Object-Oriented Problem Solving

Course Outline

Course Goals

Upon completion of this course, you should be able to:

- Implement object-oriented programs and understand underlying principles such as object composition, encapsulation, abstraction, reuse…
- Perform required analysis to solve a problem using an object-oriented approach and implement the most appropriate design for the problem.
- To do that we will use Java.

Textbook

- **Y. Daniel Liang, "Introduction to Java Programming", 10th edition, Prentice Hall.**
- Horstmann & Cornell, "Core Java, Volume 1 – Fundamentals", Sun Microsystems Press.
- B. Eckel, "Thinking in Java" http://mindview.net/Books/TIJ/

Personnel

- Instructor: Eng. Asma Abdel Karim.
- Email: a.abdelkarim@ju.edu.jo
- Website: www.asmaabdelkarim.com
- Office hours:
  - Sun, Tue: 10:30 - 11:30
  - Wed: 12:00 – 1:00
  - By appointment.
- Lectures: Sun, Tue, Thu: 12:30 – 1:30

  Mon, Wed: 8:30 - 10:00

Grading

- Assignments 10%
- Quizzes 10%
- Midterm Exam 20%
- Final Exam 25%
- Lab 35%
  - In-lab 10%
  - Mid Practical Exam 10%
  - Final Practical Exam 15%

Academic Honesty

- You are allowed to discuss assignments with other students in the class.
  - Getting verbal advice/help from people who have already taken the course is also fine.
- However, any sharing of code is <u>not acceptable</u>.
  - Under no circumstances may you hand in work done with or by someone else under your own name.

Course Outline

- Introduction
- Programming Fundamentals
- Methods
- Arrays
- Objects and Classes
- Inheritance and Polymorphism
- Abstract Classes and Interfaces
- Generics
- Exception Handling and Text I/O
- Introduction to JavaFX

Object-Oriented Problem Solving

Introduction

Based on Chapter 1 of “Introduction to Java Programming”
by Y. Daniel Liang.

Outline

- What is Programming (1.1)
- Programming Languages (1.3)
- Operating Systems (1.4)
- Procedural vs. object oriented programming.
- What is Java? (1.5)
- The Java Language Specification, API, JDK, and IDE (1.6)
- A simple Java program (1.7)
- Creating, Compiling, and Executing a Java Program. (1.8)
- Programming Errors (1.10)

What is Programming?

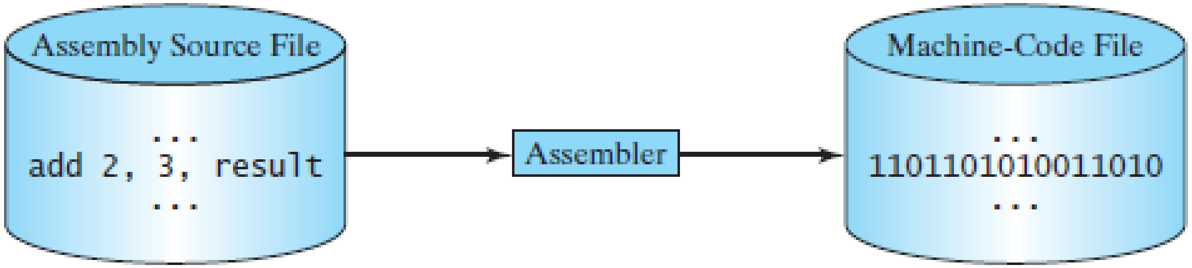
- Programming means to create software.
  - Software contains the instructions that tell a computer what to do.
- Software developers create software with the help of powerful tools called *programming languages*.

Evolution of Programming Languages Machine Language

- Computer’s native language.
- Differs among different types of computers.
- Set of built-in primitive instructions in the form of binary code.
- For example to add two numbers, you might have to write something like this:
  - 1110110001100110
- Difficult to read and modify.

Evolution of Programming Languages
Assembly Language

- Were created as an alternative to machine languages.
- Uses short descriptive words.
- One-to-one correspondence between each assembly instruction and machine instruction.
- An *assembler* is used to translate assembly-language programs into machine code.

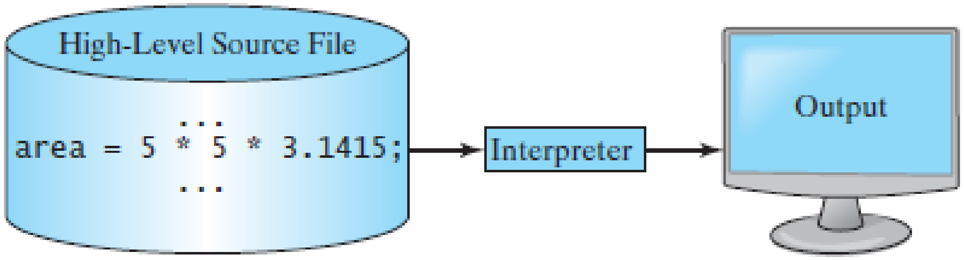


Evolution of Programming Languages High-Level Language

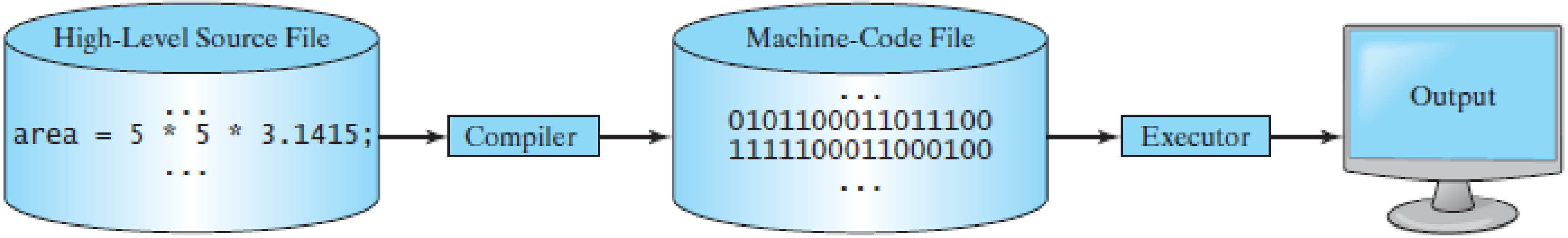
- Platform independent.
  - You can write a program in a high-level language and run it on different types of machines.
- Easy to learn and use.
- Examples of high level languages:
  - BASIC, C/C++, C#, COBOL, FORTRAN, Java, Python.
- An *interpreter* or a *compiler* is used to translate the *source code* into machine code.
  - *What is the difference between interpreters and compilers?*

Evolution of Programming Languages High-Level Language (Cont.)

- Interpreters read one statement from the source code, translate it to machine code and execute it right away.

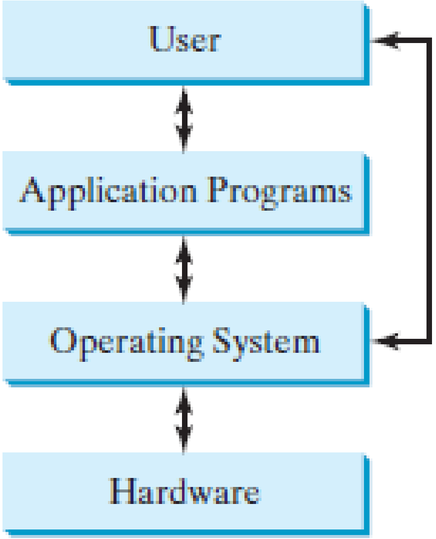


- Compilers translate the entire source code into a machine code, and the machine file is then executed.



Operating Systems

- The operating system is the most important program that runs on the computer.
- Popular operating systems for general-purpose computers include:
  - Microsoft Windows, Mac OS, and Linux.
- Major tasks of an operating system:
  - Controlling and monitoring system activities.
  - Allocating and assigning system resources.
  - Scheduling operations.



Procedural vs. Object Oriented Programming

- Procedural programming consists of designing a set of modules (*functions* or *methods*) to solve a problem.
  - – Focuses on how to manipulate the data, then on what data structures to use to make the manipulation easier.
- Object oriented programming puts data first, then look at algorithms that operate on the data.
  - – Focuses on objects and operations on objects.

What is Java?

- Was developed by a team led by *James Gosling* at Sun Microsystems.
- Powerful and versatile programming language for developing software running on:
  - Mobile devices.
  - Desktop computers.
  - Servers.

The Java Language Specification

- Programming languages have strict rules of usage.
- The *Java Language Specification* is a technical definition of the Java programming language syntax and semantic.
  - https://docs.oracle.com/javase/specs/

Java API, JDK, and IDE

- The *Application Program Interface* (*API*):
  - Also known as library.
  - Contains predefined classes and interfaces for developing Java programs.
- The *Java Development Kit* (*JDK*) consists of a set of separate programs, each invoked from command line, for developing and testing Java programs.
- The *Integrated Development Environment* (*IDE*) provides graphical user interface to edit, compile, build, and debug programs.

Java Editions

- Java *Standard Edition* (Java SE) to develop client-side standalone applications or applets.
  - The foundation upon which all other Java technology is based.
  - There are many versions of Java SE. The latest is Java SE 8.
- Java *Enterprise Edition* (Java EE) to develop server-side applications.
- Java *Micro Edition* (Java ME) to develop applications for mobile devices.

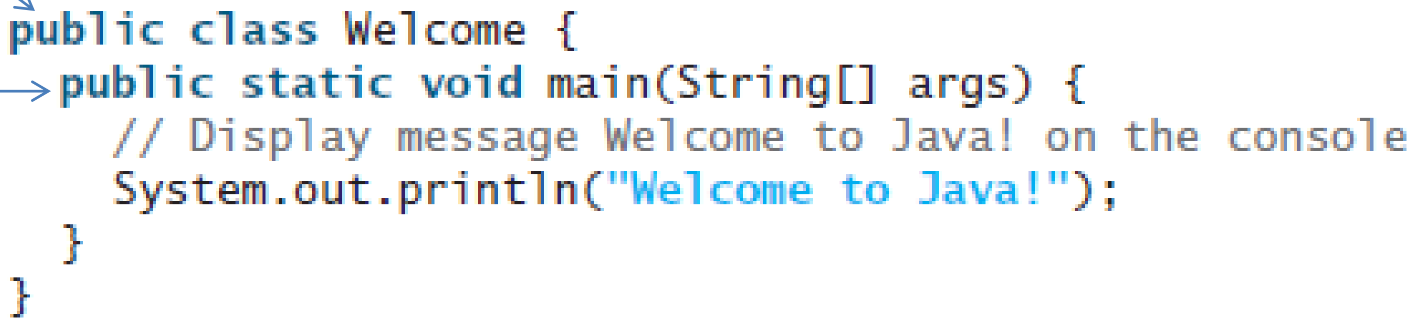
A Simple Java Program

- A Java program is executed from the *main method in the class*.
- We will begin with a simple Java program that displays the message **Welcome to Java!** on the *console*.
- The word *console* is an old computer term that refers to the text entry and display device of a computer.
  - Console input means to receive input from the keyboard.
  - Console output means to display output on the monitor

A Simple Java Program

Class definition

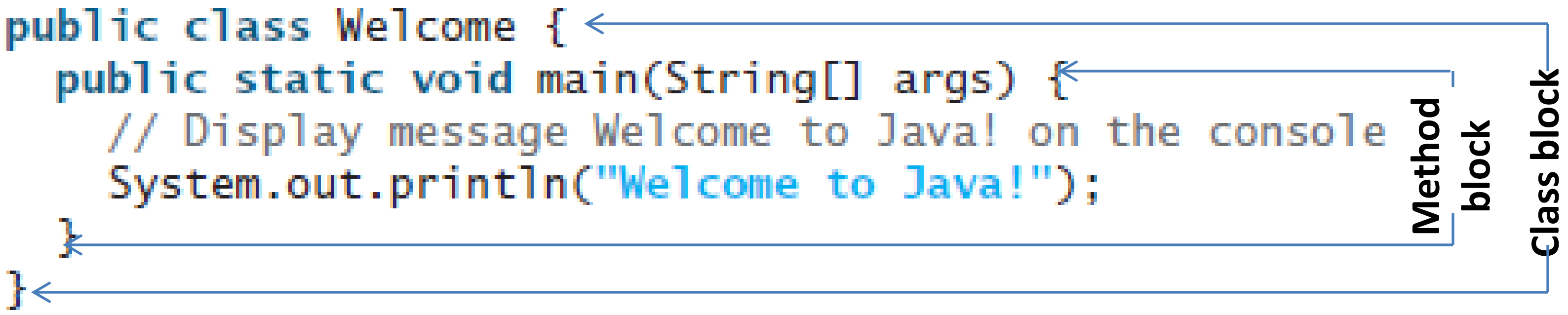
Main method definition



```
public class Welcome {
  public static void main(String[] args) {
    // Display message Welcome to Java! on the console
    System.out.println("Welcome to Java!");
  }
}
```

- Java source programs are <u>case sensitive</u>.
- Every Java program must have *at least* one *class*.
- Each class has a name. By convention, class names start with an uppercase letter.
- The *main* method is the entry point where the program begins execution.
- A class may contain several methods. A method is a construct that contains statements.

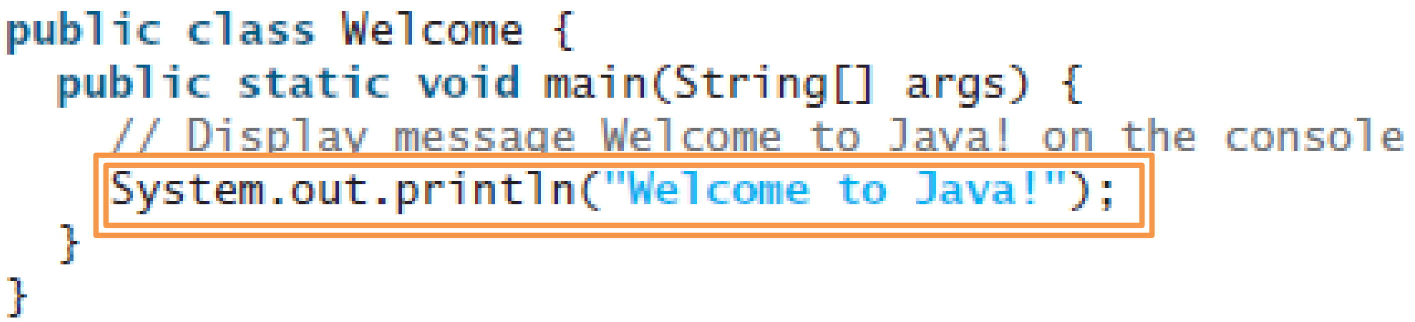
A Simple Java Program (Cont.)



```
public class Welcome {
  public static void main(String[] args) {
    // Display message Welcome to Java! on the console
    System.out.println("Welcome to Java!");
  }
}
```

- A pair of curly braces in a program forms a *block* that groups the program's component.
- Every class has a *class block* that groups the data and methods of the class.
- Every method has a *method block* that groups the statements in the method.

A Simple Java Program (Cont.)

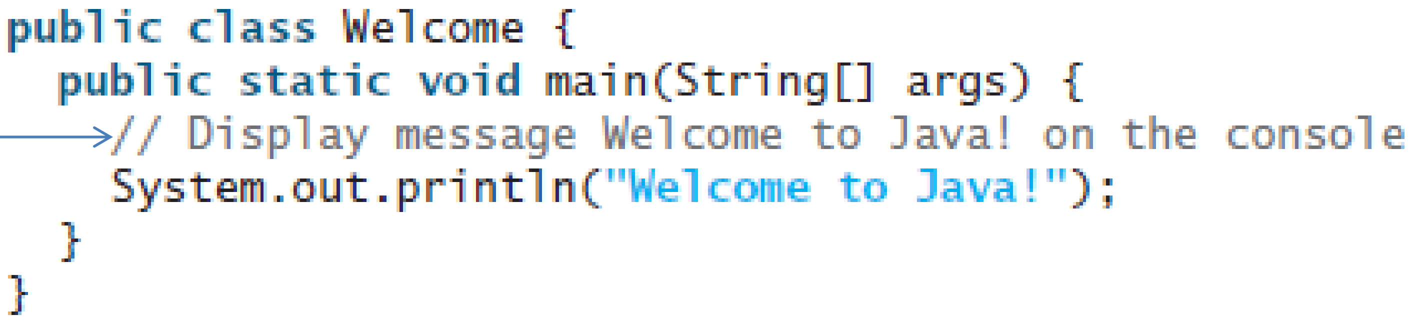


```
public class Welcome {
  public static void main(String[] args) {
    // Display message Welcome to Java! on the console
    System.out.println("Welcome to Java!");
  }
}
```

- The *System.out.println* statement displays the string *Welcome to Java* on the console.
- A *String* is a sequence of characters. Strings should be enclosed in double quotation marks.
- Every statement in Java ends with a semicolon (*;*).
- *public*, *class*, *static*, and *void* are reserved words : have a specific meaning to the compiler and cannot be used for other purposes. In the program.

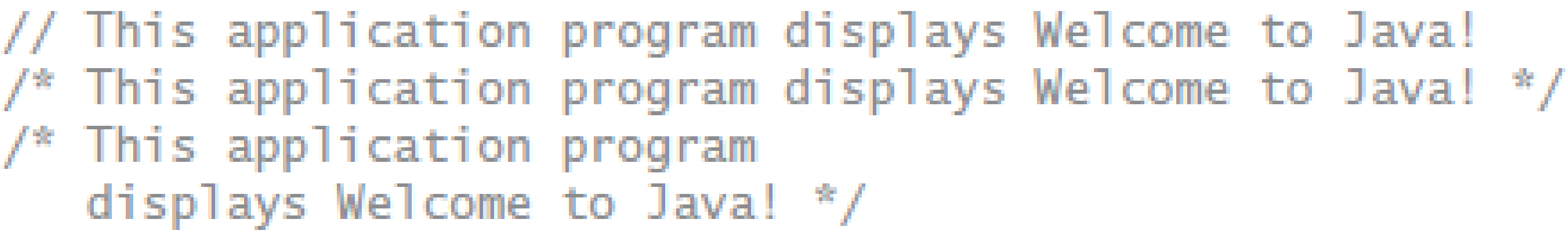
A Simple Java Program (Cont.)

Comment



```
public class Welcome {
  public static void main(String[] args) {
    // Display message Welcome to Java! on the console
    System.out.println("Welcome to Java!");
  }
}
```

- Comments are ignored by the compiler.
- Two types of comments:
  - Line comments: preceded by two slashes (//).
  - Block (or paragraph) comments: enclosed between (/*) and (*/)



```
// This application program displays Welcome to Java!
/* This application program displays Welcome to Java! */
/* This application program
   displays Welcome to Java! */
```

Java Special Characters

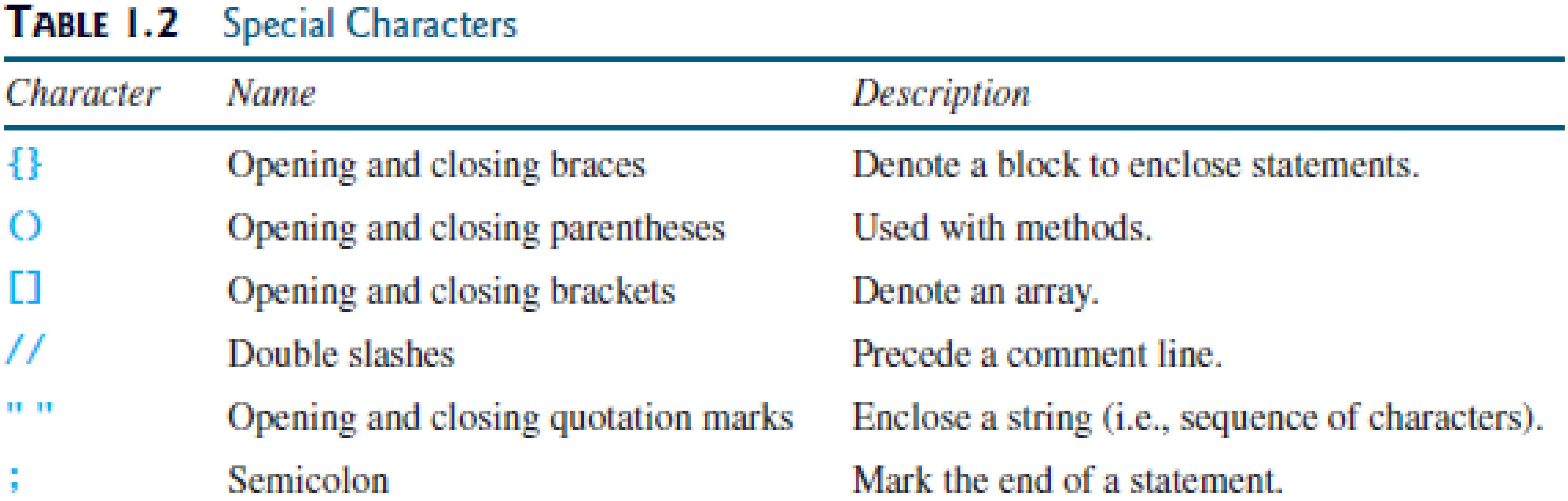
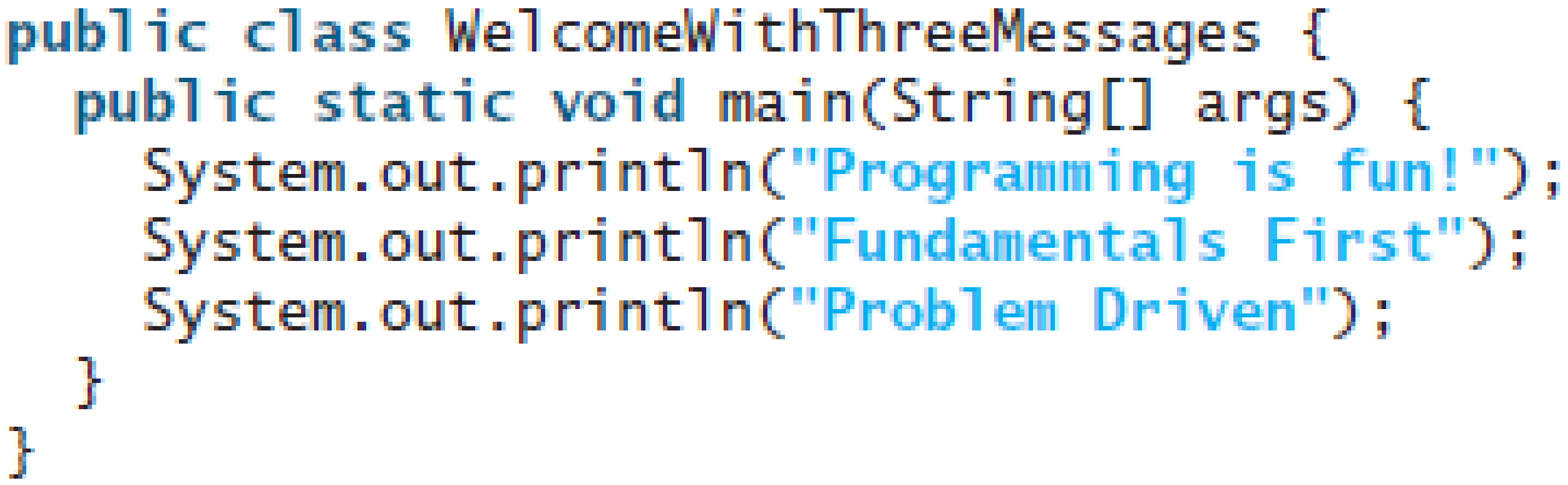


