts to access the superclass's **private** instance variables.
- ▶ These lines could have used appropriate *get* methods to retrieve the values of the superclass's instance variables.



9.4.4 CommissionEmployee– BasePlusCommissionEmployee Inheritance Hierarchy Using **protected** Instance Variables

- ▶ To enable a subclass to directly access superclass instance variables, we can declare those members as **protected** in the superclass.
- ▶ New **CommissionEmployee** class modified only lines 6–10 of Fig. 9.4 as follows:

```
protected String firstName;
protected String lastName;
protected String socialSecurityNumber;
protected double grossSales;
protected double commissionRate;
```

- ▶ With **protected** instance variables, the subclass gets access to the instance variables, but classes that are not subclasses and classes that are not in the same package cannot access these variables directly.

9.4.4 CommissionEmployee–BasePlusCommissionEmployee Inheritance Hierarchy Using protected Instance Variables (Cont.)



- ▶ Class **BasePlusCommissionEmployee** (Fig. 9.9) extends the new version of class **CommissionEmployee** with **protected** instance variables.
 - These variables are now **protected** members of **BasePlusCommissionEmployee**.
- ▶ If another class extends this version of class **BasePlusCommissionEmployee**, the new subclass also can access the **protected** members.
- ▶ The source code in Fig. 9.9 (51 lines) is considerably shorter than that in Fig. 9.6 (128 lines)
 - Most of the functionality is now inherited from **CommissionEmployee**
 - There is now only one copy of the functionality.
 - Code is easier to maintain, modify and debug—the code related to a commission employee exists only in class **CommissionEmployee**.



```
1 // Fig. 9.9: BasePlusCommissionEmployee.java
2 // BasePlusCommissionEmployee inherits protected instance
3 // variables from CommissionEmployee.
4
5 public class BasePlusCommissionEmployee extends CommissionEmployee
6 {
7     private double baseSalary; // base salary per week
8
9     // six-argument constructor
10    public BasePlusCommissionEmployee( String first, String last,
11        String ssn, double sales, double rate, double salary )
12    {
13        super( first, last, ssn, sales, rate );
14        setBaseSalary( salary ); // validate and store base salary
15    } // end six-argument BasePlusCommissionEmployee constructor
16
```

Fig. 9.9 | BasePlusCommissionEmployee inherits protected instance variables from CommissionEmployee. (Part I of 3.)



```
17 // set base salary
18 public void setBaseSalary( double salary )
19 {
20     if ( salary >= 0.0 )
21         baseSalary = salary;
22     else
23         throw new IllegalArgumentException(
24             "Base salary must be >= 0.0" );
25 } // end method setBaseSalary
26
27 // return base salary
28 public double getBaseSalary()
29 {
30     return baseSalary;
31 } // end method getBaseSalary
32
33 // calculate earnings
34 @Override // indicates that this method overrides a superclass method
35 public double earnings()
36 {
37     return baseSalary + ( commissionRate * grossSales );
38 } // end method earnings
```

Fig. 9.9 | BasePlusCommissionEmployee inherits protected instance variables from CommissionEmployee. (Part 2 of 3.)

```
39
40 // return String representation of BasePlusCommissionEmployee
41 @Override // indicates that this method overrides a superclass method
42 public String toString()
43 {
44     return String.format(
45         "%s: %s %s\n%s: %.2f\n%s: %.2f\n%s: %.2f",
46         "base-salaried commission employee", firstName, lastName,
47         "social security number", socialSecurityNumber,
48         "gross sales", grossSales, "commission rate", commissionRate,
49         "base salary", baseSalary );
50 } // end method toString
51 } // end class BasePlusCommissionEmployee
```

Fig. 9.9 | `BasePlusCommissionEmployee` inherits protected instance variables from `CommissionEmployee`. (Part 3 of 3.)

9.4.4 CommissionEmployee–BasePlusCommissionEmployee Inheritance Hierarchy Using protected Instance Variables (Cont.)



- ▶ Inheriting **protected** instance variables slightly increases performance, because we can directly access the variables in the subclass without incurring the overhead of a *set or get method call*.
- ▶ In most cases, it's better to use **private** instance variables to encourage proper software engineering, and leave code optimization issues to the compiler.
 - Code will be easier to maintain, modify and debug.

9.4.4 CommissionEmployee–BasePlusCommissionEmployee Inheritance Hierarchy Using protected Instance Variables (Cont.)



- ▶ Using **protected** instance variables creates several potential problems.
- ▶ The subclass object can set an inherited variable's value directly without using a *set method*.
 - A subclass object can assign an invalid value to the variable
- ▶ Subclass methods are more likely to be written so that they depend on the superclass's data implementation.
 - Subclasses should depend only on the superclass services and not on the superclass data implementation.