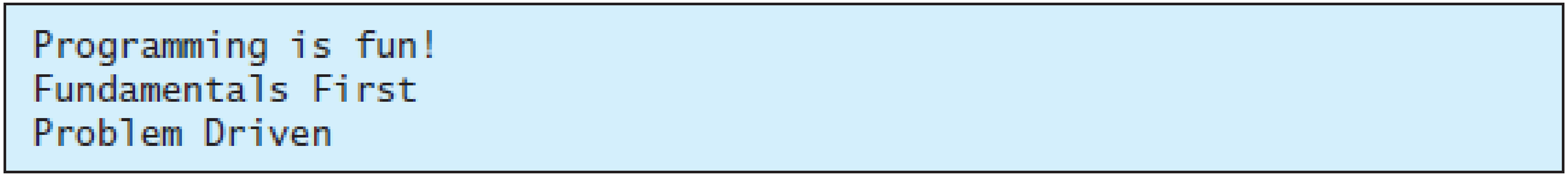
TABLE 1.2 Special Characters

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

Displaying More Messages to the Console

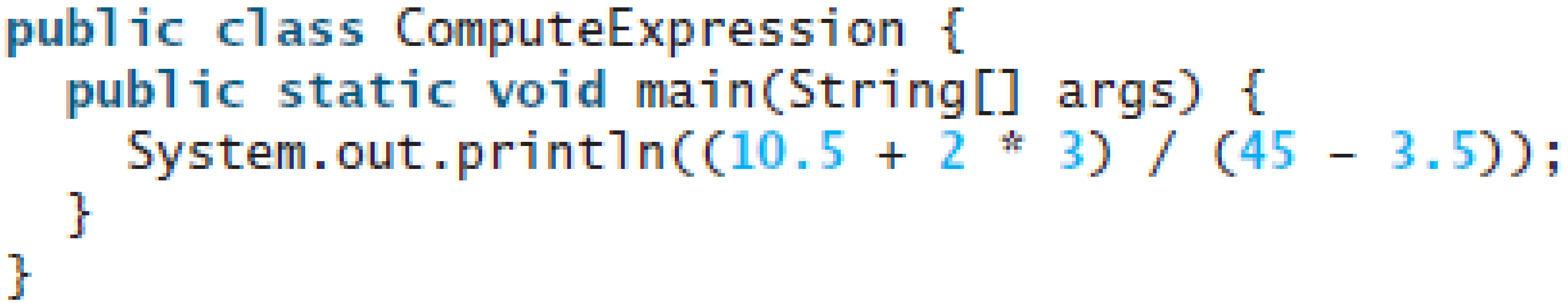


```java
public class WelcomeWithThreeMessages {
  public static void main(String[] args) {
    System.out.println("Programming is fun!");
    System.out.println("Fundamentals First");
    System.out.println("Problem Driven");
  }
}
```

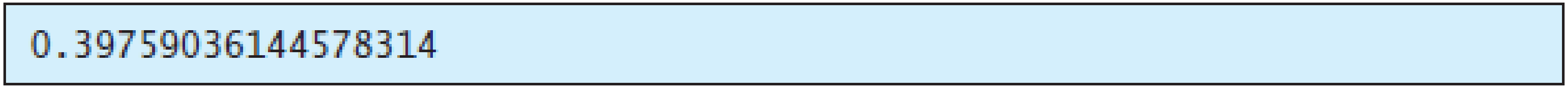


```
Programming is fun!
Fundamentals First
Problem Driven
```

Displaying the Result of a Mathematical Computation

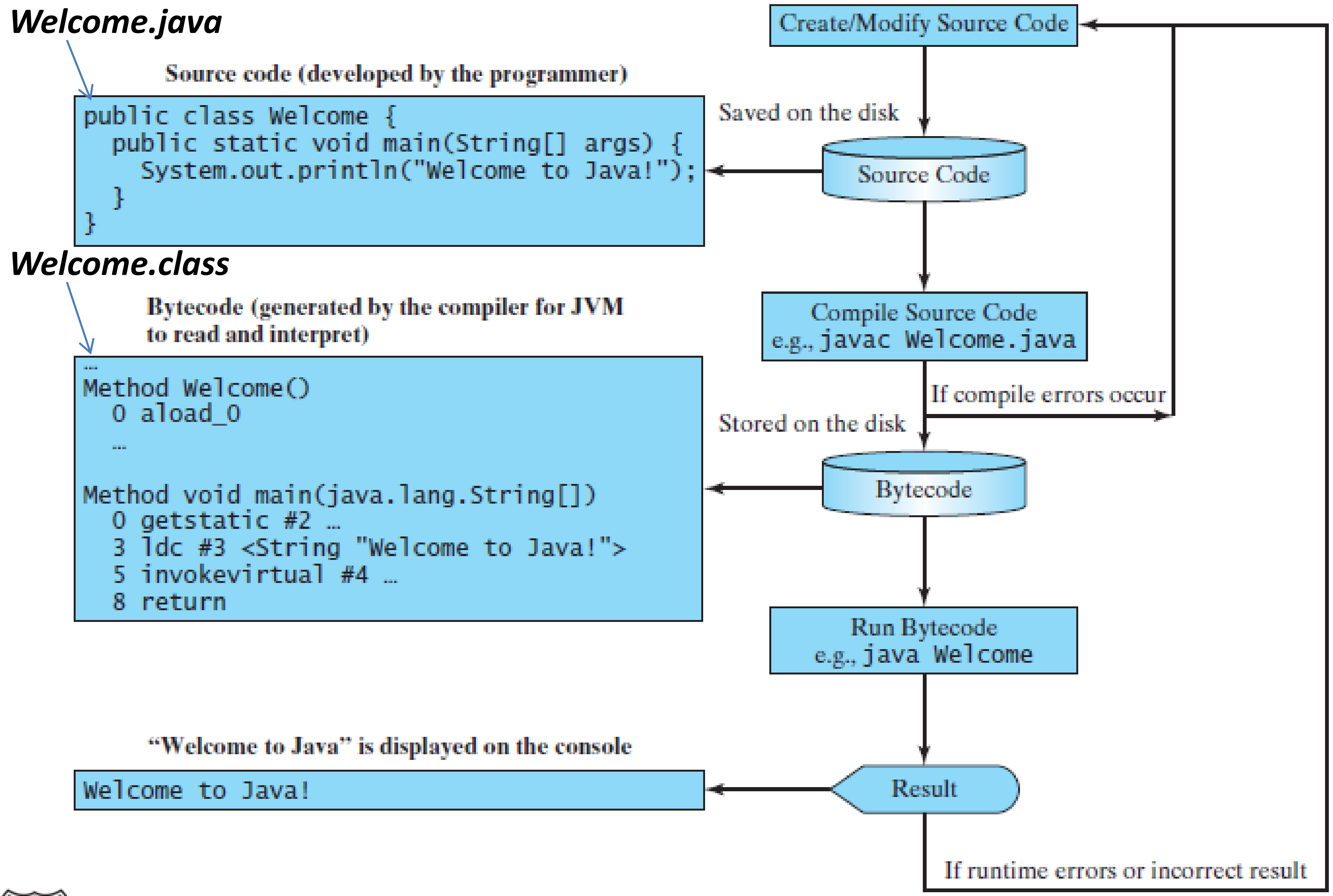


```
public class ComputeExpression {
  public static void main(String[] args) {
    System.out.println((10.5 + 2 * 3) / (45 - 3.5));
  }
}
```



```
0.39759036144578314
```

Creating, Compiling, and Executing a Java Program



Welcome.java

Source code (developed by the programmer)

```
public class Welcome {
  public static void main(String[] args) {
    System.out.println("Welcome to Java!");
  }
}
```

Welcome.class

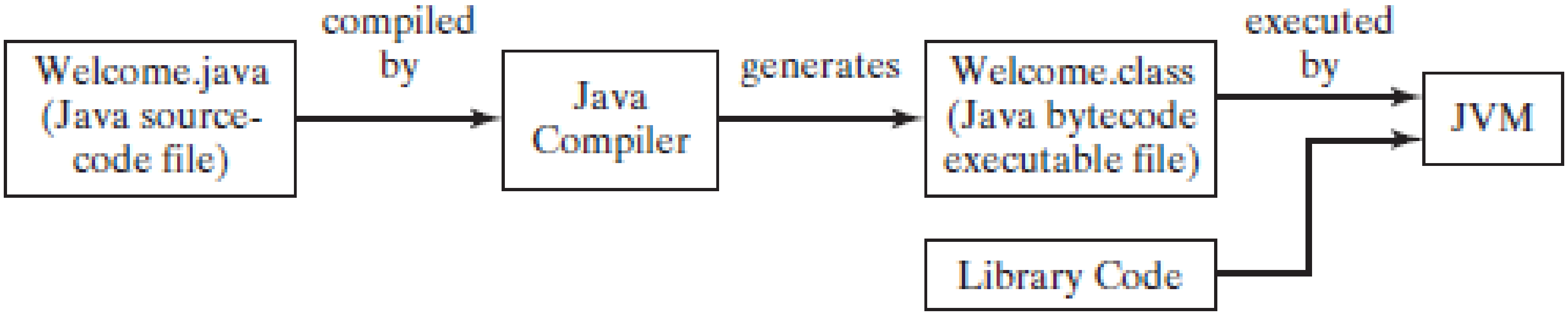
Bytecode (generated by the compiler for JVM to read and interpret)

```
…
Method Welcome()
  0 aload_0
  …

Method void main(java.lang.String[])
  0 getstatic #2 …
  3 ldc #3 <String "Welcome to Java!">
  5 invokevirtual #4 …
  8 return
```

"Welcome to Java" is displayed on the console

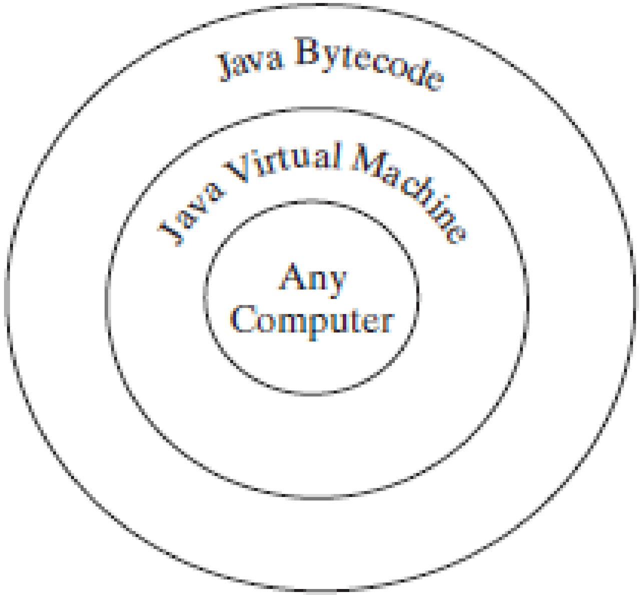
```
Welcome to Java!
```

Creating, Compiling, and Executing a Java Program (Cont.)


Welcome.java (Java source-code file)
compiled by
Java Compiler
generates
Welcome.class (Java bytecode executable file)
executed by
JVM
Library Code

- Java source code is compiled into Java *bytecode*.
- Your Java code may use the code in the Java library.
- The JVM is an *interpreter*, which translates individual instructions in the *bytecode* into the target machine language code and executes it immediately.

Creating, Compiling, and Executing a Java Program (Cont.)



- The *bytecode* is similar to machine instructions, but is *architecture neutral* and can run on any platform that has a *Java Virtual Machine* (*JVM*).

Creating, Compiling, and Executing a Java Program (Cont.)

- When executing a Java program, the JVM first loads the bytecode of the class to memory using a program called the *class loader*.
  - If your program uses other classes, the class loader dynamically loads them just before they are needed.
- After a class is loaded, the JVM uses a program called the *bytecode verifier* to check the validity of the bytecode and to ensure that the bytecode does not violate Java's security restrictions.
  - Java enforces strict security to make sure that Java class files are not tampered with and do not harm your computer.

Programming Errors

- Syntax errors.
  - Detected by the compiler.
  - Result from errors in code construction.
- Runtime errors.
  - Cause a program to terminate abnormally.
  - Occur while the program is running if the environment detects an operation that is impossible to carry out.
  - Examples include input errors and division by zero.
- Logic errors.
  - Occur when a program does not perform the way it is intended to.

Object-Oriented Problem Solving

Programming Fundamentals (Part I)

Based on sections from chapters 2, 3 & 4 of
"Introduction to Java Programming" by Y. Daniel Liang.

Eng. Asma Abdel Karim
Computer Engineering Department

Outline

- Identifiers (2.4)
- Variables (2.5)
- Assignment statement (2.6)
- Named Constants (2.7)
- Naming Conventions (2.8)
- Numeric Data Types and Operations (2.9)
- Numeric Literals (2.10)
- Evaluating Expressions and Operator Precedence (2.11)
- Augmented Assignment Operators (2.13)
- Increment and Decrement Operator (2.14)
- Numeric Type Conversion (2.15)
- Boolean Data Type (3.2)
- Character Data Type and Operations (4.3)
- The String Type (4.4)

Identifiers

- Identifiers are the names of things that appear in the program.
  - Names of variables, constants, methods, classes, packages…
- All identifiers must obey the following rules:
  - An identifier is a sequence of characters that consists of letters, digits, underscores (_), and dollar sign ($).
  - Cannot start with a digit.
  - Cannot be a reserved word.
  - Cannot be true, false, or null.
  - Can be of any length.
- Examples of legal identifiers: $2, area, Area, S_3.
- Examples of illegal identifiers: 2A, d+4, S#6.

Variables

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

  datatype variableName
- Examples of variable declarations:
  - *int count;*
  - *double rate;*
  - *char letter;*
  - *boolean found;*

Variables (Cont.)

- Several variables can be declared together:
  - *int count, limit, numberOfStudents;*
- When a variable is declared, the compiler allocates memory space for the variable based on its data type.

Assignment Statement

- An assignment statement designates a value for a variable.
- The *equal sign (=)* is used as the assignment operator.
- Examples:
  - *x = 1;*
  - *x = x+1;*
  - *area = radius * radius * 3.14159;*

Assignment Statement (Cont.)

- Variables can be declared and initialized in one step:
  - *int count = 0;*
  - *char letter = 'a';*
  - *boolean found = false;*
  - *int i = 1, j = 2;*
- *int count = 0;* is equivalent to the following two statements:
  - *int count;*
  - *count = 0;*

Assignment Statement (Cont.)

- An assignment statement can be used as an expression in Java:
  - *System.out.println(x=1);*
- A value can be assigned to multiple variables:
  - *i = j = k = 1;*
- In an assignment statement the data type of the variable on the left must be compatible with the data type of the value on the right.
  - Except if *type casting* is used.

Named Constants

- A *named constant* is an identifier that represents a permanent value.
- A *constant* must be declared and initialized in the same statement.
- The syntax for declaring a constant:
  – *final datatype CONSTANT_NAME = value;*
- Example:
  – *final double PI = 3.14159;*

Named Constants (Cont.)

- There are three benefits of using constants:

(1) You don’t have to repeatedly type the same value if it is used multiple times.

(2) If you have to change the constant value (e.g., from **3.14** to **3.14159** for **PI**), you need to change it only in a single location in the source code; and

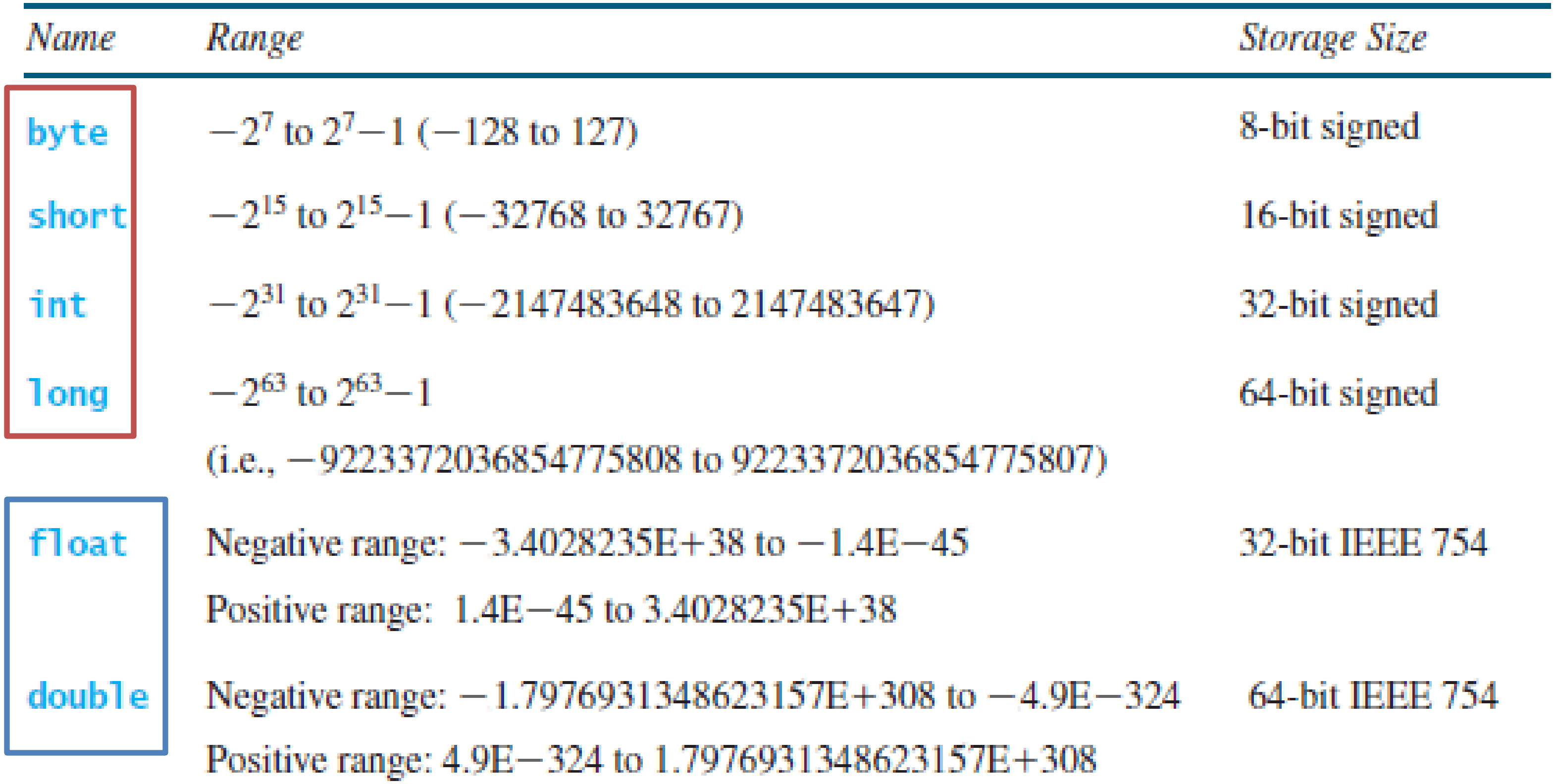
(3) A descriptive name for a constant makes the program easy to read.

Naming Conventions

- Sticking with the Java naming conventions makes your programs easy to read and avoids errors.
- Make sure that you choose descriptive names with straightforward meanings for the variables, constants, classes, and methods in your program.
- Use lowercase for variables and methods.
  - E.g. radius, count.
  - If a name consists of several words, concatenate them, make the first word lowercase and capitalize the first letter of each subsequent word.
    - E.g. numberOfStudents.
- Capitalize the first letter of each word in a class name.
  - E.g. ComputeAread, String.
- Capitalize every letter in a constant, and use underscores between words.
  - PI, MAX_VALUE.

Numeric Data Types

- Java has six built-in numeric data types.

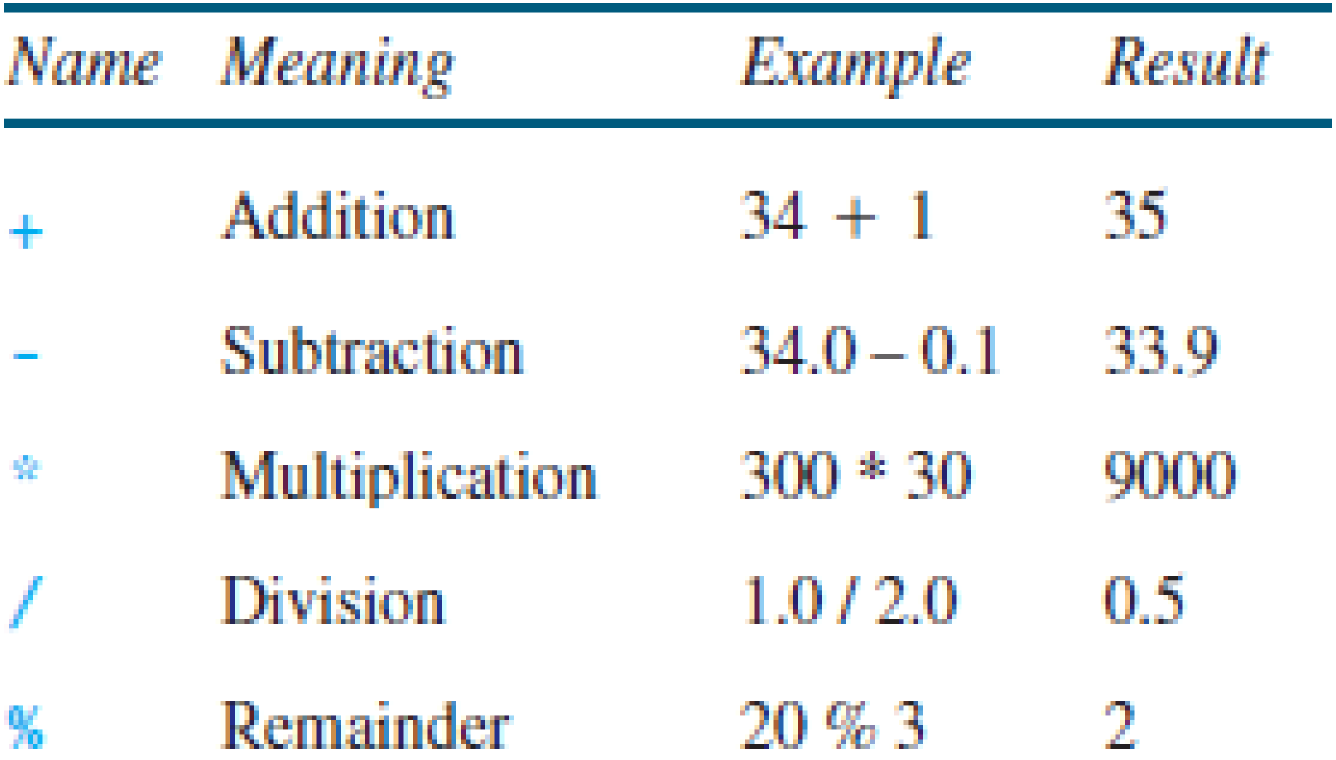


Name	*Range*	*Storage Size*
`byte`	-2^7 to 2^7-1 (−128 to 127)	8-bit signed
`short`	-2^{15} to $2^{15}-1$ (−32768 to 32767)	16-bit signed
`int`	-2^{31} to $2^{31}-1$ (−2147483648 to 2147483647)	32-bit signed
`long`	-2^{63} to $2^{63}-1$ (i.e., −9223372036854775808 to 9223372036854775807)	64-bit signed
`float`	Negative range: −3.4028235E+38 to −1.4E−45 Positive range: 1.4E−45 to 3.4028235E+38	32-bit IEEE 754
`double`	Negative range: −1.7976931348623157E+308 to −4.9E−324 Positive range: 4.9E−324 to 1.7976931348623157E+308	64-bit IEEE 754

Integers

Floating Point Numbers

Numeric Operators



Name	*Meaning*	*Example*	*Result*
+	Addition	34 + 1	35
-	Subtraction	34.0 – 0.1	33.9
*	Multiplication	300 * 30	9000
/	Division	1.0 / 2.0	0.5
%	Remainder	20 % 3	2

Numeric Literals: Integrals

- A literal is a constant value that appears directly in a program.
  - int numberOfYears = *34*;
  - double weight = *0.305*;
- An integer literal can be assigned to an integer variable as long as it can fit into the variable.
  - Otherwise a compile error occurs.
  - E.g. *byte b = 128;* will cause a compilation error.
- To denote an integer literal of the long type, append letter L or l to it.
  - E.g. *2147483648L*

Numeric Literals: Floating Points

- Floating point literals are written with a decimal point.
- By default, a floating point literal is treated as a double type value.
  - *5.0* is considered a double value.
- You can make a number a float by appending the letter f or F.
  - E.g. *100.2F*

Numeric Literals: Floating Points (Cont.)

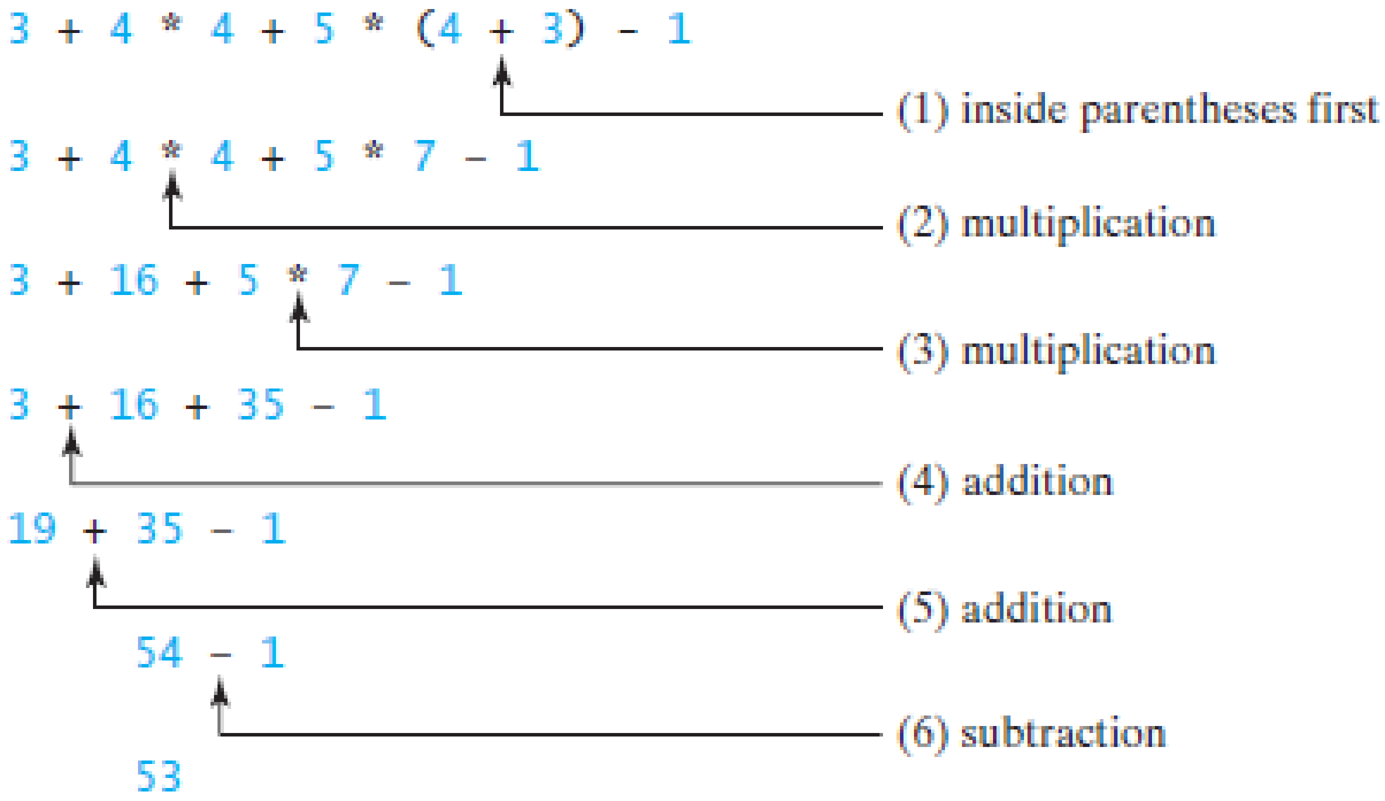
- Double type values are more accurate than the float type values.
  - *System.out.println(“1.0 / 3.0 is “ + 1.0 / 3.0);*

  Displays *1.0 / 3.0 is 0.3333333333333333*

  - *System.out.println(“1.0 / 3.0 is “ + 1.0F / 3.0F);*

  Displays *1.0 / 3.0 is 0.33333334*

Evaluating Expressions and Operator Precedence



```
3 + 4 * 4 + 5 * (4 + 3) - 1
```

(1) inside parentheses first

```
3 + 4 * 4 + 5 * 7 - 1
```

(2) multiplication

```
3 + 16 + 5 * 7 - 1
```

(3) multiplication

```
3 + 16 + 35 - 1
```

(4) addition

```
19 + 35 - 1
```

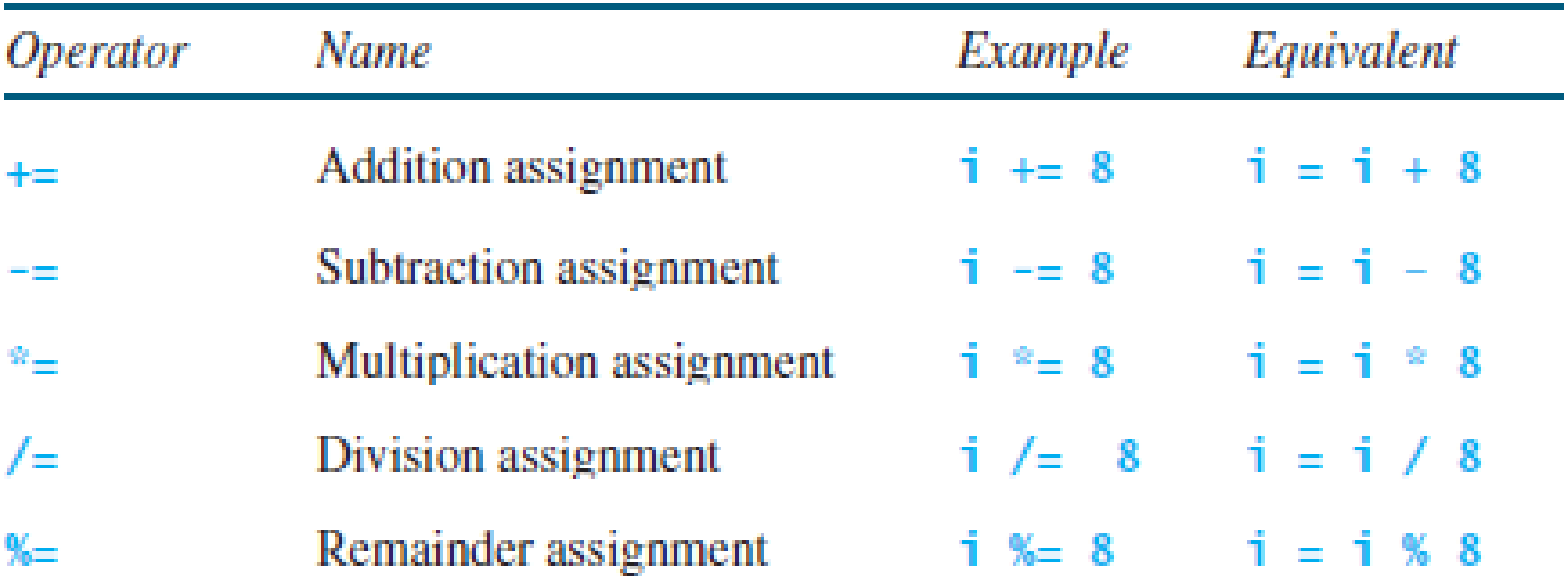
(5) addition

```
    54 - 1
```

(6) subtraction

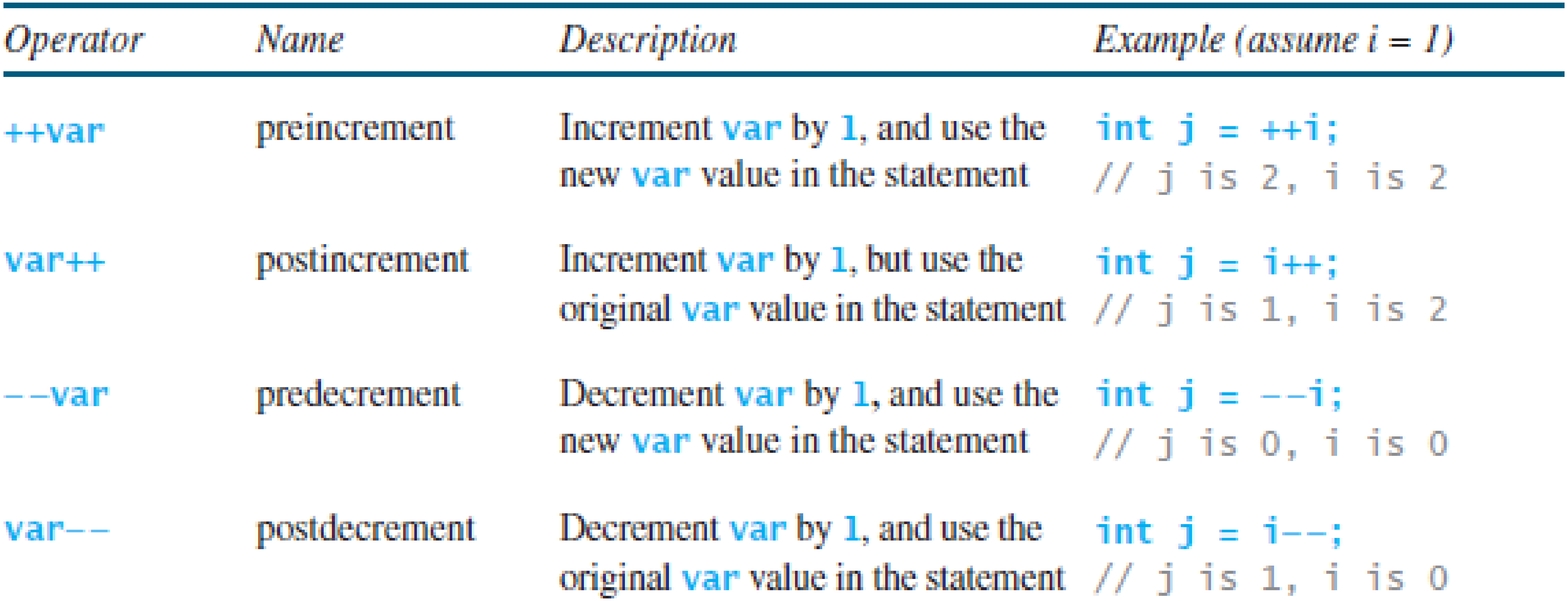
```
    53
```

Augmented Assignment Operators



Operator	*Name*	*Example*	*Equivalent*
`+=`	Addition assignment	`i += 8`	`i = i + 8`
`-=`	Subtraction assignment	`i -= 8`	`i = i - 8`
`*=`	Multiplication assignment	`i *= 8`	`i = i * 8`
`/=`	Division assignment	`i /=  8`	`i = i / 8`
`%=`	Remainder assignment	`i %= 8`	`i = i % 8`

Increment and Decrement Operators



Operator	*Name*	*Description*	*Example (assume i = 1)*

`++var`	preincrement	Increment `var` by `1`, and use the new `var` value in the statement	`int j = ++i;`
			`// j is 2, i is 2`

`var++`	postincrement	Increment `var` by `1`, but use the original `var` value in the statement	`int j = i++;`
			`// j is 1, i is 2`

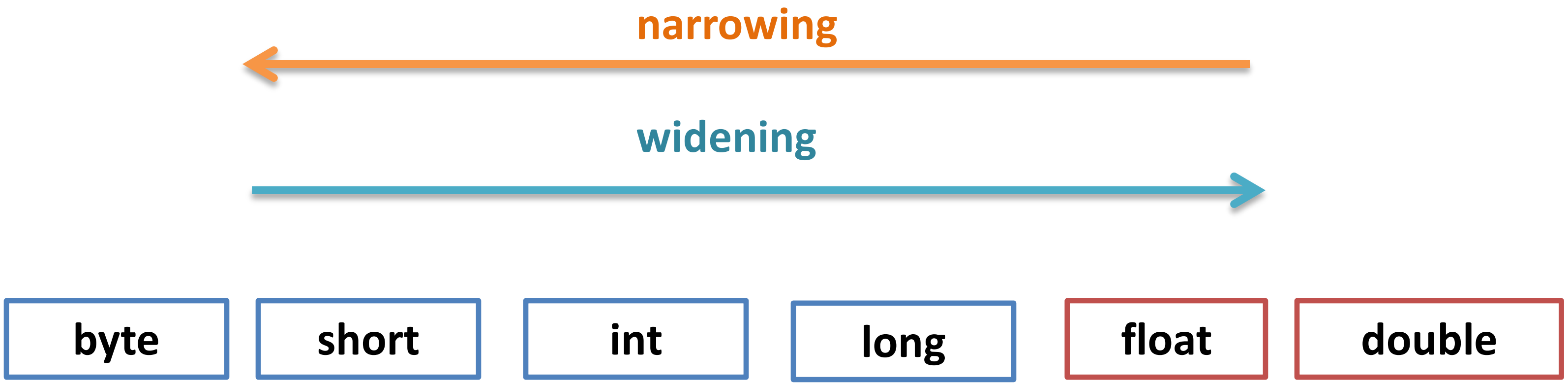
`--var`	predecrement	Decrement `var` by `1`, and use the new `var` value in the statement	`int j = --i;`
			`// j is 0, i is 0`

`var--`	postdecrement	Decrement `var` by `1`, and use the original `var` value in the statement	`int j = i--;`
			`// j is 1, i is 0`

Numeric Type Conversions

- You can always assign a value to a numeric variable whose type supports a larger range of values.
  - You can assign a **long** value to a **float** variable.
- You cannot, however, assign a value to a variable of a type with a smaller range unless you use *type casting*.
- *Casting* is an operation that converts a value of one data type into a value of another data type.
  - *Widening a type* is casting a type with a small range to a type with a larger range.
    - E.g. Integer to floating point: *3 * 4.5* is same as *3.0 * 4.5*.
  - *Narrowing a type* is casting a type with a large range to a type with a smaller range.
    - E.g. floating point to integer: *System.out.println ((int)1.7);*
- Java automatically widens a type, but you must narrow a type explicitly.

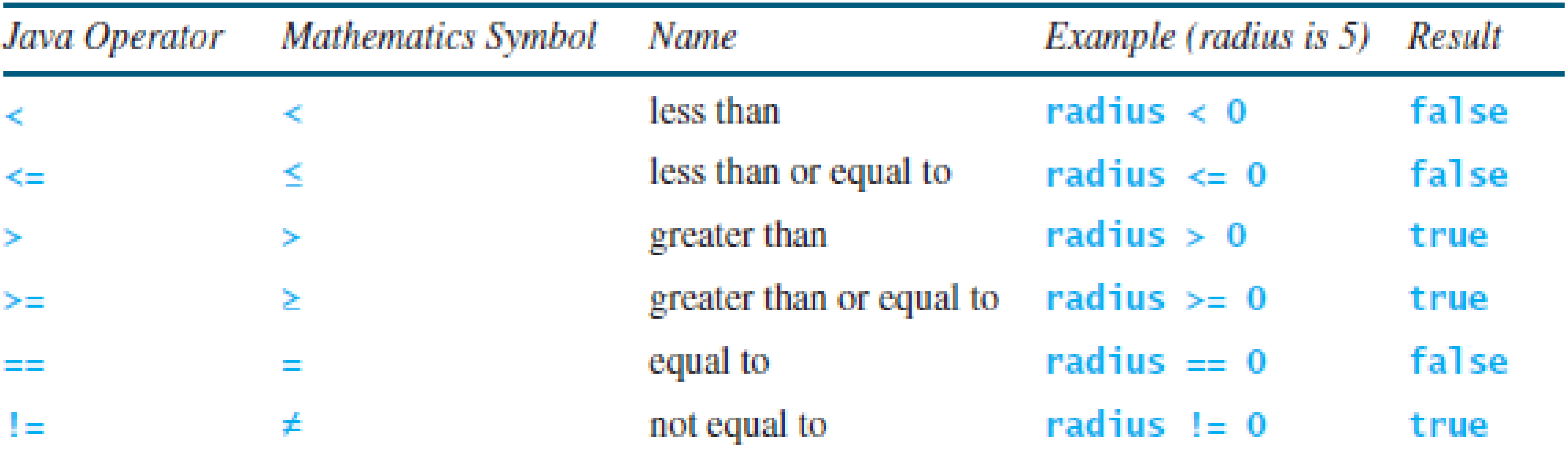
Numeric Type Conversions



boolean Data Type

- A *boolean* data type declares a variable with the value *true* or *false*.
- Boolean expressions represent conditions that are used to make decisions in the program.

Comparison Operators



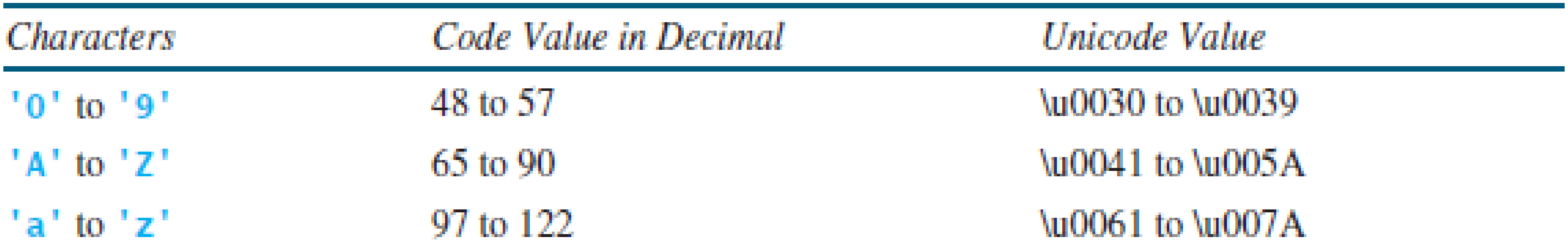
Java Operator	*Mathematics Symbol*	*Name*	*Example (radius is 5)*	*Result*
`<`	<	less than	`radius < 0`	`false`
`<=`	≤	less than or equal to	`radius <= 0`	`false`
`>`	>	greater than	`radius > 0`	`true`
`>=`	≥	greater than or equal to	`radius >= 0`	`true`
`==`	=	equal to	`radius == 0`	`false`
`!=`	≠	not equal to	`radius != 0`	`true`

- The result of a comparison is a boolean value: *true* or *false*.
- Note that the equality comparison operator is two equal signs (==), not a single equal sign (=); this is the assignment operator

The Character Data Type

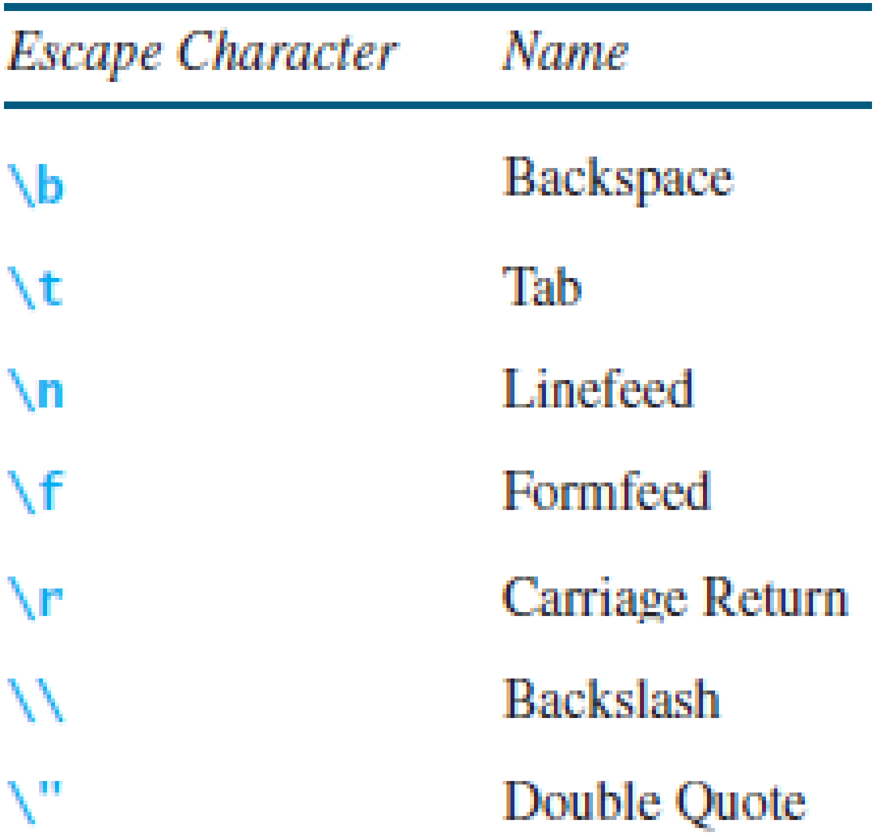
- The *character* data type represents a single character.
- A character literal is enclosed in single quotation marks.
- Examples:
  - *char letter = 'A';*
  - *char numChar = '4';*
- Java uses *Unicode* which is designed as a 16-bit character encoding.

Unicode for Commonly Used Characters



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

Character Literals



Escape Character	*Name*
`\b`	Backspace
`\t`	Tab
`\n`	Linefeed
`\f`	Formfeed
`\r`	Carriage Return
`\\`	Backslash
`\"`	Double Quote

Casting between *char* and Numeric Types

- A *char* can be cast into any numeric type, and vice versa.
- When an integer is cast into a **char**, only its lower 16 bits of data are used; the other part is ignored.
  - *char ch = (char)0XAB0041;*
  - *System.out.println(ch);   // ch is character A*
- When a floating-point value is cast into a *char*, the floating-point value is first cast into an *int*, which is then cast into a *char*.
  - *char ch = (char)65.25;*
  - *System.out.println(ch);   // ch is character A*

Casting between *char* and Numeric Types (Cont.)

- When a *char* is cast into a numeric type, the character’s Unicode is cast into the specified numeric type.
  - *int i = (int)'A';*
  - *System.out.println(i);   // i is 65*
- Implicit casting can be used if the result of a casting fits into the target variable. Otherwise, explicit casting must be used.
  - *byte b = 'a';*
  - *int i = 'a';*
- Any positive integer between **0** and **FFFF** in hexadecimal can be cast into a character implicitly. Any number not in this range must be cast into a *char* explicitly.

The String Type

- A string is a sequence of characters.
- To represent a string of characters, use the data type called *String*:
  - E.g. *String message = “Welcome to Java”;*
- *String* is a predefined class in the Java library.
- The String type is not a *primitive* type. It is known as a *reference* type.
- A string literal must be enclosed on quotation marks (“ ”).

The String Type (Cont.)

- The *plus sign* (*+*) is the *concatenation* operator if at least one of the operands is a string.
  - If one of the operands is a non string (e.g. a number), the non string value is converted into a string and concatenated with the string.
  - Examples:
    - *String message = “Welcome ” + “to “ + ”Java!”;*
      message becomes: Welcome to Java!
    - *String s = “Chapter” + 2;*
      s becomes: Chapter2
    - *String appendix = “Appendix” + ‘B’;*
      appendix becomes: AppendixB
- If neither of the operands is a string, the *plus sign* (*+*) is the *addition* operator.

Object-Oriented Problem Solving

Programming Fundamentals (Part II)

Based on sections from chapters 3 & 5 of “Introduction to Java Programming” by Y. Daniel Liang.

Eng. Asma Abdel Karim
Computer Engineering Department

Outline

- Selections
  - One-way if statements (3.3)
  - Two-way if-else statements (3.4)
  - Nested If and Multi-Way if-else Statements (3.5)
  - Logical Operators (3.10)
  - Switch statements (3.13)
  - Conditional expressions (3.14)
- Loops
  - While loops (5.2)
  - The do-while loops (5.3)
  - For loops (5.4)
  - Nested Loops (5.6)
  - Keywords break and continue (5.9)

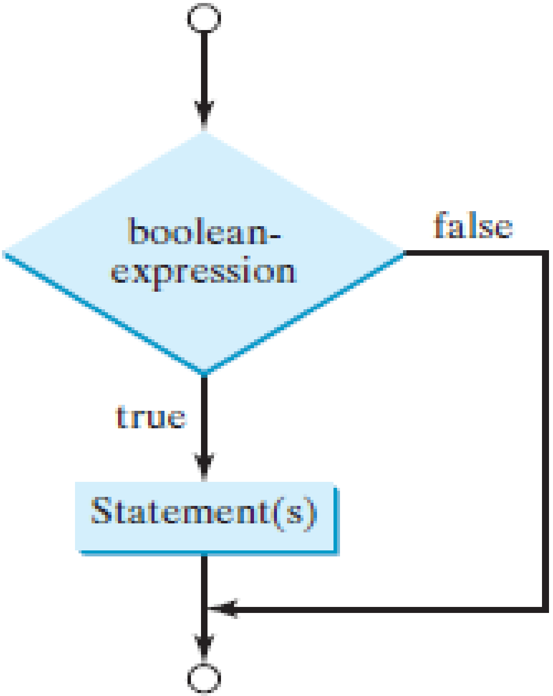
Selections

- The program can decide which statements to execute based on a condition.
- Selection statements use conditions that are *Boolean expressions*.
    - A *Boolean expression* is an expression that evaluates to a Boolean value: *true* or *false*.