9.4.4 CommissionEmployee–BasePlusCommissionEmployee Inheritance Hierarchy Using protected Instance Variables (Cont.)



- ▶ With **protected** instance variables in the superclass, we may need to modify all the subclasses of the superclass if the superclass implementation changes.
 - Such software is said to be **fragile** or **brittle**, because a small change in the superclass can “break” subclass implementation.
 - You should be able to change the superclass implementation while still providing the same services to the subclasses.
 - If the superclass services change, we must reimplement our subclasses.
- ▶ A class’s **protected** members are visible to all classes in the same package as the class containing the **protected** members—this is not always desirable.



Software Engineering Observation 9.4

Use the protected access modifier when a superclass should provide a method only to its subclasses and other classes in the same package, but not to other clients.



Software Engineering Observation 9.5

*Declaring superclass instance variables **private** (as opposed to **protected**) enables the superclass implementation of these instance variables to change without affecting subclass implementations.*



Error-Prevention Tip 9.2

When possible, do not include protected instance variables in a superclass. Instead, include non-private methods that access private instance variables. This will help ensure that objects of the class maintain consistent states.

9.4.5 CommissionEmployee–BasePlusCommissionEmployee Inheritance Hierarchy Using private Instance Variables

- ▶ Hierarchy reengineered using good software engineering practices.
- ▶ Class **CommissionEmployee** declares instance variables **firstName**, **lastName**, **socialSecurityNumber**, **grossSales** and **commissionRate** as **private** and provides **public** methods for manipulating these values.

9.4.5 CommissionEmployee–BasePlusCommissionEmployee Inheritance Hierarchy Using private Instance Variables (Cont.)

- ▶ **CommissionEmployee** methods **earnings** and **toString** use the class's *get* methods to obtain the values of its instance variables.
 - If we decide to change the internal representation of the data (e.g., variable names) only the bodies of the *get and set methods that directly manipulate the instance variables will need to change.*
 - These changes occur solely within the superclass—no changes to the subclass are needed.
 - Localizing the effects of changes like this is a good software engineering practice.
- ▶ Subclass **BasePlusCommissionEmployee** inherits **CommissionEmployee**'s non-private methods and can access the **private** superclass members via those methods.



```
1 // Fig. 9.10: CommissionEmployee.java
2 // CommissionEmployee class uses methods to manipulate its
3 // private instance variables.
4 public class CommissionEmployee
5 {
6     private String firstName;
7     private String lastName;
8     private String socialSecurityNumber;
9     private double grossSales; // gross weekly sales
10    private double commissionRate; // commission percentage
11
12    // five-argument constructor
13    public CommissionEmployee( String first, String last, String ssn,
14        double sales, double rate )
15    {
16        // implicit call to Object constructor occurs here
17        firstName = first;
18        lastName = last;
19        socialSecurityNumber = ssn;
20        setGrossSales( sales ); // validate and store gross sales
21        setCommissionRate( rate ); // validate and store commission rate
22    } // end five-argument CommissionEmployee constructor
```

Fig. 9.10 | CommissionEmployee class uses methods to manipulate its **private** instance variables. (Part I of 6.)



```
23
24     // set first name
25     public void setFirstName( String first )
26     {
27         firstName = first; // should validate
28     } // end method setFirstName
29
30     // return first name
31     public String getFirstName()
32     {
33         return firstName;
34     } // end method getFirstName
35
36     // set last name
37     public void setLastName( String last )
38     {
39         lastName = last; // should validate
40     } // end method setLastName
41
```

Fig. 9.10 | CommissionEmployee class uses methods to manipulate its **private** instance variables. (Part 2 of 6.)



```
42     // return last name
43     public String getLastname()
44     {
45         return lastName;
46     } // end method getLastname
47
48     // set social security number
49     public void setSocialSecurityNumber( String ssn )
50     {
51         socialSecurityNumber = ssn; // should validate
52     } // end method setSocialSecurityNumber
53
54     // return social security number
55     public String getSocialSecurityNumber()
56     {
57         return socialSecurityNumber;
58     } // end method getSocialSecurityNumber
59
```

Fig. 9.10 | CommissionEmployee class uses methods to manipulate its **private** instance variables. (Part 3 of 6.)

```
60 // set gross sales amount
61 public void setGrossSales( double sales )
62 {
63     if ( sales >= 0.0 )
64         grossSales = sales;
65     else
66         throw new IllegalArgumentException(
67             "Gross sales must be >= 0.0" );
68 } // end method setGrossSales
69
70 // return gross sales amount
71 public double getGrossSales()
72 {
73     return grossSales;
74 } // end method getGrossSales
75
```

Fig. 9.10 | CommissionEmployee class uses methods to manipulate its **private** instance variables. (Part 4 of 6.)



```
76 // set commission rate
77 public void setCommissionRate( double rate )
78 {
79     if ( rate > 0.0 && rate < 1.0 )
80         commissionRate = rate;
81     else
82         throw new IllegalArgumentException(
83             "Commission rate must be > 0.0 and < 1.0" );
84 } // end method setCommissionRate
85
86 // return commission rate
87 public double getCommissionRate()
88 {
89     return commissionRate;
90 } // end method getCommissionRate
91
92 // calculate earnings
93 public double earnings()
94 {
95     return getCommissionRate() * getGrossSales();
96 } // end method earnings
97
```

Fig. 9.10 | CommissionEmployee class uses methods to manipulate its **private** instance variables. (Part 5 of 6.)



```
98     // return String representation of CommissionEmployee object
99     @Override // indicates that this method overrides a superclass method
100    public String toString()
101    {
102        return String.format( "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f",
103            "commission employee", getFirstName(), getLastName(),
104            "social security number", getSocialSecurityNumber(),
105            "gross sales", getGrossSales(),
106            "commission rate", getCommissionRate() );
107    } // end method toString
108 } // end class CommissionEmployee
```

Fig. 9.10 | CommissionEmployee class uses methods to manipulate its **private** instance variables. (Part 6 of 6.)



9.4.5 CommissionEmployee–BasePlusCommissionEmployee Inheritance Hierarchy Using private Instance Variables (Cont.)

- ▶ Class **BasePlusCommissionEmployee** (Fig. 9.11) has several changes that distinguish it from Fig. 9.9.
- ▶ Methods **earnings** and **toString** each invoke their superclass versions and do not access instance variables directly.



```
1 // Fig. 9.11: BasePlusCommissionEmployee.java
2 // BasePlusCommissionEmployee class inherits from CommissionEmployee
3 // and accesses the superclass's private data via inherited
4 // public methods.
5
6 public class BasePlusCommissionEmployee extends CommissionEmployee
7 {
8     private double baseSalary; // base salary per week
9
10    // six-argument constructor
11    public BasePlusCommissionEmployee( String first, String last,
12        String ssn, double sales, double rate, double salary )
13    {
14        super( first, last, ssn, sales, rate );
15        setBaseSalary( salary ); // validate and store base salary
16    } // end six-argument BasePlusCommissionEmployee constructor
17
```

Fig. 9.11 | BasePlusCommissionEmployee class inherits from CommissionEmployee and accesses the superclass's `private` data via inherited `public` methods. (Part I of 3.)



```
18 // set base salary
19 public void setBaseSalary( double salary )
20 {
21     if ( salary >= 0.0 )
22         baseSalary = salary;
23     else
24         throw new IllegalArgumentException(
25             "Base salary must be >= 0.0" );
26 } // end method setBaseSalary
27
28 // return base salary
29 public double getBaseSalary()
30 {
31     return baseSalary;
32 } // end method getBaseSalary
33
34 // calculate earnings
35 @Override // indicates that this method overrides a superclass method
36 public double earnings()
37 {
38     return getBaseSalary() + super.earnings();
39 } // end method earnings
```

Fig. 9.11 | BasePlusCommissionEmployee class inherits from CommissionEmployee and accesses the superclass's private data via inherited public methods. (Part 2 of 3.)



```
40 // return String representation of BasePlusCommissionEmployee
41 @Override // indicates that this method overrides a superclass method
42 public String toString()
43 {
44     return String.format( "%s %s\n%s: %.2f", "base-salaried",
45                           super.toString(), "base salary", getBaseSalary() );
46 }
47 } // end method toString
48 } // end class BasePlusCommissionEmployee
```

Fig. 9.11 | BasePlusCommissionEmployee class inherits from CommissionEmployee and accesses the superclass's **private** data via inherited **public** methods. (Part 3 of 3.)

9.4.5 CommissionEmployee–BasePlusCommissionEmployee Inheritance Hierarchy Using private Instance Variables (Cont.)



- ▶ Method **earnings** overrides class the superclass's **earnings** method.
- ▶ The new version calls **CommissionEmployee**'s **earnings** method with **super.earnings()**.
 - Obtains the earnings based on commission alone
- ▶ Placing the keyword **super** and a dot (.) separator before the superclass method name invokes the superclass version of an overridden method.
- ▶ Good software engineering practice
 - If a method performs all or some of the actions needed by another method, call that method rather than duplicate its code.



Common Programming Error 9.3

When a superclass method is overridden in a subclass, the subclass version often calls the superclass version to do a portion of the work. Failure to prefix the superclass method name with the keyword `super` and a dot (.) separator when calling the superclass's method causes the subclass method to call itself, potentially creating an error called infinite recursion. Recursion, used correctly, is a powerful capability discussed in Chapter 18.

9.4.5 CommissionEmployee–BasePlusCommissionEmployee Inheritance Hierarchy Using private Instance Variables (Cont.)



- ▶ `BasePlusCommissionEmployee`'s `toString` method overrides class `CommissionEmployee`'s `toString` method.
- ▶ The new version creates part of the `String` representation by calling `CommissionEmployee`'s `toString` method with the expression `super.toString()`.