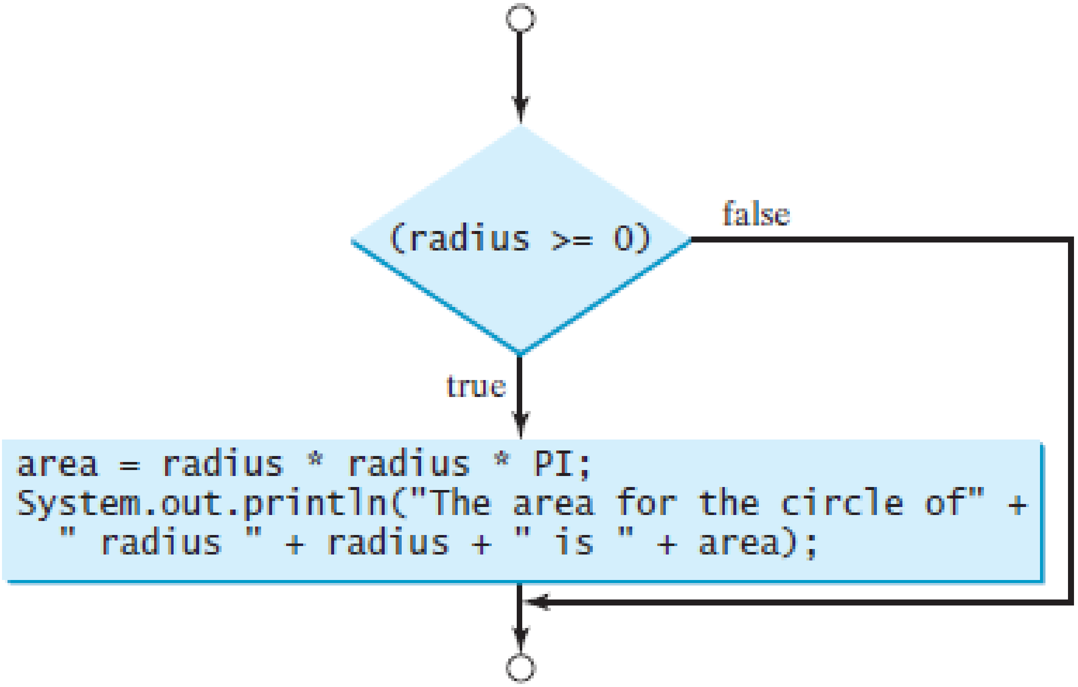
One-way If Statements

- An *if* statement is a construct that enables a program to specify alternative paths of execution.
- A *one-way if* statement executes an action *if an only if* the condition is *true*.
  - If the condition is *false*, nothing is done.
- The syntax for a one-way if statement is:

```
if (boolean-expression){
        statement(s);
}
```



One-way If Statements (Example)



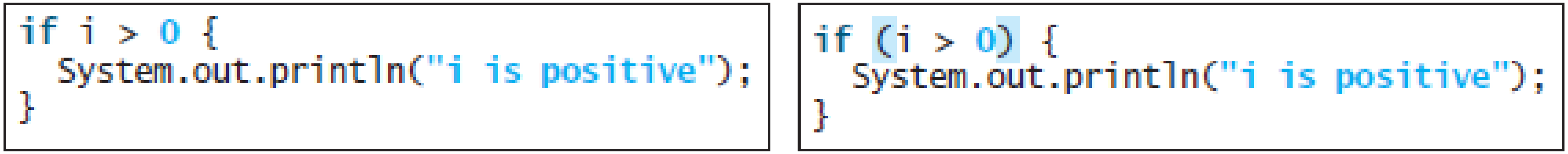
(radius >= 0)

```
area = radius * radius * PI;
System.out.println("The area for the circle of" +
  " radius " + radius + " is " + area);
```

```
if (radius >= 0) {
  area = radius * radius * PI;
  System.out.println("The area for the circle of radius " +
    radius + " is " + area);
}
```

One-way If Statements (Cont.)

- The boolean expression is enclosed in parentheses.

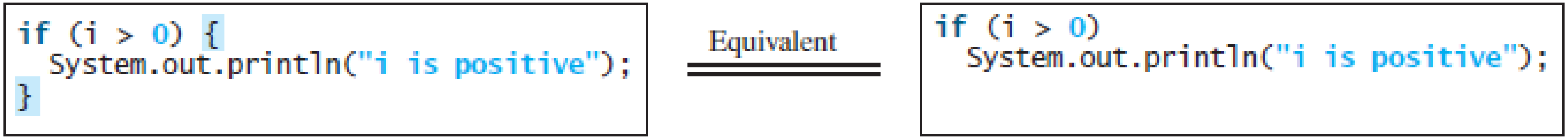


```
if i > 0 {
  System.out.println("i is positive");
}
```

```
if (i > 0) {
  System.out.println("i is positive");
}
```

(a) Wrong (b) Correct

- The block braces can be omitted if they enclose a single statement.



```
if (i > 0) {
  System.out.println("i is positive");
}
```

Equivalent

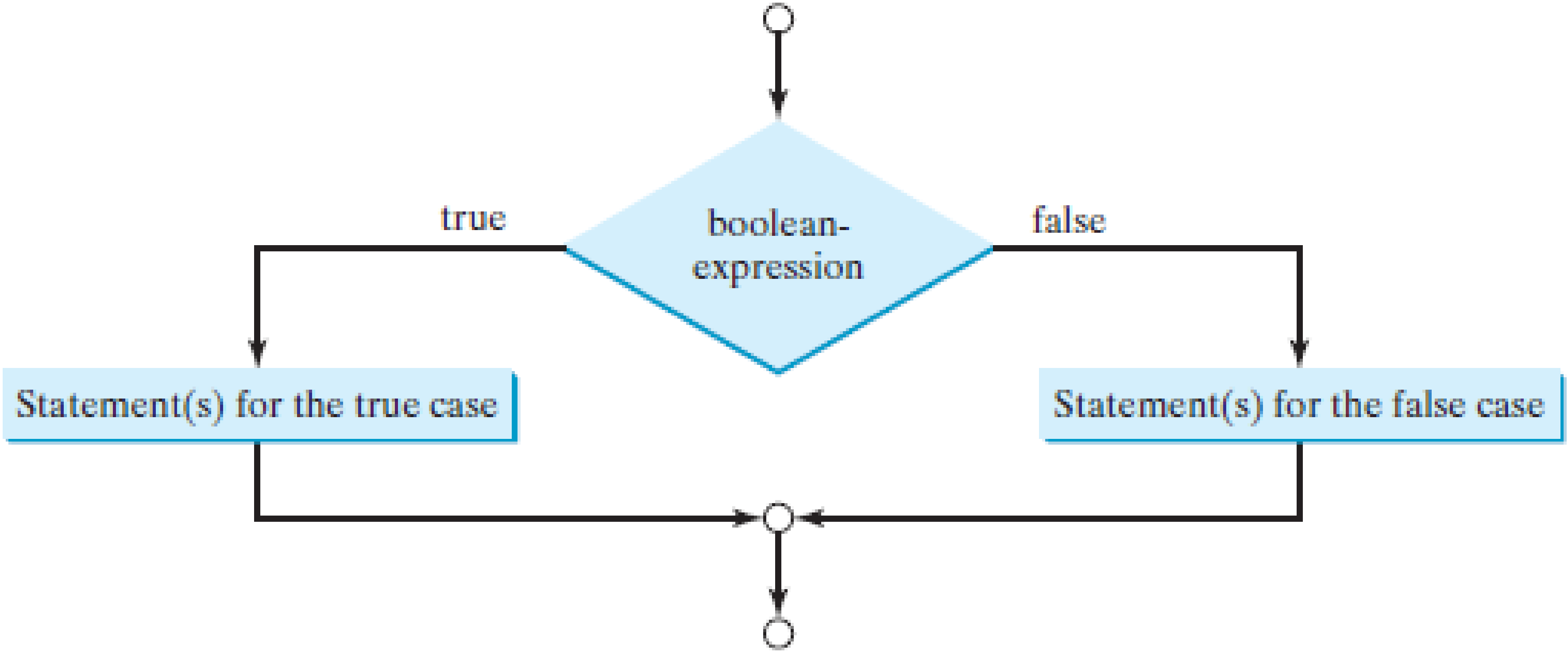
```
if (i > 0)
  System.out.println("i is positive");
```

Two-way If-else Statements

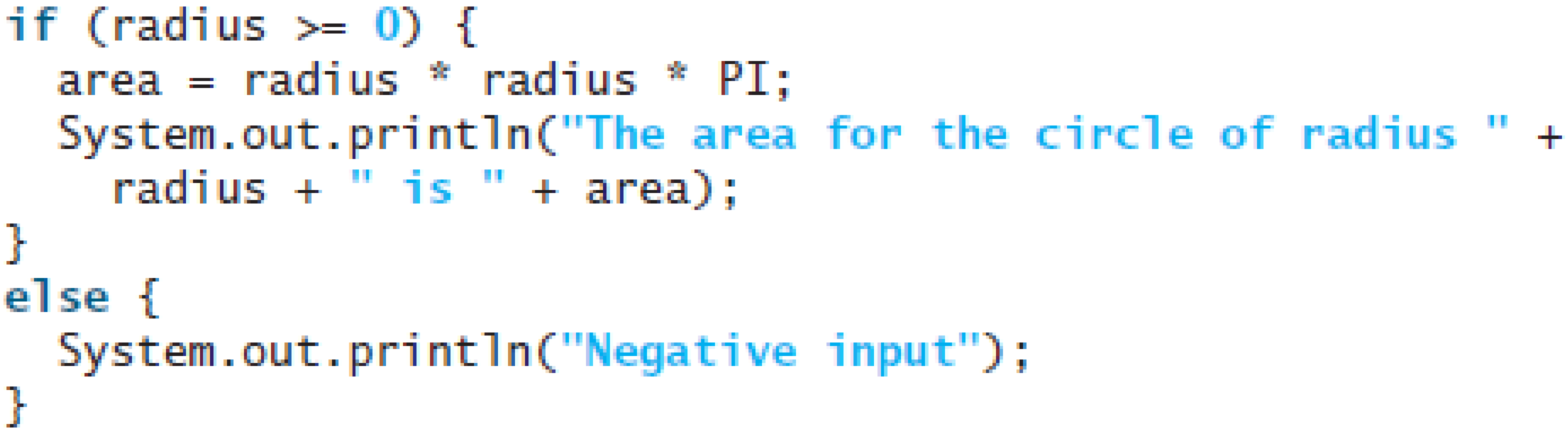
- A *two-way if-else* statement executes an action if the condition is *true* and another action if the condition is *false*.
- The syntax for a two-way if-else statement is:

```
if (boolean-expression){
    statement(s)-for-the-true-case;
}
else{
    statement(s)-for-the-false-case;
}
```

Two-way If-else Statements (Cont.)


true
boolean-expression
false
Statement(s) for the true case
Statement(s) for the false case

Two-way If-else Statements (Example)



```
if (radius >= 0) {
  area = radius * radius * PI;
  System.out.println("The area for the circle of radius " +
    radius + " is " + area);
}
else {
  System.out.println("Negative input");
}
```

Nested If and Multi-Way if-else Statements

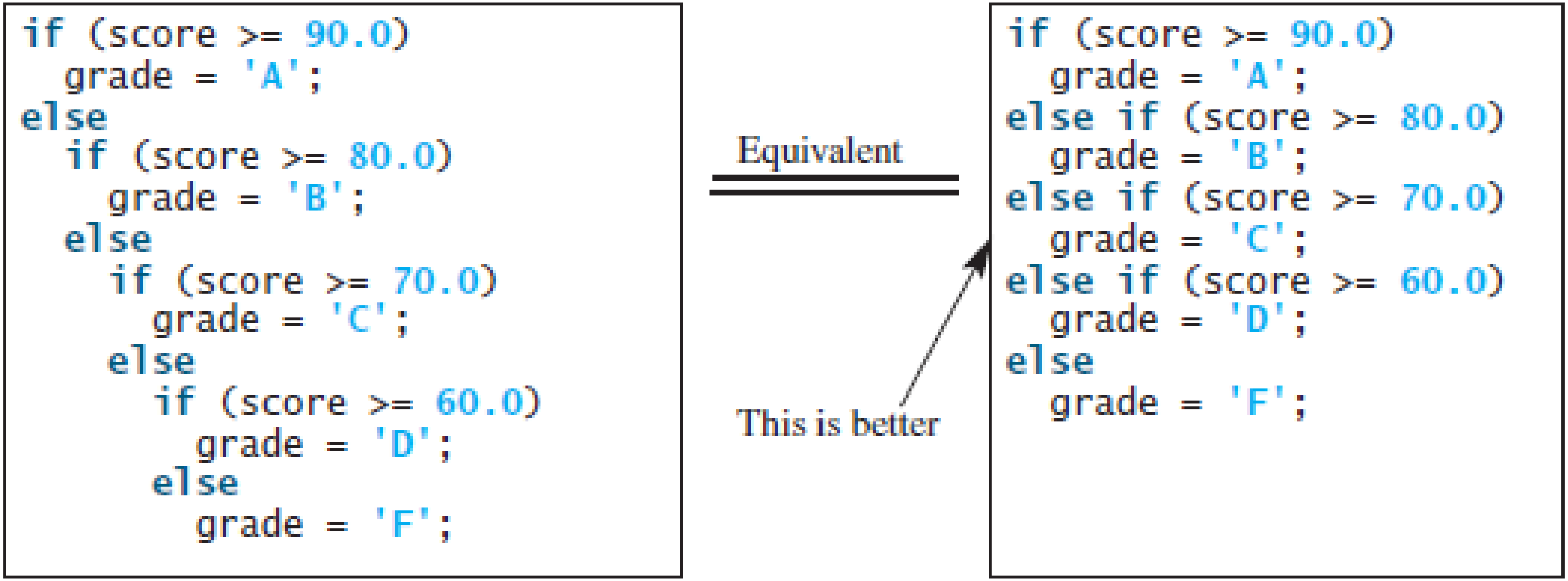
- An if statement can be inside another if statement to form a *nested* if statement.
- Example:

```
if (i > k){
    if (j > k)
        System.out.println(“i and j are greater than k”);
}
else
    System.out.println(“i is less than or equal to k”);
```

Executed only if i>k and j>k

Executed if i<=k

Nested If and Multi-Way if-else Statements (Example)



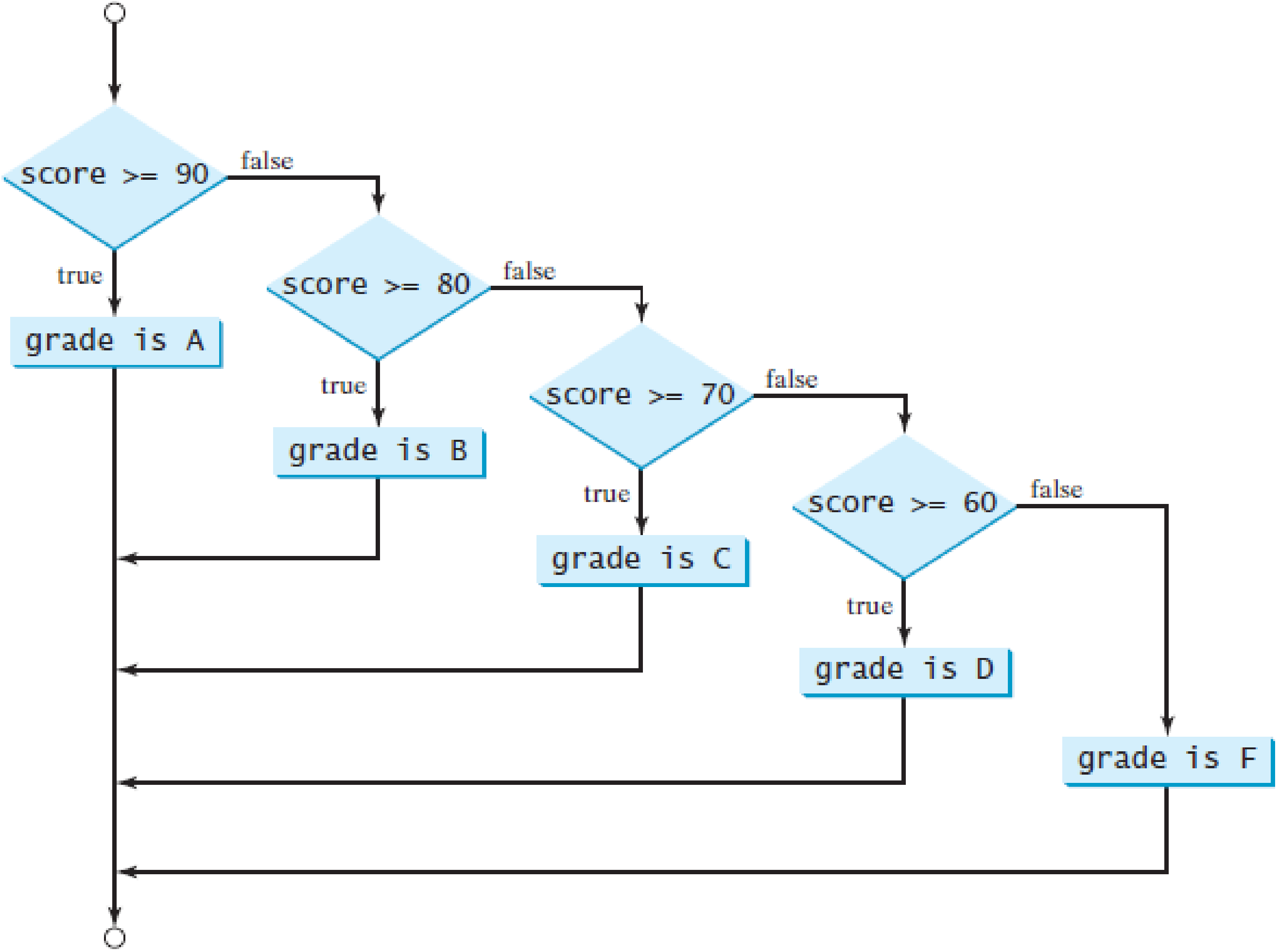
```
if (score >= 90.0)
  grade = 'A';
else
  if (score >= 80.0)
    grade = 'B';
  else
    if (score >= 70.0)
      grade = 'C';
    else
      if (score >= 60.0)
        grade = 'D';
      else
        grade = 'F';
```

Equivalent

This is better

```
if (score >= 90.0)
  grade = 'A';
else if (score >= 80.0)
  grade = 'B';
else if (score >= 70.0)
  grade = 'C';
else if (score >= 60.0)
  grade = 'D';
else
  grade = 'F';
```

Nested If and Multi-Way if-else Statements (Example)

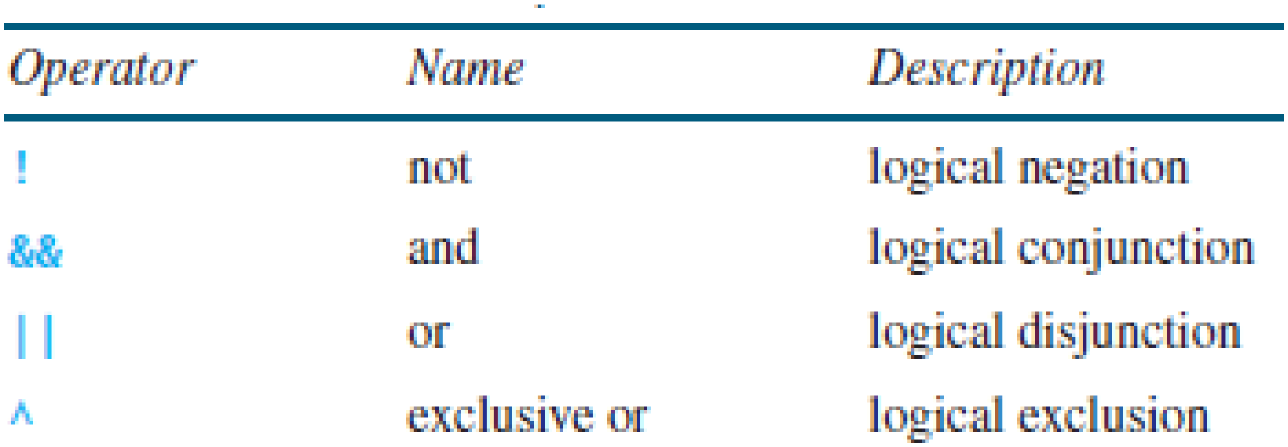


Note

- Check section 3.6 (Common Errors and Pitfalls).

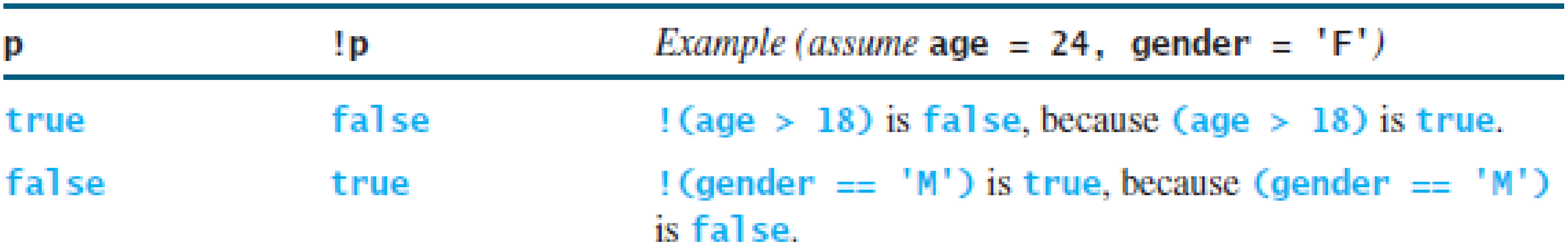
Logical Operators

- *Logical operators* can be used to create a *compound* Boolean expression.

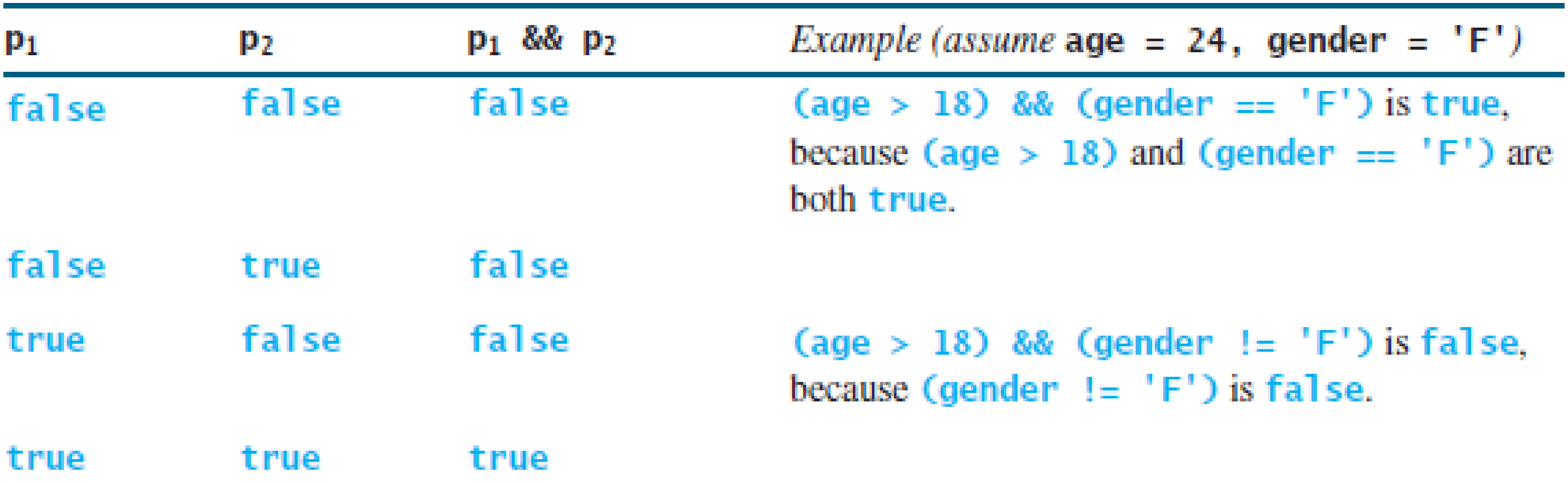


Operator	*Name*	*Description*
!	not	logical negation
&&	and	logical conjunction
||	or	logical disjunction
^	exclusive or	logical exclusion

Logical Operators (Cont.)

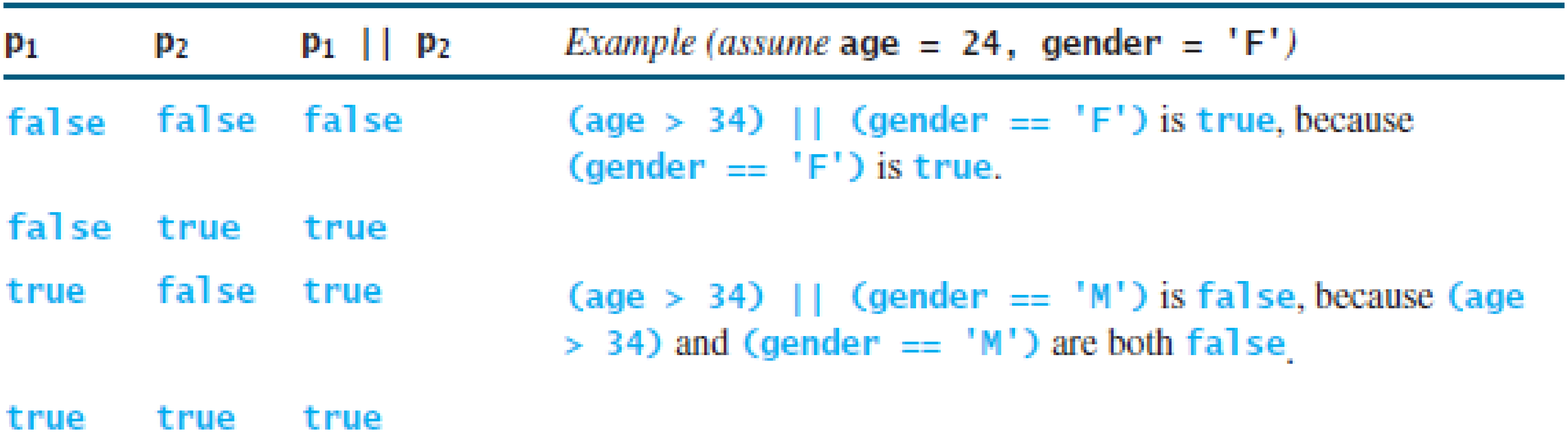


p	!p	*Example (assume* age = 24, gender = 'F'*)*
true	false	!(age > 18) is false, because (age > 18) is true.
false	true	!(gender == 'M') is true, because (gender == 'M') is false.

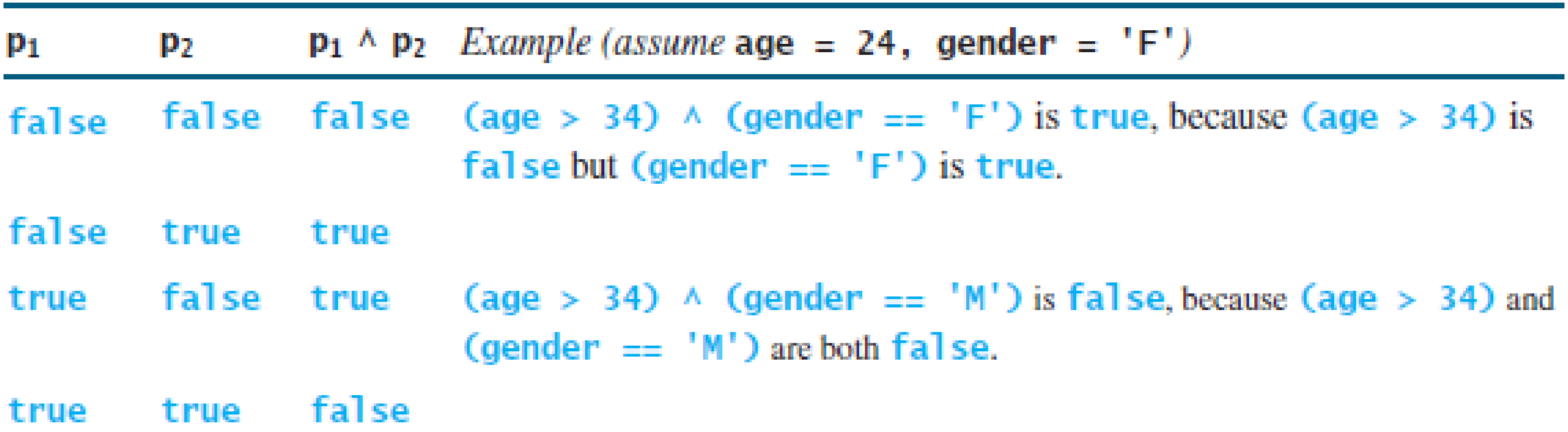


p_1	p_2	p_1 && p_2	*Example (assume* age = 24, gender = 'F'*)*
false	false	false	(age > 18) && (gender == 'F') is true, because (age > 18) and (gender == 'F') are both true.
false	true	false	
true	false	false	(age > 18) && (gender != 'F') is false, because (gender != 'F') is false.
true	true	true	

Logical Operators (Cont.)



p1	p2	p1 || p2	*Example (assume* age = 24, gender = 'F'*)*
false	false	false	(age > 34) || (gender == 'F') is true, because (gender == 'F') is true.
false	true	true	
true	false	true	(age > 34) || (gender == 'M') is false, because (age > 34) and (gender == 'M') are both false.
true	true	true	



p1	p2	p1 ^ p2	*Example (assume* age = 24, gender = 'F'*)*
false	false	false	(age > 34) ^ (gender == 'F') is true, because (age > 34) is false but (gender == 'F') is true.
false	true	true	
true	false	true	(age > 34) ^ (gender == 'M') is false, because (age > 34) and (gender == 'M') are both false.
true	true	false	

switch Statements

- Nested if can be used to write code for multiple conditions.
  - However, it makes the program difficult to read.
- A *switch* statement simplifies coding for multiple conditions.
- A *switch* statement executes statements based on the value of a variable or an expression.

switch Statements (Cont.)

- The syntax for the switch statement is:

```
switch (switch-expression){
   case value1: statement(s)1;
                break;
   case value2: statement(s)2;
                break;
   .....
   case valueN: statement(s)N;
                break;
   default:     statement(s)-for-default;
}
```

Must yield a value of char, byte, short, int, or string

Constant expressions of the same type as the value of switch-expression

When the value in a case statement matches the value of the switch-expression, statements starting from this case are executed until either a break statement or the end of the switch statement is reached

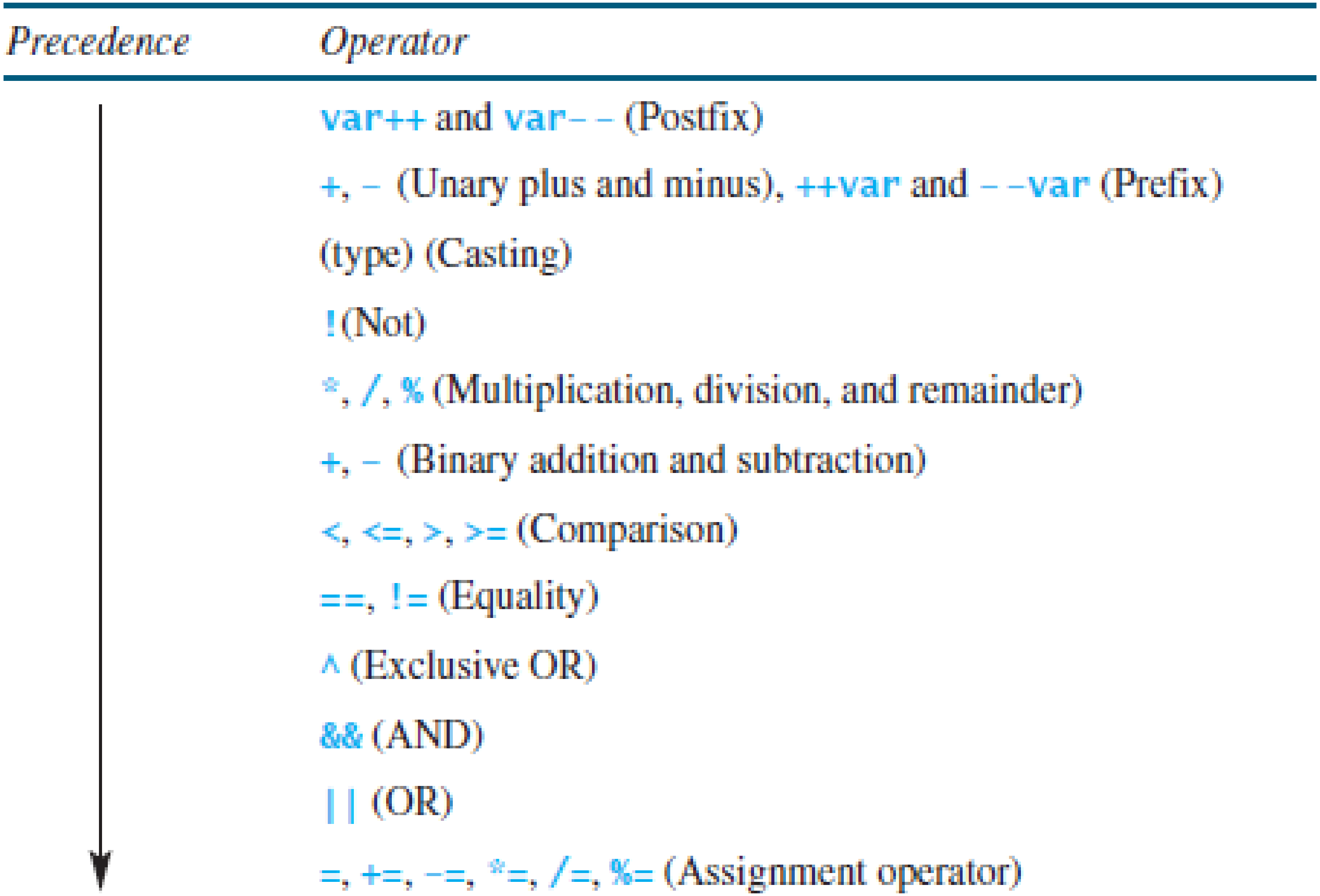
Statements of the default case are executed when none of the specified cases matches the switch-expression.

Conditional Expressions

- A *conditional expression* evaluates an expression based on a condition.
- The syntax is:
  - *boolean-expression ? expression1 : expression2;*
  - The result of the conditional expression is *expression1* if *boolean-expression* is *true*, otherwise the result is *expression2*.
- Example:

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

Operators Precedence Revisited



Precedence	*Operator*
	var++ and var-- (Postfix)
	+, - (Unary plus and minus), ++var and --var (Prefix)
	(type) (Casting)
	!(Not)
	*, /, % (Multiplication, division, and remainder)
	+, - (Binary addition and subtraction)
	<, <=, >, >= (Comparison)
	==, != (Equality)
	^ (Exclusive OR)
	&& (AND)
	|| (OR)
	=, +=, -=, *=, /=, %= (Assignment operator)

Loops

- A *loop* can be used to tell a program to execute statements *repeatedly*.
- Three types of loop statements:
  - While loops.
  - Do-while loops.
  - For loops.

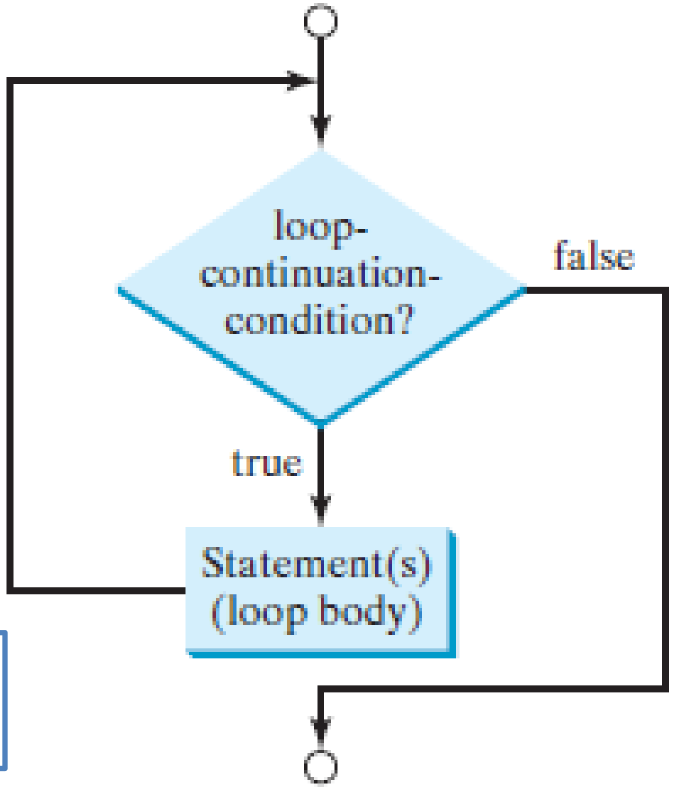
While Loops

- A *while* loop executes statements repeatedly while the condition is *true*.
- The syntax for the *while* loop is:

```
while (loop-continuation-condition){
    statement(s);
}
```

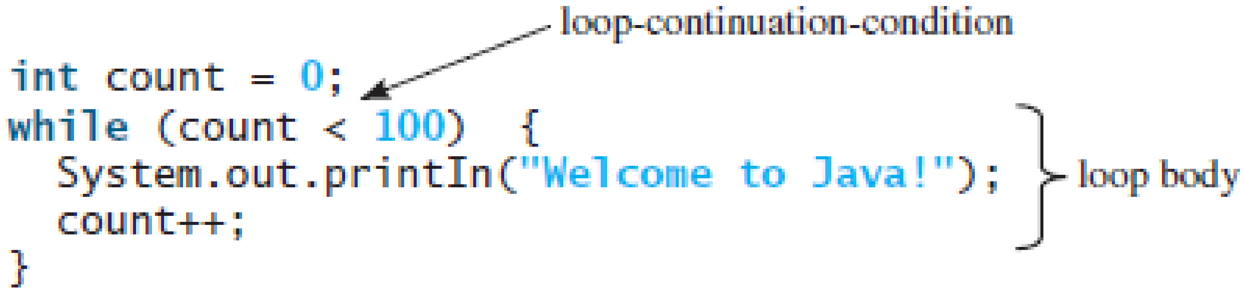
Loop body

Evaluated each time to determine whether to execute the loop body



While Loops (Cont.)

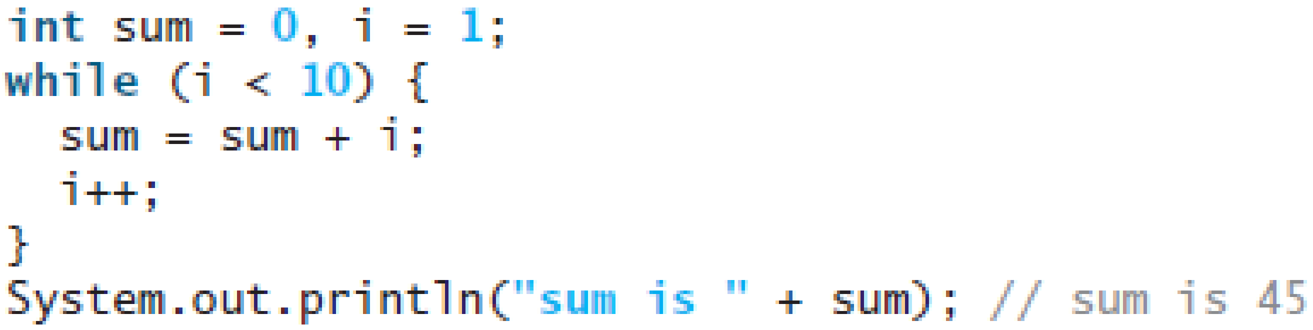
- A *while* loop that displays “Welcome to Java!” a hundred times:



```
int count = 0;
while (count < 100)  {
  System.out.printIn("Welcome to Java!");
  count++;
}
```

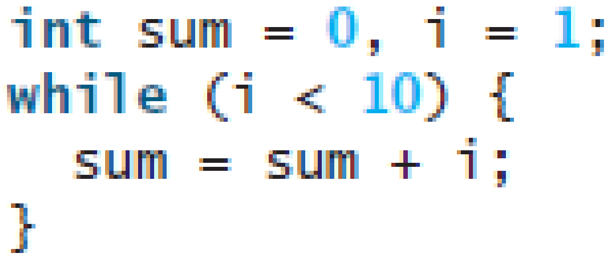
- Two types of loops:
  - *Counter-controlled* loops
    - A control variable is used to count the number of iterations.
  - *Sentinel-controlled* loops
    - A special input value signifies the end of the iterations.

While Loops (Examples)



```java
int sum = 0, i = 1;
while (i < 10) {
  sum = sum + i;
  i++;
}
System.out.println("sum is " + sum); // sum is 45
```

- Wrong implementation of a loop:

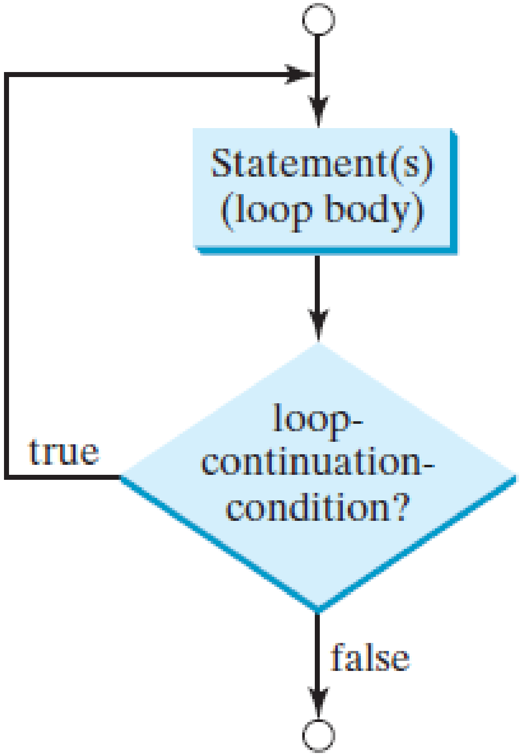


```java
int sum = 0, i = 1;
while (i < 10) {
  sum = sum + i;
}
```

The do-While Loops

- Same as the *while* loop except that it executes the loop body first then checks the loop continuation condition.
- The syntax for the *do-while* loop:

```
do {
   statement(s);
} while (loop-continuation-condition);
```

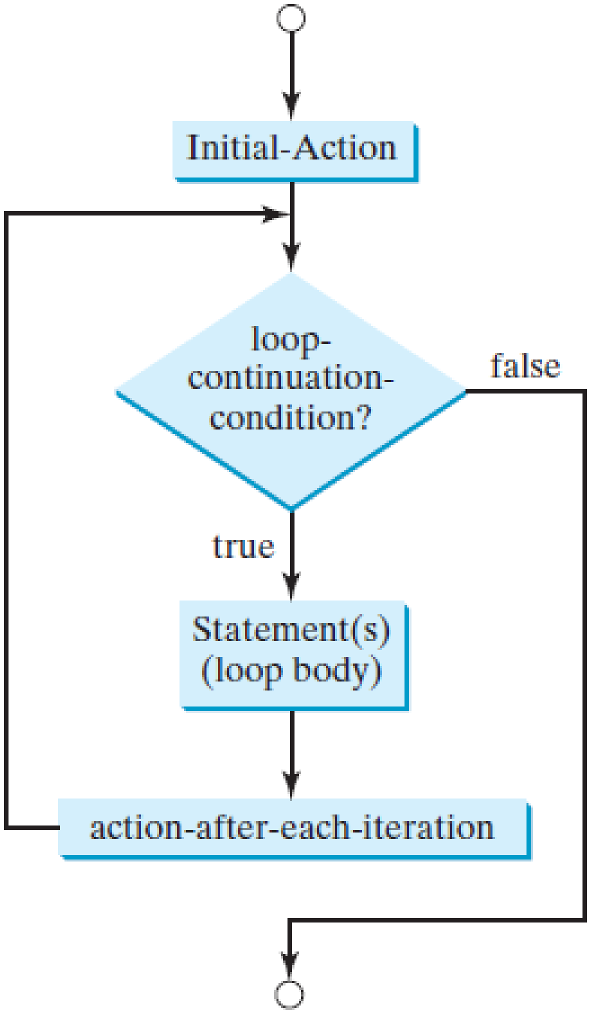


The for Loop

- A for loop has a concise syntax for writing loops.
- The syntax for the for loop is:

```
for (initial-action;  loop-continuation-condition;
action-after-each-iteration){
      statement(s);
}
```

The for Loop (Cont.)



The for Loop (Cont.)

- A *for* loop that displays “Welcome to Java!” a hundred times:

```
 for (int i = 0; i < 100; i++){
     System.out.println(“Welcome to Java!”);
 }
```

- The *initial-condition* in a *for* loop can be a list of zero or more comma-separated variable declaration/assignment statements:

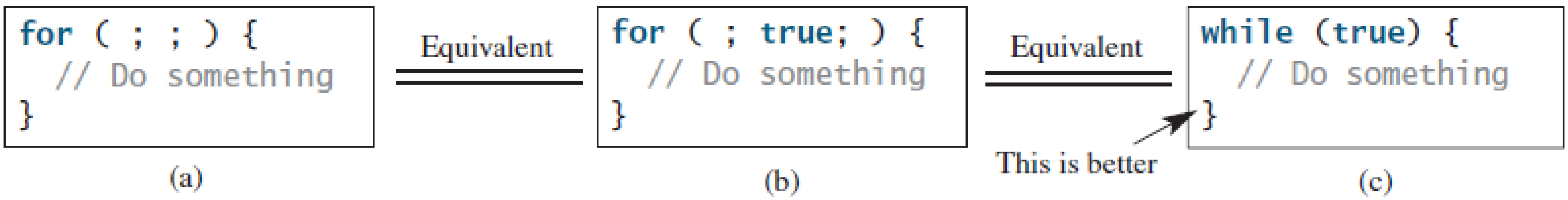
```
for (int i = 0, j = 0; (i + j < 10); i++, j++) {
        //Do something
}
```

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

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

Infinite Loops

- Examples of *infinite* loops



```
for ( ; ; ) {
  // Do something
}
```

(a)

Equivalent

```
for ( ; true; ) {
  // Do something
}
```

(b)

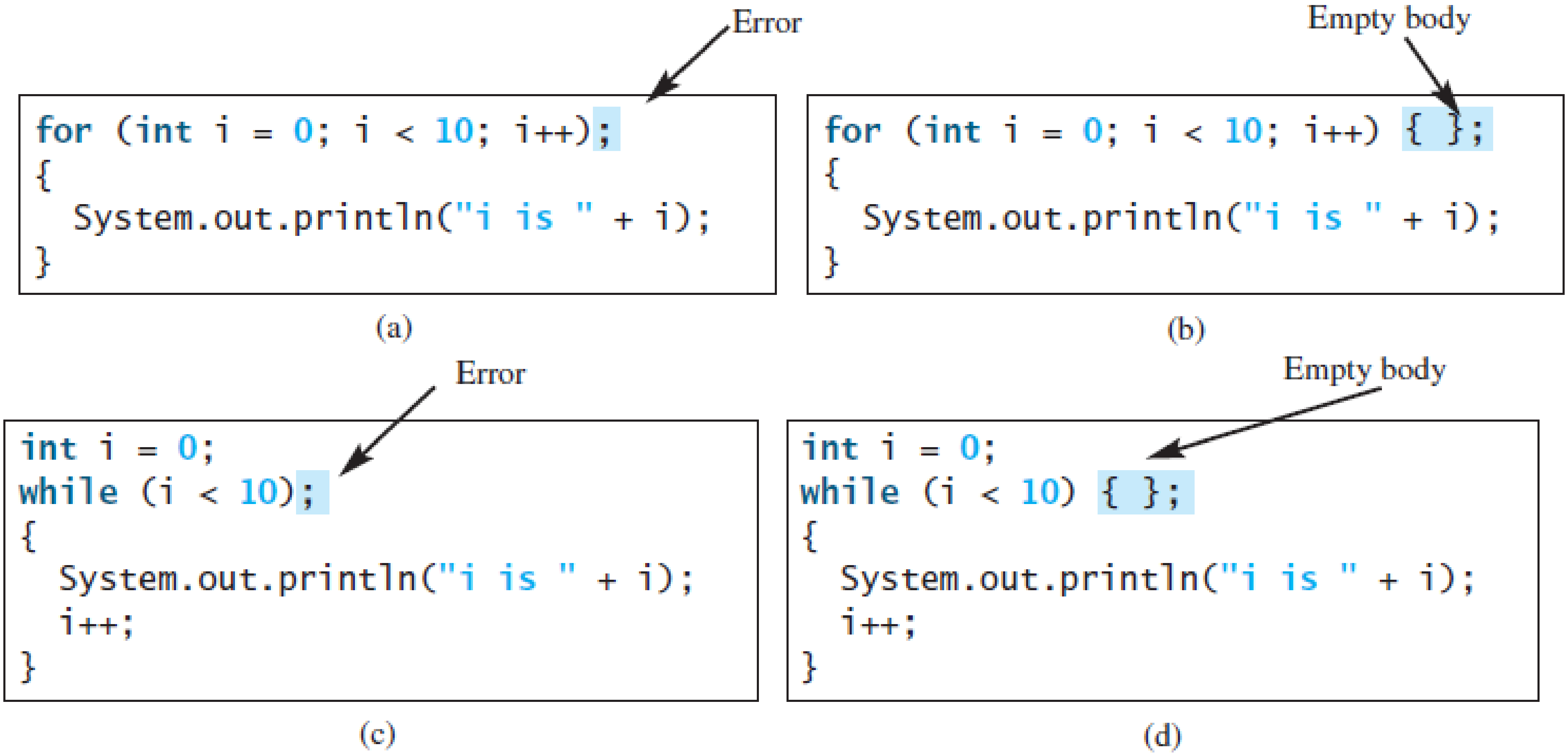
Equivalent

```
while (true) {
  // Do something
}
```