9.5 Constructors in Subclasses

- ▶ Instantiating a subclass object begins a chain of constructor calls
 - The subclass constructor, before performing its own tasks, invokes its direct superclass's constructor
- ▶ If the superclass is derived from another class, the superclass constructor invokes the constructor of the next class up the hierarchy, and so on.
- ▶ The last constructor called in the chain is always class **Object**'s constructor.
- ▶ Original subclass constructor's body finishes executing last.
- ▶ Each superclass's constructor manipulates the superclass instance variables that the subclass object inherits.



Software Engineering Observation 9.6

Java ensures that even if a constructor does not assign a value to an instance variable, the variable is still initialized to its default value (e.g., 0 for primitive numeric types, false for booleans, null for references).



9.6 Software Engineering with Inheritance

- ▶ When you extend a class, the new class inherits the superclass's members—though the **private** superclass members are hidden in the new class.
- ▶ You can customize the new class to meet your needs by including additional members and by overriding superclass members.
 - Doing this does not require the subclass programmer to change (or even have access to) the superclass's source code.
 - Java simply requires access to the superclass's `.class` file.



Software Engineering Observation 9.7

Although inheriting from a class does not require access to the class's source code, developers often insist on seeing the source code to understand how the class is implemented. Developers in industry want to ensure that they're extending a solid class—for example, a class that performs well and is implemented robustly and securely.



Software Engineering Observation 9.8

At the design stage in an object-oriented system, you'll often find that certain classes are closely related. You should "factor out" common instance variables and methods and place them in a superclass. Then use inheritance to develop subclasses, specializing them with capabilities beyond those inherited from the superclass.



Software Engineering Observation 9.9

Declaring a subclass does not affect its superclass's source code. Inheritance preserves the integrity of the superclass.



Software Engineering Observation 9.10

Designers of object-oriented systems should avoid class proliferation. Such proliferation creates management problems and can hinder software reusability, because in a huge class library it becomes difficult to locate the most appropriate classes. The alternative is to create fewer classes that provide more substantial functionality, but such classes might prove cumbersome.



9.7 Object Class

- ▶ All classes in Java inherit directly or indirectly from **Object**, so its 11 methods are inherited by all other classes.
- ▶ Figure 9.12 summarizes **Object**'s methods.
- ▶ Every array has an overridden **clone** method that copies the array.
 - If the array stores references to objects, the objects are not copied—a *shallow copy* is performed.



Method	Description
<code>clone</code>	This protected method, which takes no arguments and returns an <code>Object</code> reference, makes a copy of the object on which it's called. The default implementation performs a so-called shallow copy —instance-variable values in one object are copied into another object of the same type. For reference types, only the references are copied. A typical overridden <code>clone</code> method's implementation would perform a deep copy that creates a new object for each reference-type instance variable. Implementing <code>clone</code> correctly is difficult. For this reason, its use is discouraged. Many industry experts suggest that object serialization should be used instead. We discuss object serialization in Chapter 17, Files, Streams and Object Serialization.

Fig. 9.12 | Object methods. (Part 1 of 3.)



Method	Description
<code>equals</code>	This method compares two objects for equality and returns <code>true</code> if they're equal and <code>false</code> otherwise. The method takes any <code>Object</code> as an argument. When objects of a particular class must be compared for equality, the class should override method <code>equals</code> to compare the <i>contents</i> of the two objects. For the requirements of implementing this method, refer to the method's documentation at download.oracle.com/javase/6/docs/api/java/lang/Object.html#equals(java.lang.Object) . The default <code>equals</code> implementation uses operator <code>==</code> to determine whether two references <i>refer to the same object</i> in memory. Section 16.3.3 demonstrates class <code>String</code> 's <code>equals</code> method and differentiates between comparing <code>String</code> objects with <code>==</code> and with <code>equals</code> .
<code>finalize</code>	This <code>protected</code> method (introduced in Section 8.10) is called by the garbage collector to perform termination housekeeping on an object just before the garbage collector reclaims the object's memory. Recall that it's unclear whether, or when, method <code>finalize</code> will be called. For this reason, most programmers should avoid method <code>finalize</code> .

Fig. 9.12 | Object methods. (Part 2 of 3.)



Method	Description
<code>getClass</code>	Every object in Java knows its own type at execution time. Method <code>getClass</code> (used in Sections 10.5, 14.5 and 24.3) returns an object of class <code>Class</code> (package <code>java.lang</code>) that contains information about the object's type, such as its class name (returned by <code>Class</code> method <code>getName</code>).
<code>hashCode</code>	Hashcodes are <code>int</code> values that are useful for high-speed storage and retrieval of information stored in a data structure that's known as a hashtable (discussed in Section 20.11). This method is also called as part of class <code>Object</code> 's default <code>toString</code> method implementation.
<code>wait</code> , <code>notify</code> , <code>notifyAll</code>	Methods <code>notify</code> , <code>notifyAll</code> and the three overloaded versions of <code>wait</code> are related to multithreading, which is discussed in Chapter 26.
<code>toString</code>	This method (introduced in Section 9.4.1) returns a <code>String</code> representation of an object. The default implementation of this method returns the package name and class name of the object's class followed by a hexadecimal representation of the value returned by the object's <code>hashCode</code> method.