This is better

(c)

Common Errors



Error

```java
for (int i = 0; i < 10; i++);
{
  System.out.println("i is " + i);
}
```

(a)

Empty body

```java
for (int i = 0; i < 10; i++) { };
{
  System.out.println("i is " + i);
}
```

(b)

Error

```java
int i = 0;
while (i < 10);
{
  System.out.println("i is " + i);
  i++;
}
```

(c)

Empty body

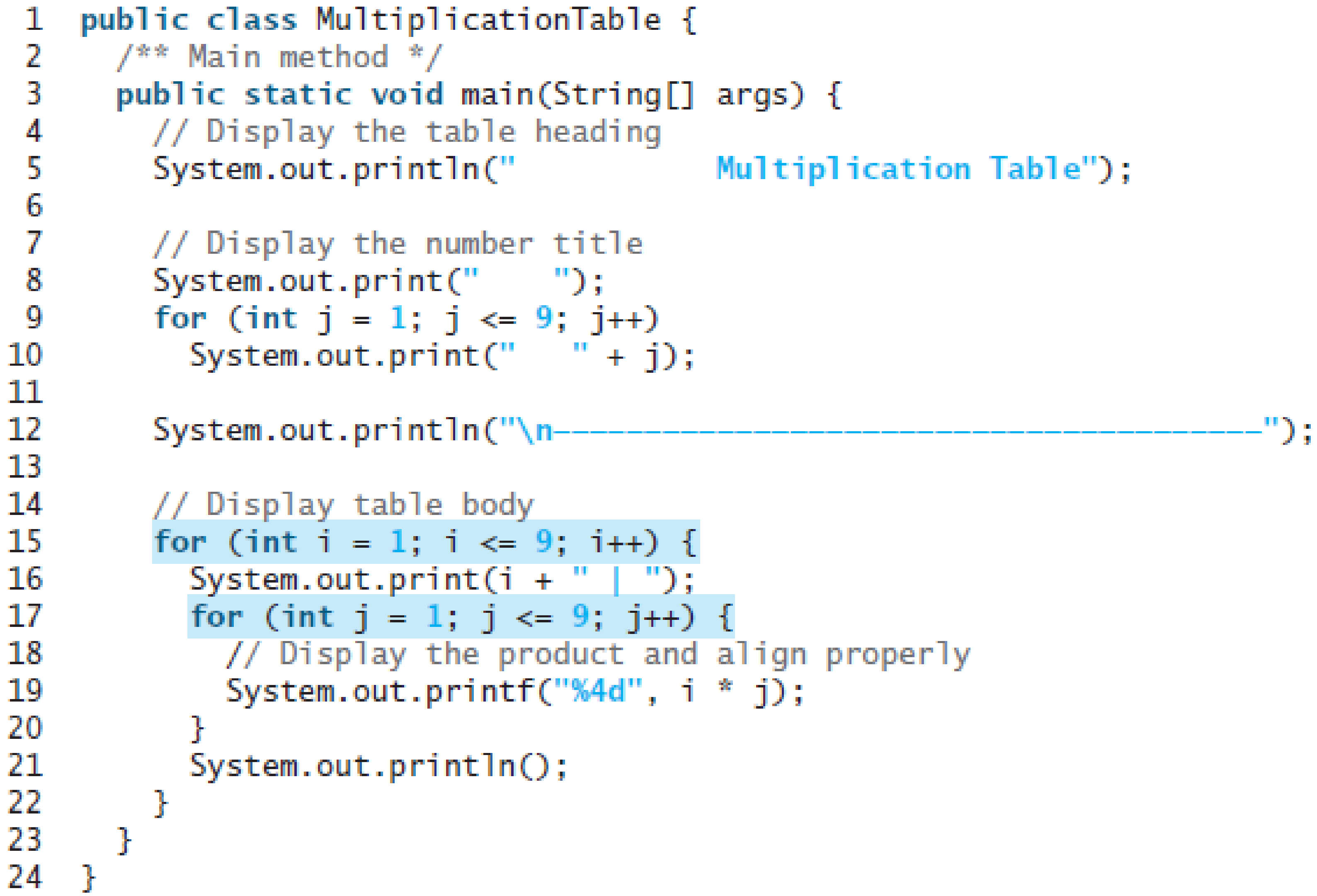
```java
int i = 0;
while (i < 10) { };
{
  System.out.println("i is " + i);
  i++;
}
```

(d)

Nested Loops

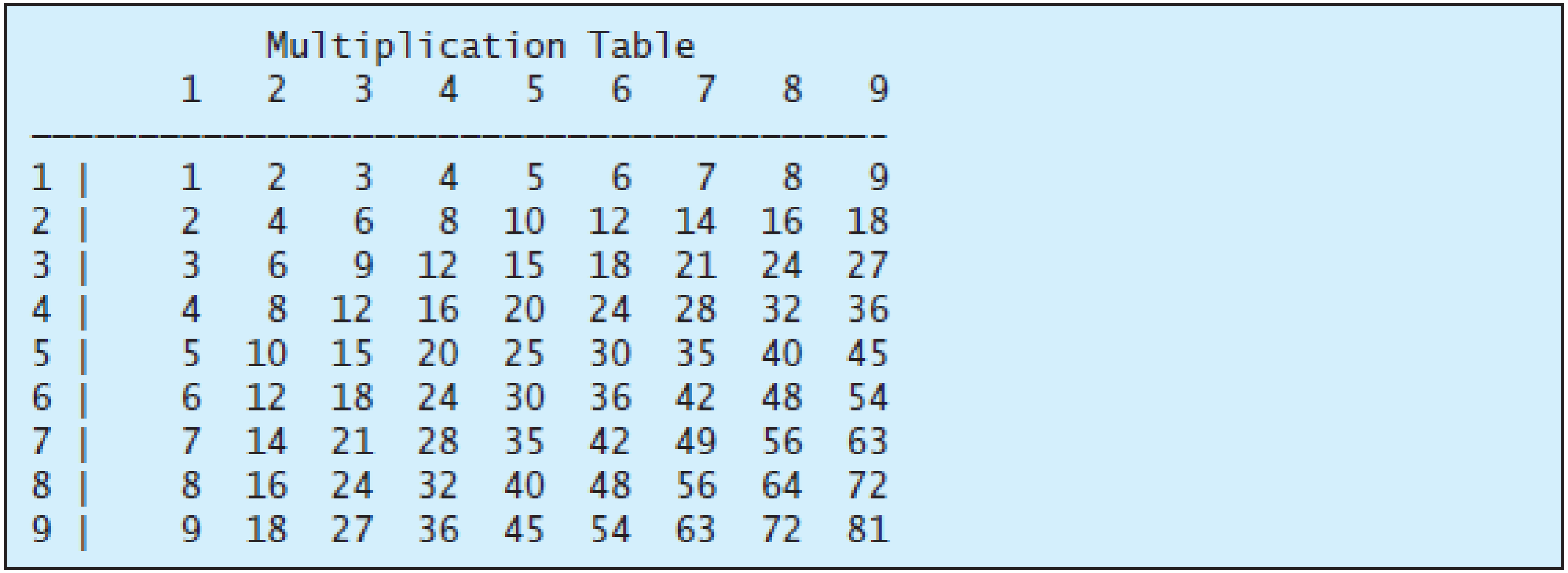
- Nested loops consist of an *outer* loop and one or more *inner* loops.
- Each time, the outer loop is repeated, the inner loops are reentered.

Nested Loops (Example)



```java
public class MultiplicationTable {
  /** Main method */
  public static void main(String[] args) {
    // Display the table heading
    System.out.println("           Multiplication Table");

    // Display the number title
    System.out.print("    ");
    for (int j = 1; j <= 9; j++)
      System.out.print("   " + j);

    System.out.println("\n---------------------------------------");

    // Display table body
    for (int i = 1; i <= 9; i++) {
      System.out.print(i + " | ");
      for (int j = 1; j <= 9; j++) {
        // Display the product and align properly
        System.out.printf("%4d", i * j);
      }
      System.out.println();
    }
  }
}
```

Nested Loops (Example)



```
           Multiplication Table
       1   2   3   4   5   6   7   8   9
----------------------------------------
1 |    1   2   3   4   5   6   7   8   9
2 |    2   4   6   8  10  12  14  16  18
3 |    3   6   9  12  15  18  21  24  27
4 |    4   8  12  16  20  24  28  32  36
5 |    5  10  15  20  25  30  35  40  45
6 |    6  12  18  24  30  36  42  48  54
7 |    7  14  21  28  35  42  49  56  63
8 |    8  16  24  32  40  48  56  64  72
9 |    9  18  27  36  45  54  63  72  81
```

Keywords *break* and *continue*

- The *break* and *continue* keywords provide additional controls in a loop.
- The *break* keyword is used in a loop to immediately terminate the loop.
- Example of using the *break* keyword:

```
for (int n=0, sum=0; n<20; n++){
        sum += n;
        if (sum >= 100) break;
}
```

Keywords *break* and *continue* (Cont.)

- The *continue* keyword is used in a loop to end the current iteration and program control goes to the end of the loop body.
- Example of using the *continue* keyword:

```
for (int n=0, sum=0; n<20; n++){
      if (n == 10 || n == 11) continue;
     sum += n;
}
```

Object-Oriented Problem Solving

Methods

Based on Chapter 6 of “Introduction to Java Programming” by Y. Daniel Liang.

Eng. Asma Abdel Karim
Computer Engineering Department

Outline

- Introduction (6.1)
- Defining a method (6.2)
- Calling a method (6.3)
- A void Method Example (6.4)
- Passing Arguments by Values (6.5)
- Modularizing Code (6.6)
- Overloading methods (6.8)
- The scope of variables (6.9)

Introduction

- A *method* is a collection of statements grouped together to perform an operation.
- *Methods* can be used to define reusable code and organize and simplify code.

Example of a Reusable Code

```
int sum = 0;
for ( int i = 1; i <= 10; i++)
     sum += i;
System.out.println(“Sum from 1 to 10 is “+ sum);
```

Compute the sum from 1 to 10

```
int sum = 0;
for ( int i = 20; i <= 37; i++)
     sum += i;
System.out.println(“Sum from 20 to 37 is “+ sum);
```

Compute the sum from 20 to 37

```
int sum = 0;
for ( int i = 35; i <= 49; i++)
     sum += i;
System.out.println(“Sum from 35 to 49 is “+ sum);
```

Compute the sum from 35 to 49

Example of a Reusable Code (Cont.)

- It would be nice to write the common code once and reuse it.
- This is achieved by:
  - *Defining* a method that contains the common code.
  - Reuse it by *invoking* it with different values.

```
public static int sum (int i1, int i2){
    int result= 0;
    for ( int i = i1; i <= i2; i++)
        result += i;
    return result;
}
```

```
public static void main (String [] args){
    System.out.println(“Sum from 1 to 10
    is” + sum (1, 10) );
     System.out.println(“Sum from 20 to 37
    is” + sum (20, 37) );
     System.out.println(“Sum from 35 to 49
    is” + sum (35, 49) );
}
```

Defining a Method

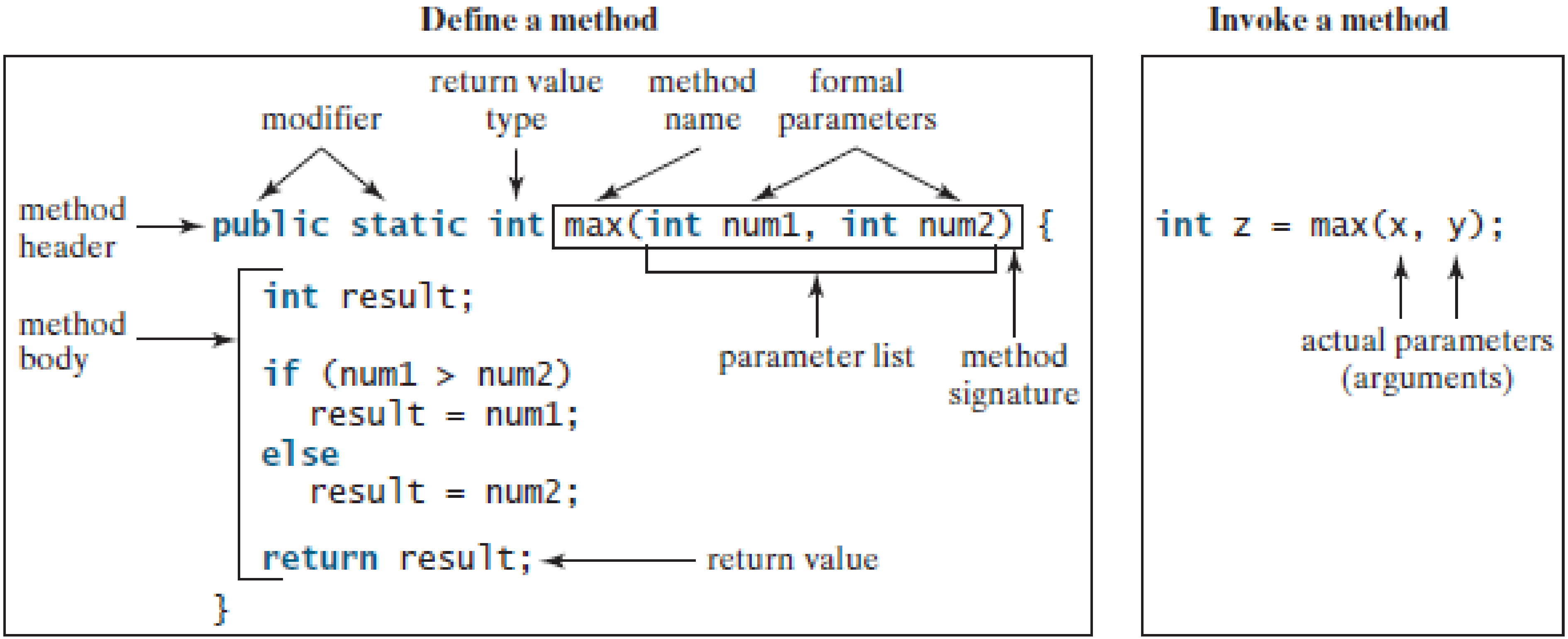
- A method definition consists of:
  - Return value type.
  - Method name.
  - Parameters.
  - Body.
- The syntax for defining a method is:

```
modifier returnValueType methodName (list of parameters){
    //method body
}
```

Defining a Method (Cont.)

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

Method Definition: An Example



```
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

```
int z = max(x, y);
```

- In a method definition, you define what the method is to do.

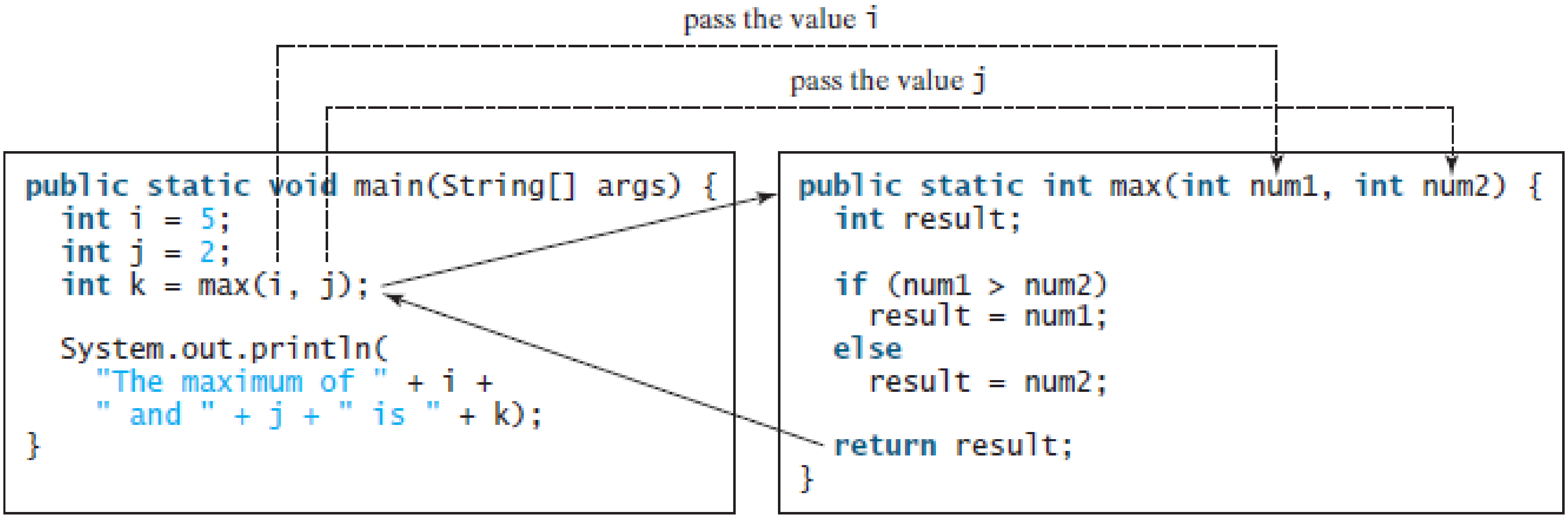
Calling a method

- Calling a method executes the code in the method.
- There are two ways to call a method, depending on whether the method returns a value or not.
- If a method returns a value, a call to the method is usually treated as a value.
  - *int larger = max(3, 4);*

  //calls max(3, 4) and assigns the result of the method to the variable larger.
  - *System.out.println(max(3, 4));*

  //prints the return value of the method call max(3, 4).
- If a method returns *void*, a call to the method must be a statement.
  - For example, the method *println* returns *void*. The following call is a statement:
  - *System.out.println("Welcome to Java!");*

Method Invocation: An Example

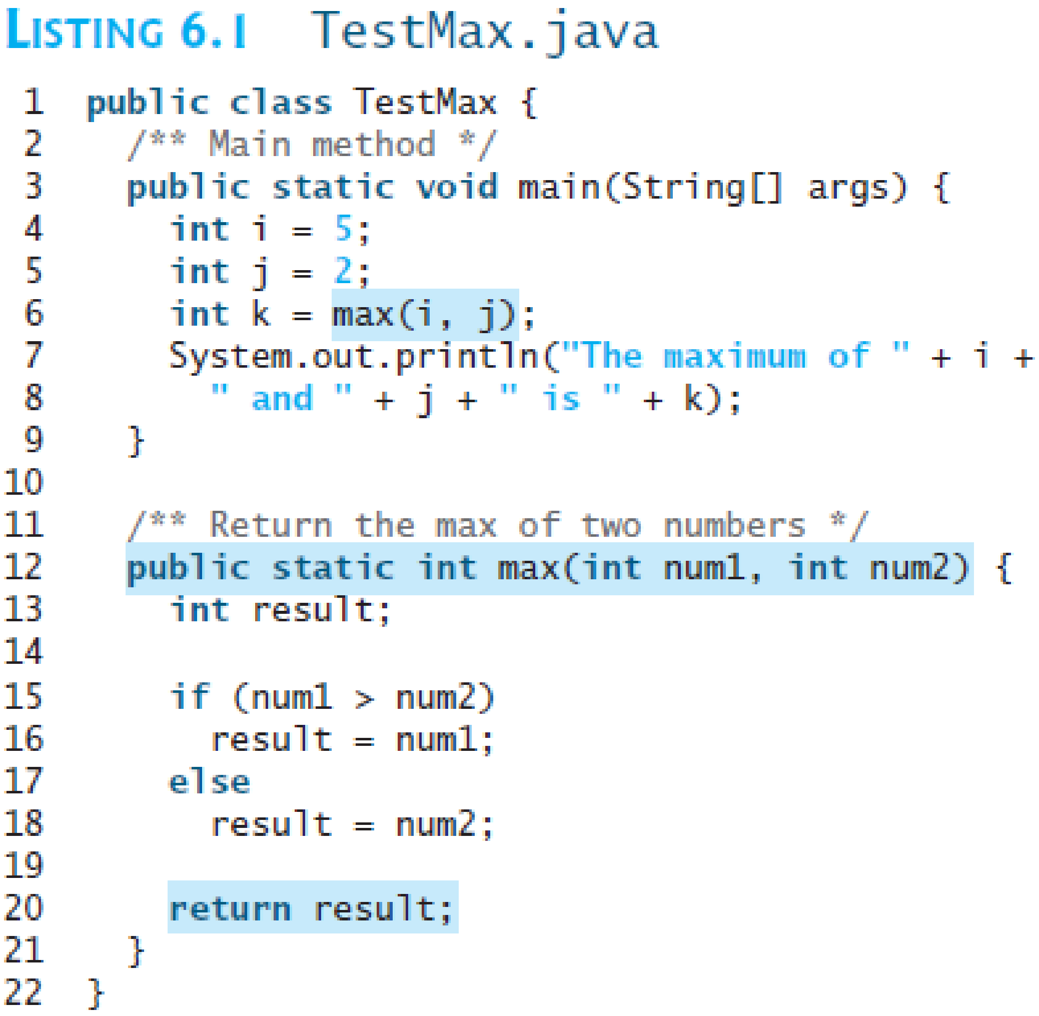


```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
    "The maximum of " + i +
    " and " + j + " is " + k);
}
```

```
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

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

TestMax.java



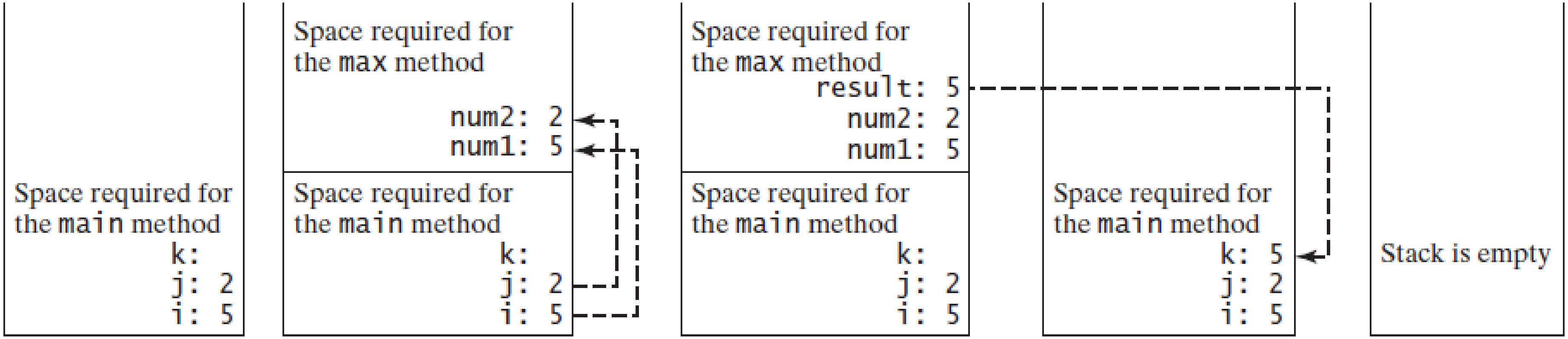
LISTING 6.1 TestMax.java

```java
public class TestMax {
  /** Main method */
  public static void main(String[] args) {
    int i = 5;
    int j = 2;
    int k = max(i, j);
    System.out.println("The maximum of " + i +
      " and " + j + " is " + k);
  }

  /** Return the max of two numbers */
  public static int max(int num1, int num2) {
    int result;

    if (num1 > num2)
      result = num1;
    else
      result = num2;

    return result;
  }
}
```

What happens when a method is invoked?

- Each time a method is invoked, the system creates an *activation record*.
  - *Activation record* stores parameters and variables for the method .
  - *Activation record* is placed in an area of memory known as the *call stack*, or simply the *stack*.
- When a method invokes another method, the caller's activation record is kept intact, and a new activation record is created.
- When a method finishes its work and returns to its caller, its activation record is removed from the stack.
- A call stack stores methods in last-in, first-out fashion.

What happens when a method is invoked? (Example)



Space required for the main method
k:
j: 2
i: 5
Space required for the max method
num2: 2
num1: 5
Space required for the main method
k:
j: 2
i: 5
Space required for the max method
result: 5
num2: 2
num1: 5
Space required for the main method
k:
j: 2
i: 5
Space required for the main method
k: 5
j: 2
i: 5
Stack is empty

(a) The main method is invoked.

(b) The max method is invoked.

(c) The max method is being executed.

(d) The max method is finished and the return value is sent to k.

(e) The main method is finished.

A *void* Method Example

- A *void* method does not return a value.

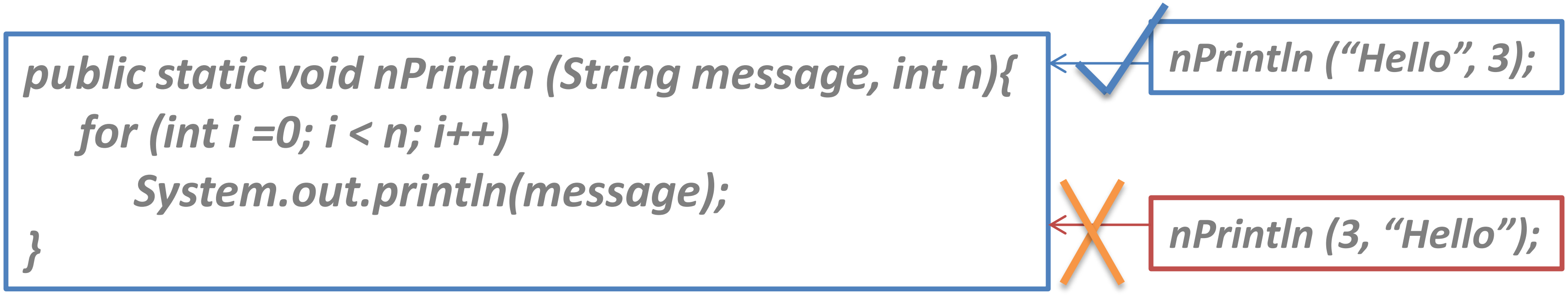
```
public static void printGrade(double score){
        if (score >= 90.0)
                System.out.println (‘A’);
        else if (score >= 80.0)
                System.out.println (‘B’);
        else if (score >= 70.0)
                System.out.println (‘C’);
        else if (score >= 60.0)
                System.out.println (‘D’);
        else
                System.out.println (‘F’);
}
```

Example of calling this method:

```
System.out.print (“The grade is ”);
printGrade(78.5);
```

Passing Arguments by Values

- When calling a method, you need to provide *arguments*, which must match the *parameters* defined in the method signature in:
  - Order
  - Number.
  - Compatible type.



```
public static void nPrintln (String message, int n){
    for (int i =0; i < n; i++)
        System.out.println(message);
}
```

✓ `nPrintln (“Hello”, 3);`

✗ `nPrintln (3, “Hello”);`

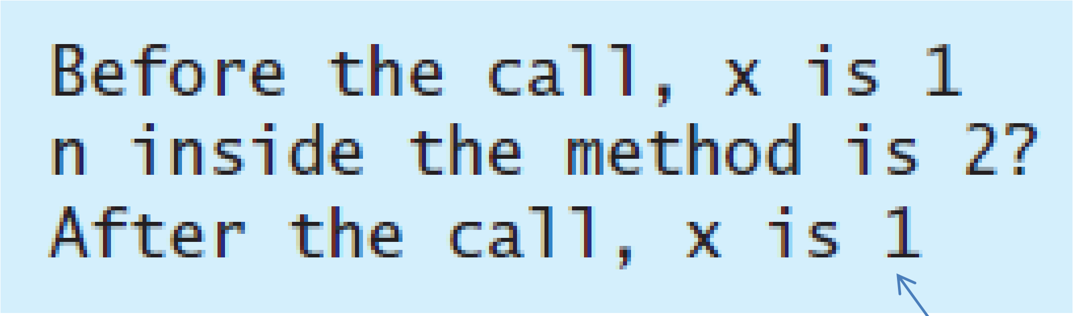
Passing Arguments by Values (Cont.)

- When you invoke a method with an argument, the value of the argument is passed to the parameter.
  - This is referred to as *pass-by-value*.
- If a value of a variable is passed as an argument to a parameter, <u>the variable is not affected</u>, regardless of the changes made to the parameter inside the method.

Passing Arguments by Values (Example)

```
public class Increment{
    public static void main (String [] args){
            int x = 1;
            System.out.println(“Before the call, x is ”+   x);
            increment (x);
            System.out.println(“After the call, x is ”+      x);
    }

    public static void increment (int n){
            n++;
            System.out.println(“n inside the method is ” + n);
    }
}
```

Passing Arguments by Values (Example Cont.)



```
Before the call, x is 1
n inside the method is 2?
After the call, x is 1
```

Value of x does not change

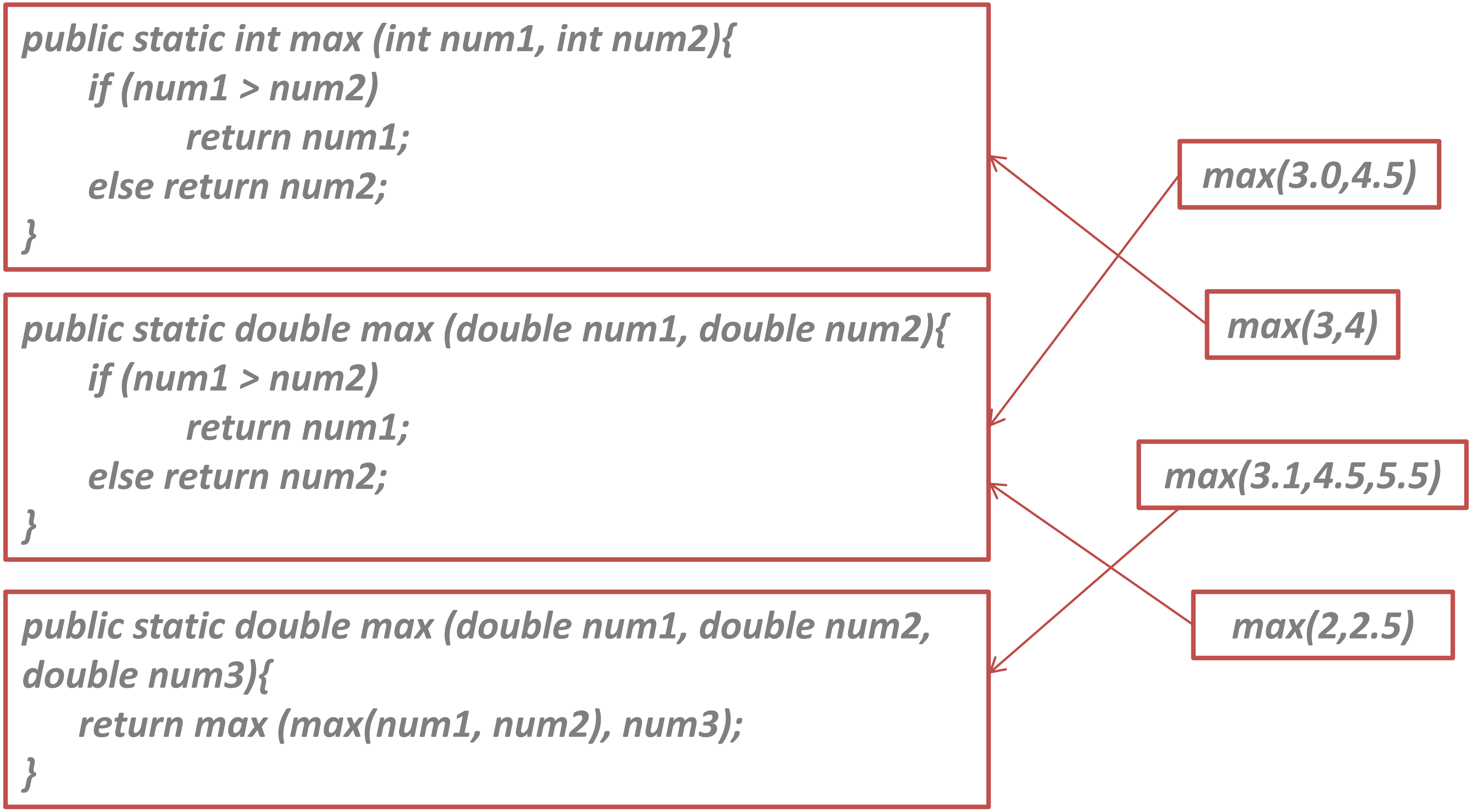
Modularizing Code

- Modularizing makes the code:
  - Clear and easy to read.
    - Isolates parts used to perform specific computations from the rest of the code.
  - Easy to maintain and debug.
    - Narrows the scope of debugging.
  - Reusable.
    - Code can be reused by other programs.

Overloading Methods

- Overloading methods enables you to define the methods with the same name as long as their signatures are different.
- Methods overloading is having two or more methods that have the *same name*, but *different parameter lists* within one class.
- The Java compiler determines which method to use based on the *method signature*.
  - It finds the most specific method for a method invocation.

Overloading Methods: An Example



```
public static int max (int num1, int num2){
      if (num1 > num2)
               return num1;
      else return num2;
}
```

```
public static double max (double num1, double num2){
      if (num1 > num2)
               return num1;
      else return num2;
}
```

```
public static double max (double num1, double num2,
double num3){
     return max (max(num1, num2), num3);
}
```

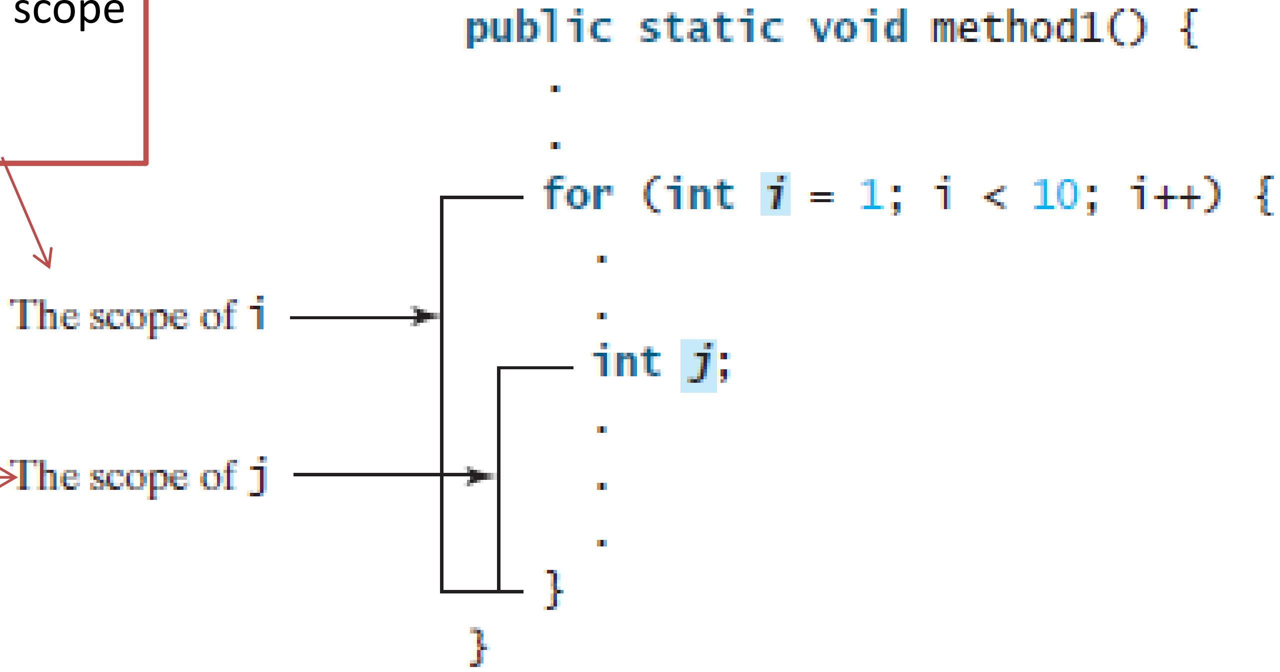
The Scope of Variables

- The *scope* of a variable is the part of the program where the variable can be referenced.
- A variable defined inside a method is referred to as a *local variable*.
  - The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable.
  - A local variable must be declared and assigned a value before it can be used.
- A parameter is actually a local variable.
  - The scope of a method parameter covers the entire method.

The Scope of Variables (Cont.)

A variable declared in the initial-action part of a for-loop header has its scope in the entire loop.

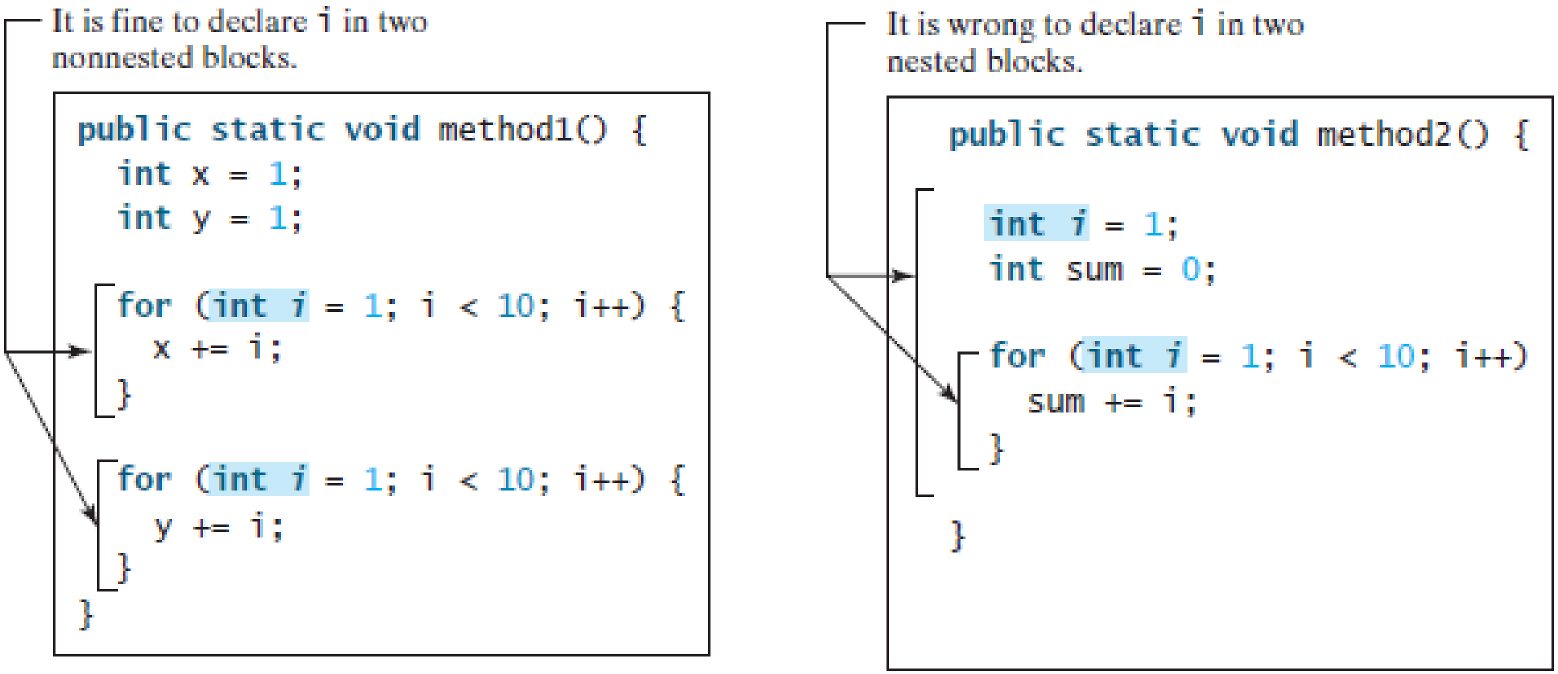
A variable declared inside a for-loop body has its scope limited in the loop body from its declaration to the end of the block.



```
public static void method1() {
    .
    .
    for (int i = 1; i < 10; i++) {
        .
        .
        int j;
        .
        .
        .
    }
}
```

The Scope of Variables (Cont.)

- You can declare a local variable with the same name in different blocks in a method.
- But you cannot declare a local variable twice in the same block or in nested blocks.



It is fine to declare i in two nonnested blocks.

```
public static void method1() {
  int x = 1;
  int y = 1;

  for (int i = 1; i < 10; i++) {
    x += i;
  }

  for (int i = 1; i < 10; i++) {
    y += i;
  }
}
```

It is wrong to declare i in two nested blocks.

```
public static void method2() {

    int i = 1;
    int sum = 0;

    for (int i = 1; i < 10; i++)
      sum += i;
    }

}
```

Object-Oriented Problem Solving

Arrays

Based on Chapters 7 & 8 of “Introduction to Java Programming” by Y. Daniel Liang.

Eng. Asma Abdel Karim
Computer Engineering Department

Outline

- Introduction (7.1)
- Array Basics - Declaring Arrays (7.2.1)
- Array Basics - Creating Arrays (7.2.2)
- Array Basics - Array Size and Default Values (7.2.3)
- Array Basics – Accessing Array Elements (7.2.4)
- Array Basics - Array Initializers (7.2.5)
- Array Bascis - Processing Arrays (7.2.6)
- Array Basics - Foreach Loops (7.2.7)
- Copying Arrays (7.5)
- Passing Arrays to Methods (7.6)
- Returning an Array from a Method (7.7)
- Variable-Length Argument Lists (7.9)
- Command-Line Arguments (7.13)
- Two-Dimensional Arrays Basics (8.2)
- Processing Two-Dimensional Arrays (8.3)

Introduction

- An *array* is a data structure which stores a fixed-size sequential collection of elements of the same type.
- A single array variable can reference a large collection of data.

Array Basics
Declaring Arrays

- To use an array in a program, you must *declare* a variable to reference the array and specify the array's elements type.
- The syntax for declaring an array variable is:

  elementType [] arrayRefVar;

- The *elementType* can be any data type.
  - All elements in the array will have the same data type.
- Example:

  double [] myList;

Array Basics
Declaring Arrays (Cont.)

- Unlike declarations for primitive data type variables, the declaration of an array variable does not allocate any space in memory for the array.
  - It creates only a storage location for the reference to an array.
  - If a variable does not contain a reference to an array, the value of the variable is *null*.

Array Basics
Creating Arrays

- An array is created using the *new* operator with the following syntax:

```
arrayRefVar = new elementType [arraySize];
```

  - This statement does two things:
    - It creates an array using new elementType [arraySize]
    - It assigns the reference of the newly created array to the variable arrayRefVar.

- Example:

```
double [] myList;                    //array declaration
mylist = new double [10];                  //array creation
```

Array Basics
Creating Arrays (Cont.)

- Declaring an array variable, creating an array, and assigning the reference of the array to the variable can be combined in one statement as:

  elementType [] arrayRefVar = new elementType [arraySize];

- Example:

  double [] myList = new double [10];

  – This statement declares an array variable, *myList*, creates an array of ten elements of *double* type, and assigns its reference to *myList*.

Array Basics
Creating Arrays (Cont.)

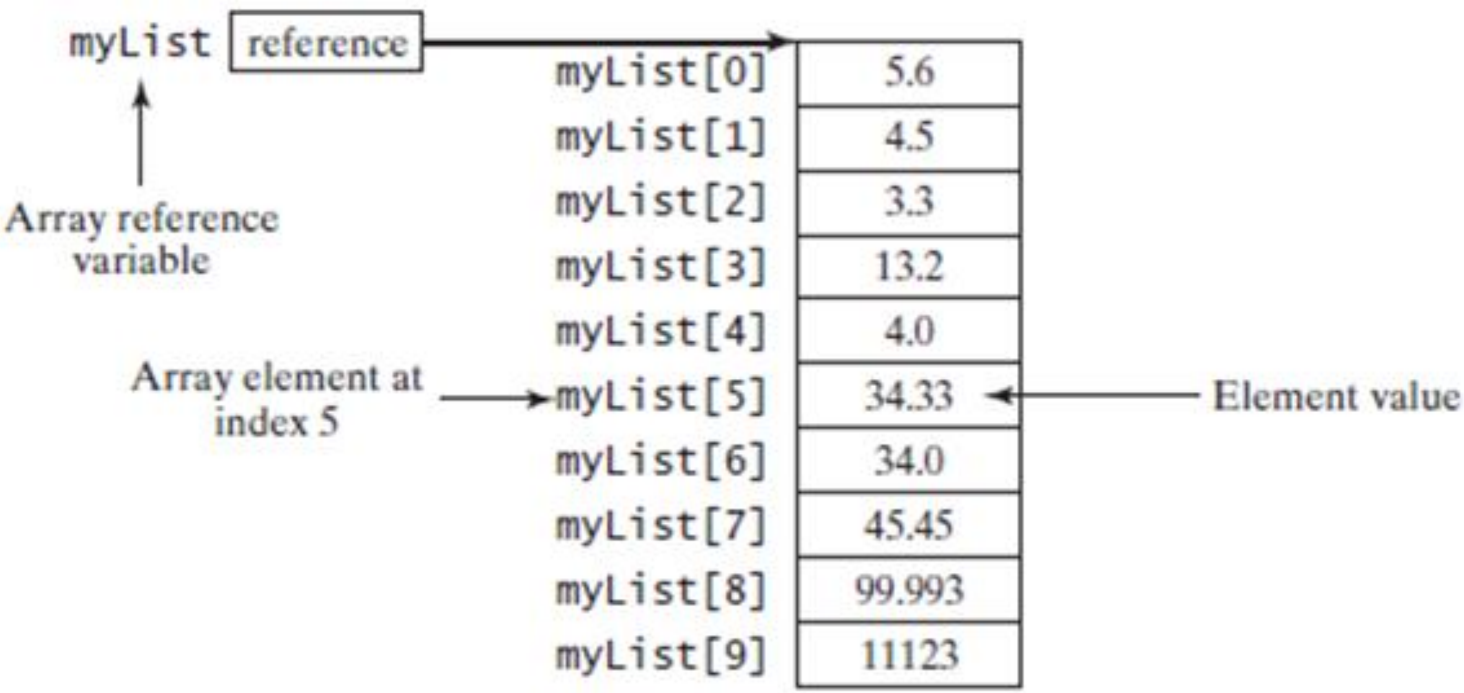
- The syntax to assign values to array elements:

```
 arrayRefVar [index] = value;
```

- Example:

```
double [] myList = new double [10];
myList[0] = 5.6;
myList[1] = 4.5;
myList[2] = 3.3;
          .
          .
myList[9] = 11123;
```

Array Basics
Creating Arrays (Cont.)



myList reference

Array reference variable

Array element at index 5

Element value

myList[0]	5.6
myList[1]	4.5
myList[2]	3.3
myList[3]	13.2
myList[4]	4.0
myList[5]	34.33
myList[6]	34.0
myList[7]	45.45
myList[8]	99.993
myList[9]	11123

- An array variable that appears to hold an array actually contains a reference to that array.
- Strictly speaking, an array variable and an array are different, but most of the time the distinction can be ignored.

Array Basics
Array Size and Default Values

- When space for an array is allocated, the array size must be given, specifying the number of elements that can be stored in it.
- The size of an array cannot be changed after the array is created.
- Array size can be obtained using: *arrayRefVar.length*
- When an array is created, its elements are assigned their default values:
  - *0* for *numeric* primitive data types.
  - *\u0000* for *char* types.
  - *False* for *boolean* types.

Array Basics
Accessing Array Elements

- The array elements are accessed through the index.
- Array indices are 0 based.
  - They range from *0* to *arrayRefVar.length-1*.
- Each element in the array is represented using the following syntax:

  arrayRefVar [index]

Array Basics
Accessing Array Elements (Cont.)