Fig. 9.12 | Object methods. (Part 3 of 3.)



9.8 (Optional) GUI and Graphics Case Study: Displaying Text and Images Using Labels

- ▶ **Labels** are a convenient way of identifying GUI components on the screen and keeping the user informed about the current state of the program.
- ▶ A **JLabel** (from package `javax.swing`) can display text, an image or both.
- ▶ The example in Fig. 9.13 demonstrates several **JLabel** features, including a plain text label, an image label and a label with both text and an image.



```
1 // Fig 9.13: LabelDemo.java
2 // Demonstrates the use of Labels.
3 import java.awt.BorderLayout;
4 import javax.swing.ImageIcon;
5 import javax.swing.JLabel;
6 import javax.swing.JFrame;
7
8 public class LabelDemo
9 {
10     public static void main( String[] args )
11     {
12         // Create a Label with plain text
13         JLabel northLabel = new JLabel( "North" );
14
15         // create an icon from an image so we can put it on a JLabel
16         ImageIcon labelIcon = new ImageIcon( "GUITip.gif" );
17
18         // create a Label with an Icon instead of text
19         JLabel centerLabel = new JLabel( labelIcon );
20
21         // create another Label with an Icon
22         JLabel southLabel = new JLabel( labelIcon );
23 }
```

Fig. 9.13 | `JLabel` with text and with images. (Part I of 3.)



```
24     // set the label to display text (as well as an icon)
25     southLabel.setText( "South" );
26
27     // create a frame to hold the labels
28     JFrame application = new JFrame();
29
30     application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
31
32     // add the labels to the frame; the second argument specifies
33     // where on the frame to add the label
34     application.add( northLabel, BorderLayout.NORTH );
35     application.add( centerLabel, BorderLayout.CENTER );
36     application.add( southLabel, BorderLayout.SOUTH );
37
38     application.setSize( 300, 300 ); // set the size of the frame
39     application.setVisible( true ); // show the frame
40 } // end main
41 } // end class LabelDemo
```

Fig. 9.13 | JLabel with text and with images. (Part 2 of 3.)



Fig. 9.13 | JLabel with text and with images. (Part 3 of 3.)



9.8 (Optional) GUI and Graphics Case Study: Displaying Text and Images Using Labels (Cont.)

- ▶ An **ImageIcon** represents an image that can be displayed on a **JLabel**.
- ▶ The constructor for **ImageIcon** receives a **String** that specifies the path to the image.
- ▶ **ImageIcon** can load images in GIF, JPEG and PNG image formats.
- ▶ **JLabel** method **setText** changes the text the label displays.



9.8 (Optional) GUI and Graphics Case Study: Displaying Text and Images Using Labels (Cont.)

- ▶ An overloaded version of method `add` that takes two parameters allows you to specify the GUI component to add to a `JFrame` and the location in which to add it.
 - The first parameter is the component to attach.
 - The second is the region in which it should be placed.
- ▶ Each `JFrame` has a `layout` to position GUI components.
 - Default layout for a `JFrame` is `BorderLayout`.
 - Five regions—`NORTH` (top), `SOUTH` (bottom), `EAST` (right side), `WEST` (left side) and `CENTER` (constants in class `BorderLayout`)
 - Each region is declared as a constant in class `BorderLayout`.
- ▶ When calling method `add` with one argument, the `JFrame` places the component in the `BorderLayout`'s `CENTER` automatically.

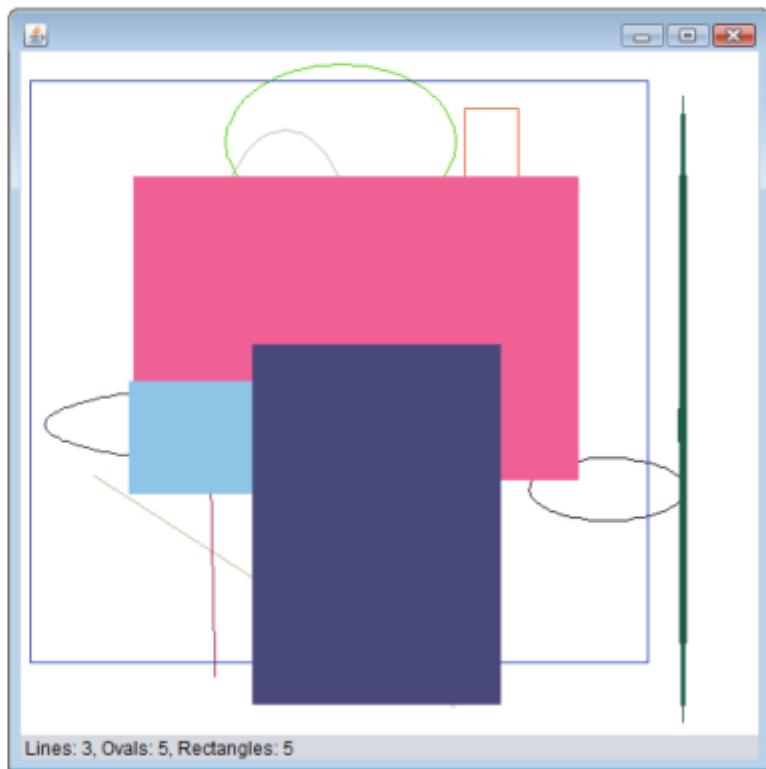


Fig. 9.14 | JLabel displaying shape statistics.



Packages

Java™ How to Program, 9/e



© Copyright 1992–2012 by Pearson
Education, Inc. All Rights Reserved.



Creating Packages

- ▶ Each class in the Java API belongs to a package that contains a group of related classes.
- ▶ Packages are defined once, but can be imported into many programs.
- ▶ Packages help programmers manage the complexity of application components.
- ▶ Packages facilitate software reuse by enabling programs to import classes from other packages, rather than copying the classes into each program that uses them.
- ▶ Packages provide a convention for unique class names, which helps prevent class-name conflicts.



Creating Packages (Cont.)

- ▶ The steps for creating a reusable class:
- ▶ Declare a `public` class; otherwise, it can be used only by other classes in the same package.
- ▶ Choose a unique package name and add a **package declaration** to the source-code file for the reusable class declaration.
 - In each Java source-code file there can be only one **package** declaration, and it must precede all other declarations and statements.
- ▶ Compile the class so that it's placed in the appropriate package directory.
- ▶ Import the reusable class into a program and use the class.



Creating Packages (Cont.)

- ▶ Placing a **package** declaration at the beginning of a Java source file indicates that the class declared in the file is part of the specified package.
- ▶ Only **package** declarations, **import** declarations and comments can appear outside the braces of a class declaration.
- ▶ A Java source-code file must have the following order:
 - a **package** declaration (if any),
 - **import** declarations (if any), then
 - class declarations.
- ▶ Only one of the class declarations in a particular file can be **public**.
- ▶ Other classes in the file are placed in the package and can be used only by the other classes in the package.
- ▶ Non-**public** classes are in a package to support the reusable classes in the package.



```
1 // Fig. 8.15: Time1.java
2 // Time1 class declaration maintains the time in 24-hour format.
3 package com.deitel.jhttp.ch08;
4
5 public class Time1
6 {
7     private int hour; // 0 - 23
8     private int minute; // 0 - 59
9     private int second; // 0 - 59
10
11    // set a new time value using universal time; throw an
12    // exception if the hour, minute or second is invalid
13    public void setTime( int h, int m, int s )
14    {
15        // validate hour, minute and second
16        if ( ( h >= 0 && h < 24 ) && ( m >= 0 && m < 60 ) &&
17            ( s >= 0 && s < 60 ) )
18        {
19            hour = h;
20            minute = m;
21            second = s;
22        } // end if
```

Fig. 8.15 | Packaging class Time1 for reuse. (Part 1 of 2.)



```
23     else
24         throw new IllegalArgumentException(
25             "hour, minute and/or second was out of range" );
26 } // end method setTime
27
28 // convert to String in universal-time format (HH:MM:SS)
29 public String toUniversalString()
30 {
31     return String.format( "%02d:%02d:%02d", hour, minute, second );
32 } // end method toUniversalString
33
34 // convert to String in standard-time format (H:MM:SS AM or PM)
35 public String toString()
36 {
37     return String.format( "%d:%02d:%02d %s",
38         ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ),
39         minute, second, ( hour < 12 ? "AM" : "PM" ) );
40 } // end method toString
41 } // end class Time1
```

Fig. 8.15 | Packaging class Time1 for reuse. (Part 2 of 2.)



Time Class Case Study: Creating Packages (Cont.)

- ▶ Every package name should start with your Internet domain name in reverse order.
 - For example, our domain name is `deitel.com`, so our package names begin with `com.deitel`.
 - For the domain name *yourcollege.edu*, the package name should begin with `edu.yourcollege`.
- ▶ After the domain name is reversed, you can choose any other names you want for your package.
 - We chose to use `jhtp` as the next name in our package name to indicate that this class is from *Java How to Program*.
 - The last name in our package name specifies that this package is for Chapter 8 (`ch08`).



Time Class Case Study: Creating Packages (Cont.)

- ▶ Compile the class so that it's stored in the appropriate package.
- ▶ When a Java file containing a **package** declaration is compiled, the resulting class file is placed in the directory specified by the declaration.
- ▶ The **package** declaration

```
package com.deitel.jhttp.ch08;
```
- ▶ indicates that class **Time1** should be placed in the directory

```
com  
  deitel  
    jhttp  
      ch08
```
- ▶ The directory names in the **package** declaration specify the exact location of the classes in the package.



Time Class Case Study: Creating Packages (Cont.)

- ▶ **javac** command-line option **-d** causes the **javac** compiler to create appropriate directories based on the class's **package** declaration.
 - The option also specifies where the directories should be stored.
- ▶ Example:

```
javac -d . Time1.java
```
- ▶ specifies that the first directory in our package name should be placed in the current directory (**.**).
- ▶ The compiled classes are placed into the directory that is named last in the **package** statement.



Time Class Case Study: Creating Packages (Cont.)

- ▶ The **package** name is part of the **fully qualified class name**.
 - Class `Time1`'s name is actually
`com.deitel.jhttp.ch08.Time1`
- ▶ Can use the fully qualified name in programs, or **import** the class and use its **simple name** (the class name by itself).
- ▶ If another package contains a class by the same name, the fully qualified class names can be used to distinguish between the classes in the program and prevent a **name conflict** (also called a **name collision**).



Time Class Case Study: Creating Packages (Cont.)

- ▶ Fig. 8.15, line 3 is a **single-type-import declaration**
 - It specifies one class to import.
- ▶ When your program uses multiple classes from the same package, you can import those classes with a **type-import-on-demand declaration**.
- ▶ Example:

```
import java.util.*; // import java.util classes
```
- ▶ uses an asterisk (*) at the end of the **import** declaration to inform the compiler that all **public** classes from the **java.util** package are available for use in the program.
 - Only the classes from package **java-.util** that are used in the program are loaded by the JVM.



```
1 // Fig. 8.16: Time1PackageTest.java
2 // Time1 object used in an application.
3 import com.deitel.jhttp.ch08.Time1; // import class Time1
4
5 public class Time1PackageTest
6 {
7     public static void main( String[] args )
8     {
9         // create and initialize a Time1 object
10        Time1 time = new Time1(); // invokes Time1 constructor
11
12        // output string representations of the time
13        System.out.print( "The initial universal time is: " );
14        System.out.println( time.toUniversalString() );
15        System.out.print( "The initial standard time is: " );
16        System.out.println( time.toString() );
17        System.out.println(); // output a blank line
18
```

Fig. 8.16 | Time1 object used in an application. (Part 1 of 3.)



```
19     // change time and output updated time
20     time.setTime( 13, 27, 6 );
21     System.out.print( "Universal time after setTime is: " );
22     System.out.println( time.toUniversalString() );
23     System.out.print( "Standard time after setTime is: " );
24     System.out.println( time.toString() );
25     System.out.println(); // output a blank line
26
27     // attempt to set time with invalid values
28     try
29     {
30         time.setTime( 99, 99, 99 ); // all values out of range
31     } // end try
32     catch ( IllegalArgumentException e )
33     {
34         System.out.printf( "Exception: %s\n\n", e.getMessage() );
35     } // end catch
36
```

Fig. 8.16 | Time1 object used in an application. (Part 2 of 3.)



```
37     // display time after attempt to set invalid values
38     System.out.println( "After attempting invalid settings:" );
39     System.out.print( "Universal time: " );
40     System.out.println( time.toUniversalString() );
41     System.out.print( "Standard time: " );
42     System.out.println( time.toString() );
43 } // end main
44 } // end class Time1PackageTest
```

```
The initial universal time is: 00:00:00
The initial standard time is: 12:00:00 AM

Universal time after setTime is: 13:27:06
Standard time after setTime is: 1:27:06 PM

Exception: hour, minute and/or second was out of range

After attempting invalid settings:
Universal time: 13:27:06
Standard time: 1:27:06 PM
```

Fig. 8.16 | Time1 object used in an application. (Part 3 of 3.)



Common Programming Error 8.11

Using the `import` declaration `import java.;` causes a compilation error. You must specify the exact name of the package from which you want to import classes.*



Time Class Case Study: Creating Packages (Cont.)

- ▶ Specifying the Classpath During **Compilation**
- ▶ When compiling a class that uses classes from other packages, `javac` must locate the `.class` files for all other classes being used.
- ▶ The compiler uses a special object called a **class loader** to locate the classes it needs.
 - The class loader begins by searching the standard Java classes that are bundled with the JDK.
 - Then it searches for **optional packages**.
 - If the class is not found in the standard Java classes or in the extension classes, the class loader searches the **classpath**, which contains a list of locations in which classes are stored.



Time Class Case Study: Creating Packages (Cont.)

- ▶ The classpath consists of a list of directories or **archive files**, each separated by a **directory separator**
 - Semicolon (;) on Windows or a colon (:) on UNIX/Linux/Mac OS X.
- ▶ Archive files are individual files that contain directories of other files, typically in a compressed format.
 - Archive files normally end with the .jar or .zip file-name extensions. (**discussed later**)
- ▶ The directories and archive files specified in the classpath contain the classes you wish to make available to the Java compiler and the JVM.



Time Class Case Study: Creating Packages (Cont.)

- ▶ By default, the classpath consists only of the current directory.
- ▶ The classpath can be modified by
 - providing the **-classpath** option to the **javac** compiler
 - setting the **CLASSPATH** environment variable (not recommended).
- ▶ Classpath
 - download.oracle.com/javase/6/docs/technotes/tools/index.html#genera
 - The section entitled “General Information” contains information on setting the classpath for UNIX/Linux and Windows.



Common Programming Error 8.12

Specifying an explicit classpath eliminates the current directory from the classpath. This prevents classes in the current directory (including packages in the current directory) from loading properly. If classes must be loaded from the current directory, include a dot (.) in the classpath to specify the current directory.



Software Engineering Observation 8.11

In general, it's a better practice to use the `-classpath` option of the compiler, rather than the `CLASSPATH` environment variable, to specify the classpath for a program. This enables each application to have its own classpath.



Error-Prevention Tip 8.5

Specifying the classpath with the CLASSPATH environment variable can cause subtle and difficult-to-locate errors in programs that use different versions of the same package.



Creating Packages (Cont.)

- ▶ Specifying the Classpath When **Executing** an Application
- ▶ When you execute an application, the JVM must be able to locate the `.class` files of the classes used in that application.
- ▶ Like the compiler, the `java` command uses a class loader that searches the standard classes and extension classes first, then searches the classpath (the current directory by default).
- ▶ The classpath can be specified explicitly by using either of the techniques discussed for the compiler.
- ▶ As with the compiler, it's better to specify an individual program's classpath via command-line JVM options.
 - If classes must be loaded from the current directory, be sure to include a dot (`.`) in the classpath to specify the current directory.



Package Access

- ▶ If no access modifier is specified for a method or variable when it's declared in a class, the method or variable is considered to have **package access**.
- ▶ In a program uses multiple classes from the same package, these classes can access each other's package-access members directly through references to objects of the appropriate classes, or in the case of **static** members through the class name.
- ▶ Package access is rarely used.



```
1 // Fig. 8.17: PackageDataTest.java
2 // Package-access members of a class are accessible by other classes
3 // in the same package.
4
5 public class PackageDataTest
6 {
7     public static void main( String[] args )
8     {
9         PackageData packageData = new PackageData();
10
11     // output String representation of packageData
12     System.out.printf( "After instantiation:\n%s\n", packageData );
13
14     // change package access data in packageData object
15     packageData.number = 77;
16     packageData.string = "Goodbye";
17
18     // output String representation of packageData
19     System.out.printf( "\nAfter changing values:\n%s\n", packageData );
20 } // end main
21 } // end class PackageDataTest
22
```

Fig. 8.17 | Package-access members of a class are accessible by other classes in the same package. (Part 1 of 3.)



```
23 // class with package access instance variables
24 class PackageData
25 {
26     int number; // package-access instance variable
27     String string; // package-access instance variable
28
29     // constructor
30     public PackageData()
31     {
32         number = 0;
33         string = "Hello";
34     } // end PackageData constructor
35
36     // return PackageData object String representation
37     public String toString()
38     {
39         return String.format("number: %d; string: %s", number, string );
40     } // end method toString
41 } // end class PackageData
```

Fig. 8.17 | Package-access members of a class are accessible by other classes in the same package. (Part 2 of 3.)



After instantiation:
number: 0; string: Hello

After changing values:
number: 77; string: Goodbye

Fig. 8.17 | Package-access members of a class are accessible by other classes in the same package. (Part 3 of 3.)

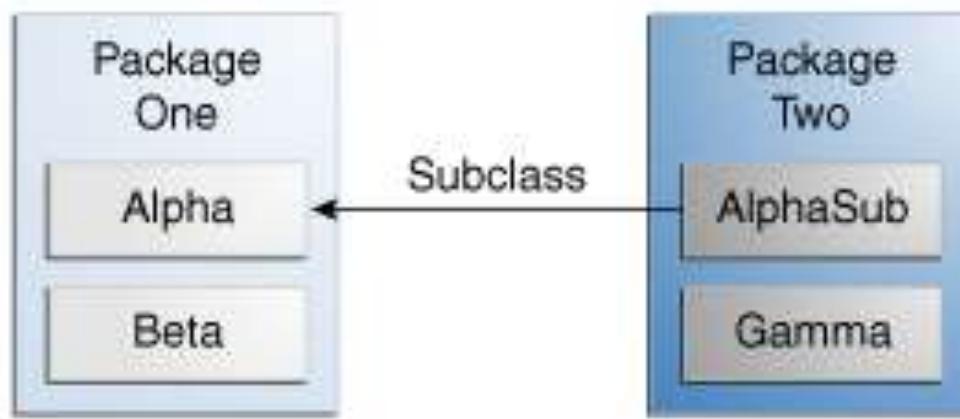


Access Modifiers in Java

- ▶ There are four types of access modifiers available in java:
 1. Default – No keyword required
 2. Private
 3. Protected
 4. Public



	default	private	protected	public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes



The following table shows where the members of the Alpha class are visible for each of the access modifiers that can be applied to them.

Modifier	Alpha	Beta	Alphasub	Gamma
public	Y	Y	Y	Y
protected	Y	Y	Y	N
<i>no modifier</i>	Y	Y	N	N
private	Y	N	N	N