- An indexed variable can be used in the same way as a regular variable.
- Examples:

```
myList[2] = myList[0] + myList[1];
for (int i=0; i < myList.length; i++){
      myList[i] = i;
}
```

Array Basics
Array Initializers

- *Array initializer* is a shorthand notation which combines the *declaration*, *creation*, and *initialization* of an array in one statement.
- The syntax for array initializer:

```
elementType [] arrayRefVar = {value0, value1, …., valuek};
```

- Example:

```
double [] myList = {1.9, 2.5, 3.4, 4.5};
```

- Using an array initializer, you have to declare, create, and initialize the array all in one statement.
  - Splitting it would cause a syntax error.

```
double[] myList;
myList = {1.9, 2.9, 3.4, 3.5};  // causes a syntax error
```

Array Basics
Processing Arrays

- When processing array elements, you will often use a *for* loop.
  - All elements in an array are of the same type and they are evenly processed in the same fashion repeatedly using a loop.
  - Since the size of the array is known, it is natural to use a *for* loop.

Array Basics
Processing Arrays (Examples)

- *Displaying arrays*: to print an array, you have to print each element in the array using a loop like the following:

```
For (int i = 0; i < myList.length; i++)
    System.out.print (myList[i] + “ ”);
```

  - *Note:* For an array of the *char[]* type, it can be printed using one print statement

```
char[] city = {‘A', ‘m', ‘m', ‘a', ‘n'};
System.out.println(city);
```

- *Summing all elements:* Use a variable named total to store the sum. Initially total is 0. Add each element in the array to total using a loop like this:

```
double total = 0;
for (int i = 0; i < myList.length; i++) {
        total += myList[i];
}
```

Array Basics
Foreach loops

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

```
for (double e: myList) {
        System.out.println(e);
}
```

  - You can read the code as "for each element *e* in *myList*, do the following."
  - Note that the variable, *e*, must be declared as the same type as the elements in *myList*.
- In general, the syntax for a *foreach* loop is:

```
for (elementType element: arrayRefVar) {
        // Process the element
}
```

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

Note

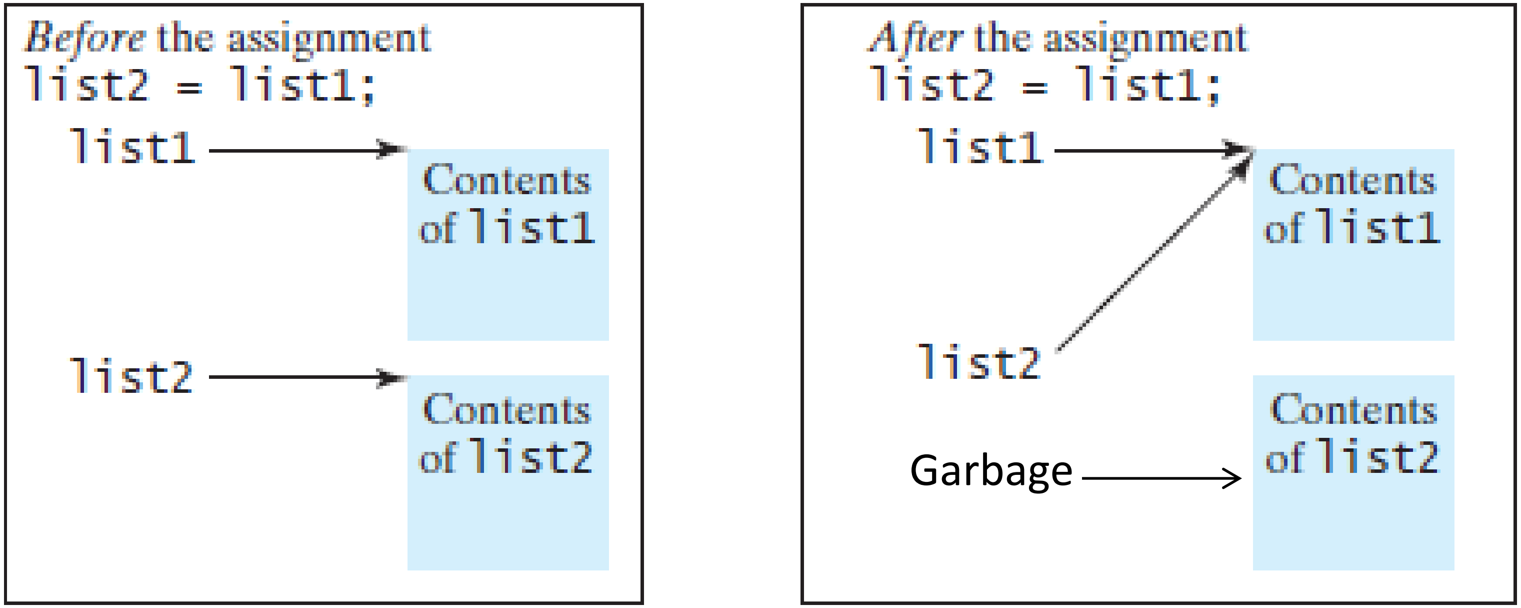
- Accessing an array out of bounds is a common programming error that throws a runtime *ArrayIndexOutOfBoundsException*.
- To avoid it, make sure that you do not use an index beyond **arrayRefVar.length – 1**.
- Programmers often mistakenly reference the first element in an array with index **1**, but it should be **0**.
  - This is called the *off-by-one error*.
- Another common off-by-one error in a loop is using **<=** where **<** should be used.
- For example, the following loop is wrong.

```
for (int i = 0; i <= list.length; i++)
    System.out.print(list[i] + " ");
```

  - The **<=** should be replaced by **<**.

Copying Arrays

- The assignment operator does not copy the contents of an array into another, it instead merely copies the reference values.



Copying Arrays (Cont.)

- To copy the contents of one array into another, you have to copy the *array's individual elements* into the other array.
- Use a loop to copy every element from the source array to the corresponding element in the target array.
- Example:

```
int [] sourceArray = {2, 3, 1, 5, 10};
int [] targetArray = new int [sourceArray.length];
for (int i=0; i < sourceArray.length; i++)
        targetArray [i] = sourceArray [i];
```

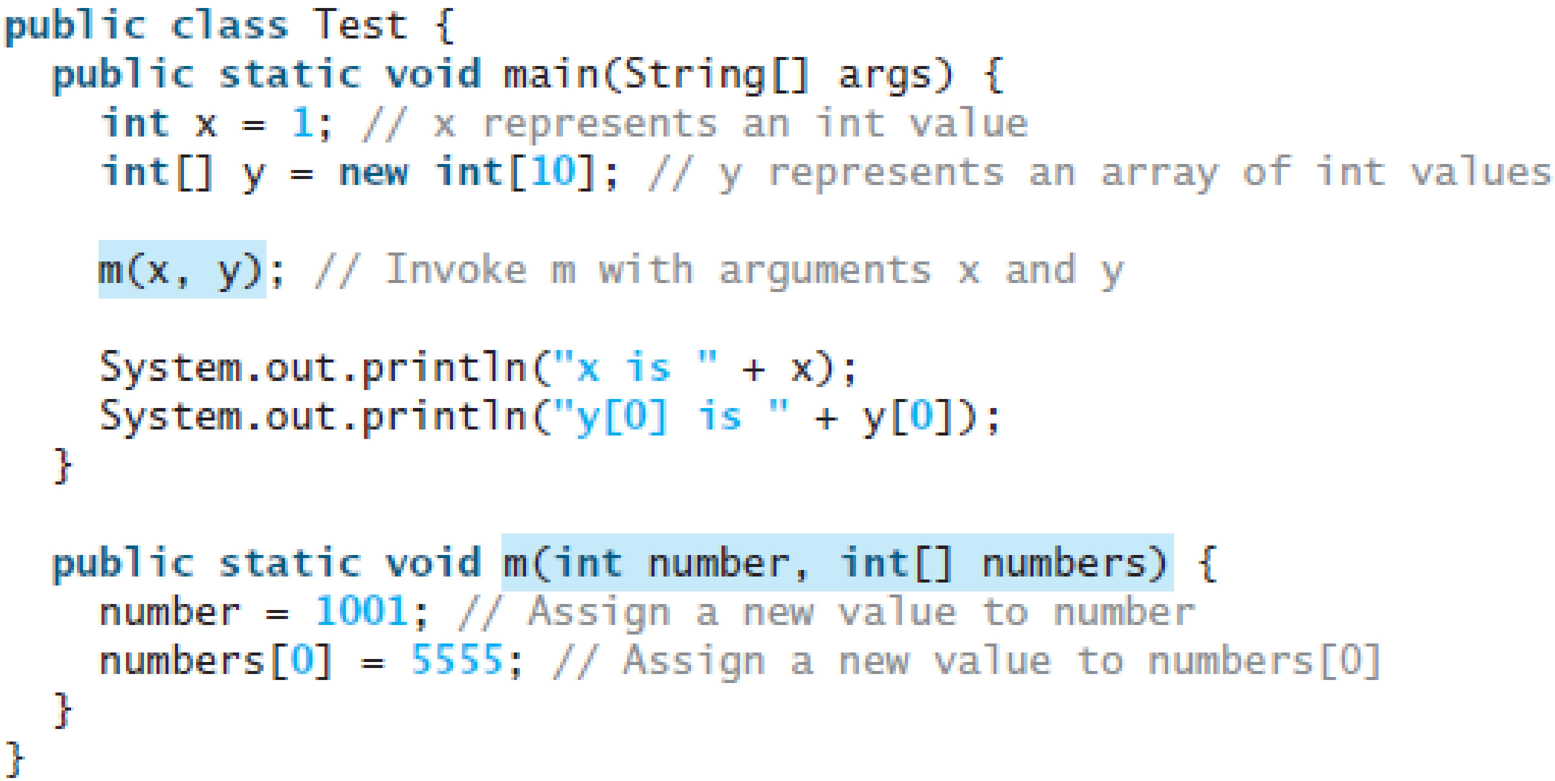
Copying Arrays (Cont.)

- Often, in a program, you need to duplicate an array or a part of an array. In such cases you could attempt to use the assignment statement (**=**), as follows:
  - *list2 = list1;*

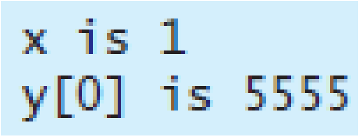
Passing Arrays to Methods

- When passing an array to a method, the *reference* of the array is passed to a method.
- This differs from passing arguments of a primitive type:
  - For an argument of a primitive type, the argument's value is passed.
    - ***The passed variable will not be affected by any change to the value inside the method.***
  - For any argument of an array type, the value of the argument is a reference to an array.
    - ***The passed array will be affected by any change inside the method.***

Passing Arrays to Methods: Example

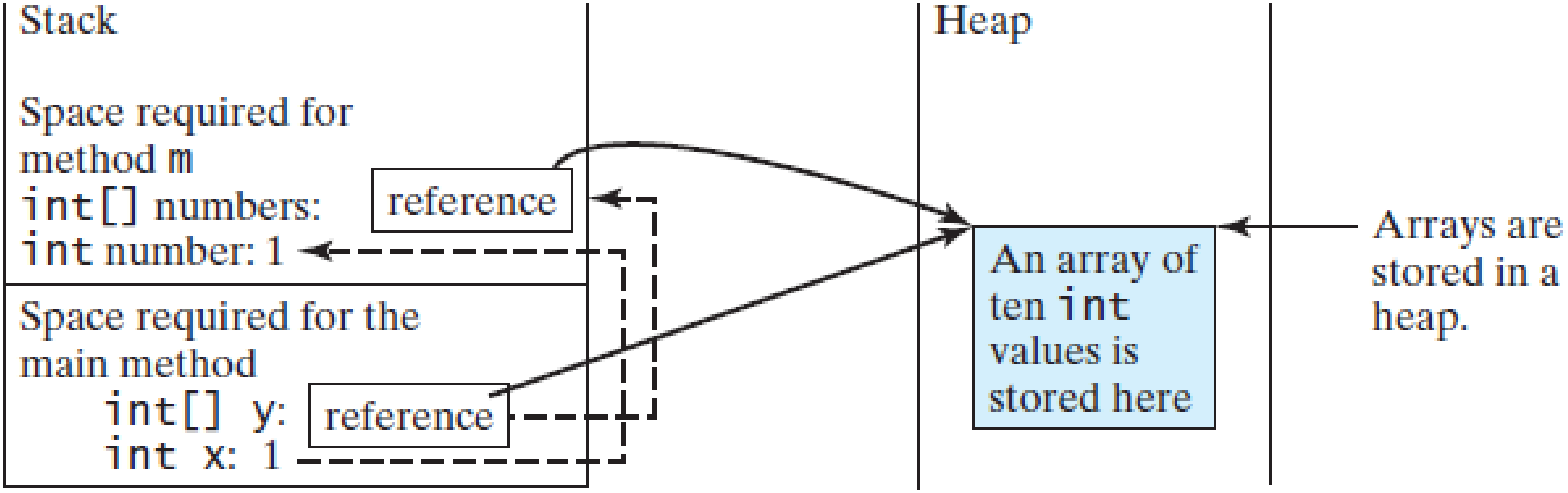


```
public class Test {
  public static void main(String[] args) {
    int x = 1; // x represents an int value
    int[] y = new int[10]; // y represents an array of int values

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

    System.out.println("x is " + x);
    System.out.println("y[0] is " + y[0]);
  }

  public static void m(int number, int[] numbers) {
    number = 1001; // Assign a new value to number
    numbers[0] = 5555; // Assign a new value to numbers[0]
  }
}
```



```
x is 1
y[0] is 5555
```

Passing Arrays to Methods: Example (Cont.)


Stack
Space required for method m
int[] numbers:
reference
int number: 1
Space required for the main method
int[] y:
reference
int x: 1
Heap
An array of ten int values is stored here
Arrays are stored in a heap.

Returning an Array from a Method

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

```
public static int[] copy(int [] list){
      int [] result = new int [list.length];
      for (int i=0; i < list.length; i++)
                result[i]=list[i];
      return result;
}
```

Example of this method invocation:

```
int [] list1 = {1, 2, 3, 4, 5};
int [] list2 = copy(list1);
```

Variable Length Argument List

- A variable number of arguments of the same type can be passed to a method and treated as an array.
- The parameter in the method is declared as follows:

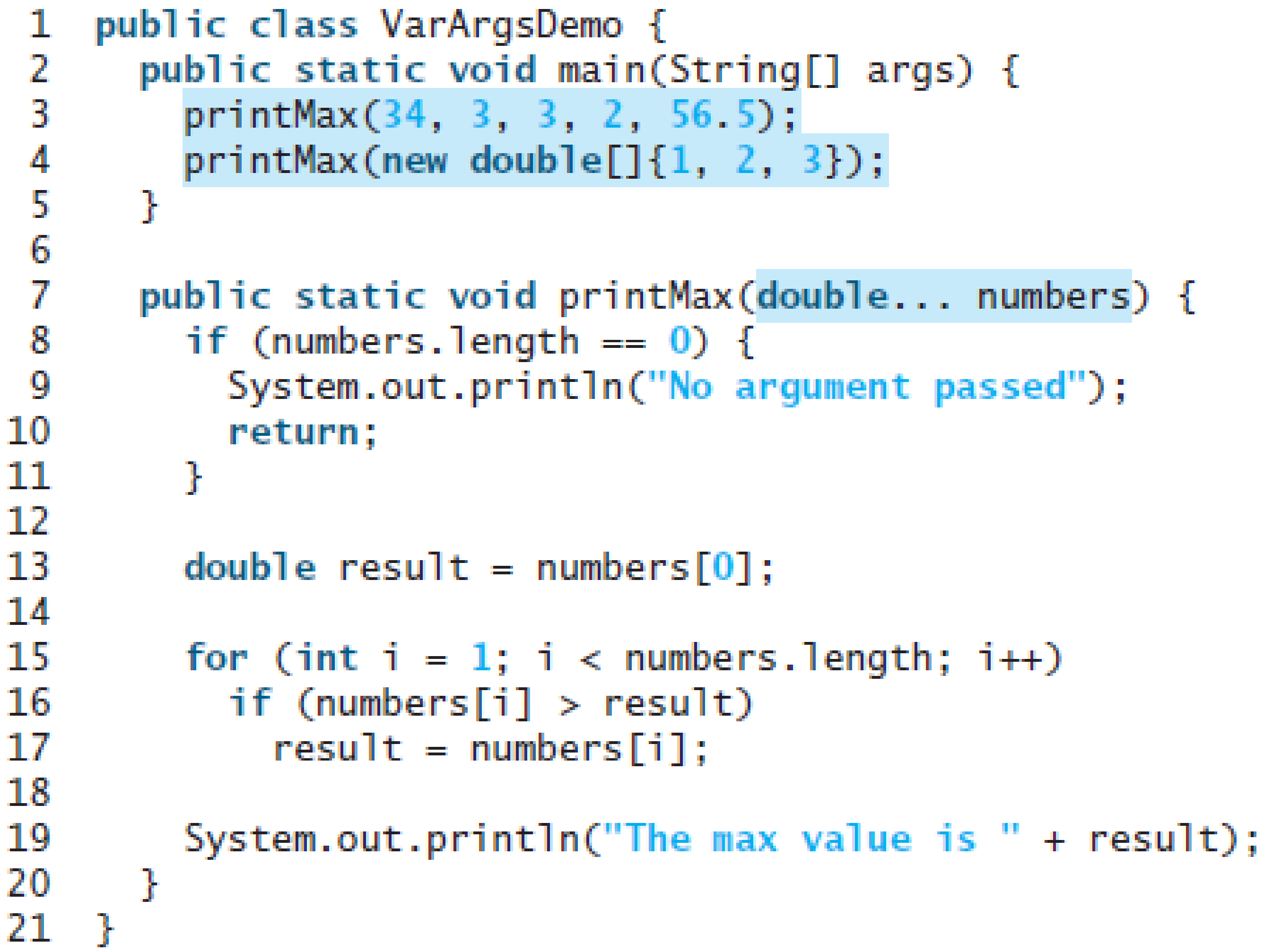
  typeName... parameterName

- In the method declaration, you specify the type followed by an ellipsis (**...**).
- Only one variable-length parameter may be specified in a method, and this parameter must be the last parameter.
  - Any regular parameters must precede it.

Variable Length Argument List (Cont.)

- Java treats a variable-length parameter as an array.
- You can pass an array or a variable number of arguments to a variable-length parameter.
- When invoking a method with a variable number of arguments, Java creates an array and passes the arguments to it.

VarArgsDemo.java



```java
public class VarArgsDemo {
  public static void main(String[] args) {
    printMax(34, 3, 3, 2, 56.5);
    printMax(new double[]{1, 2, 3});
  }

  public static void printMax(double... numbers) {
    if (numbers.length == 0) {
      System.out.println("No argument passed");
      return;
    }

    double result = numbers[0];

    for (int i = 1; i < numbers.length; i++)
      if (numbers[i] > result)
        result = numbers[i];

    System.out.println("The max value is " + result);
  }
}
```

Command-Line Arguments

- The *main* method has the parameter *args* of *String[]* type.
  - It is clear that *args* is an array of strings.
- You can pass strings to a main method from the command line when you run the program.
- The following command line, for example, starts a program named TestMain with three strings: arg0, arg1, and arg2:
  - *java TestMain arg0 arg1 arg2*
  - They don't have to appear in double quotes on the command line.
  - The strings are separated by a space.
- A string that contains a space must be enclosed in double quotes.
  - *java TestMain "First num" alpha 53*
  - Note that 53 is actually treated as a string.

Command-Line Arguments (Cont.)

- When the *main* method is invoked, the Java interpreter creates an array to hold the command-line arguments and pass the array reference to *args*.
- For example, if you invoke a program with *n* arguments, the Java interpreter creates an array like this one:
  - *args = new String[n];*

Two-Dimensional Arrays

- Two dimensional arrays are used to represent data in a matrix or a table.
- The syntax for declaring and creating two dimensional arrays is:

elementType [] [] arrayRefVar;

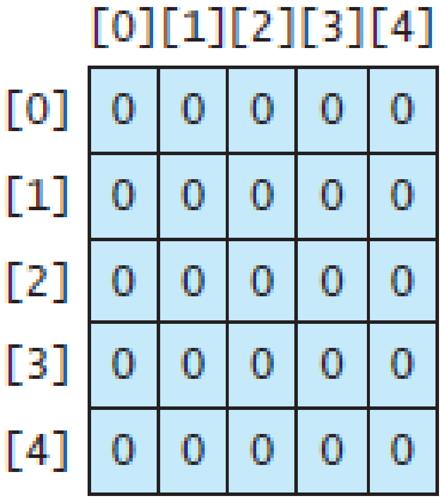
arrayRefVar = new elementType [numRows][numCols];

- An element in a two-dimensional array is accessed through a row and column index:

arrayRefVar [rowIndex][colIndex];

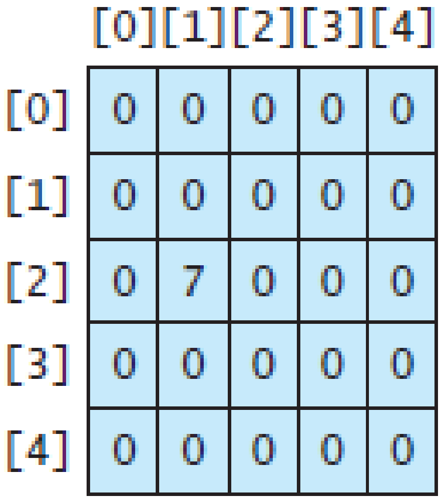
Two-Dimensional Arrays: Examples

int [][] matrix;



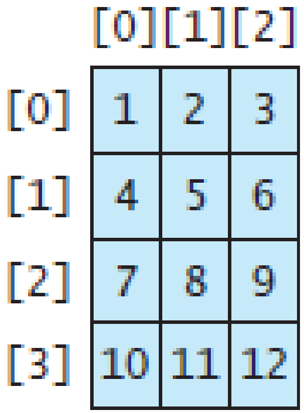
	[0]	[1]	[2]	[3]	[4]
[0]	0	0	0	0	0
[1]	0	0	0	0	0
[2]	0	0	0	0	0
[3]	0	0	0	0	0
[4]	0	0	0	0	0

```
matrix = new int[5][5];
```



	[0]	[1]	[2]	[3]	[4]
[0]	0	0	0	0	0
[1]	0	0	0	0	0
[2]	0	7	0	0	0
[3]	0	0	0	0	0
[4]	0	0	0	0	0

```
matrix[2][1] = 7;
```

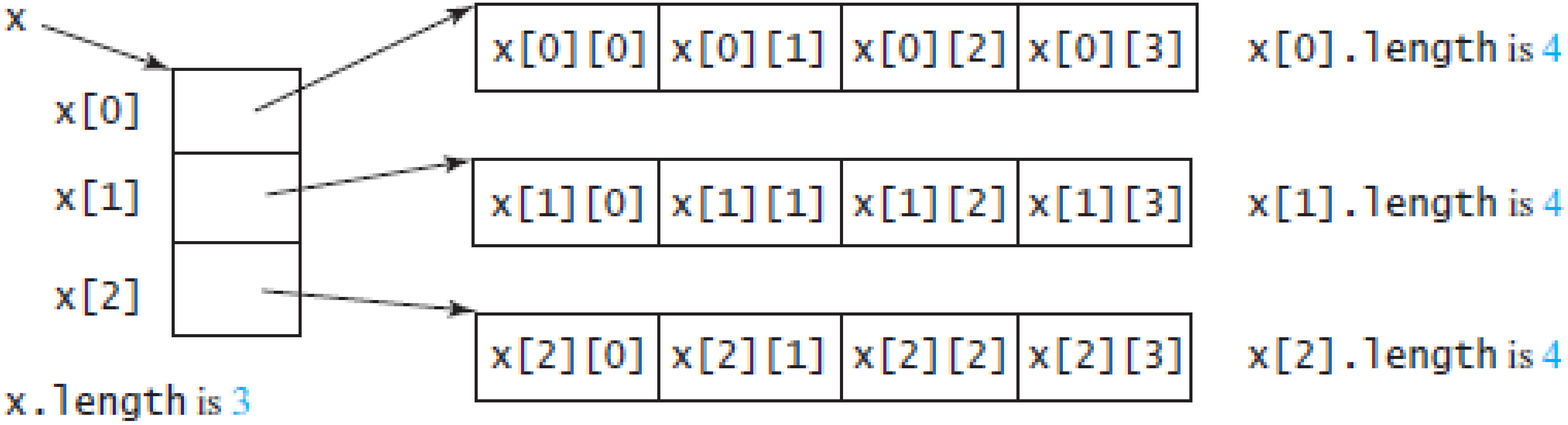


	[0]	[1]	[2]
[0]	1	2	3
[1]	4	5	6
[2]	7	8	9
[3]	10	11	12

```
int[][] array = {
  {1, 2, 3},
  {4, 5, 6},
  {7, 8, 9},
  {10, 11, 12}
};
```

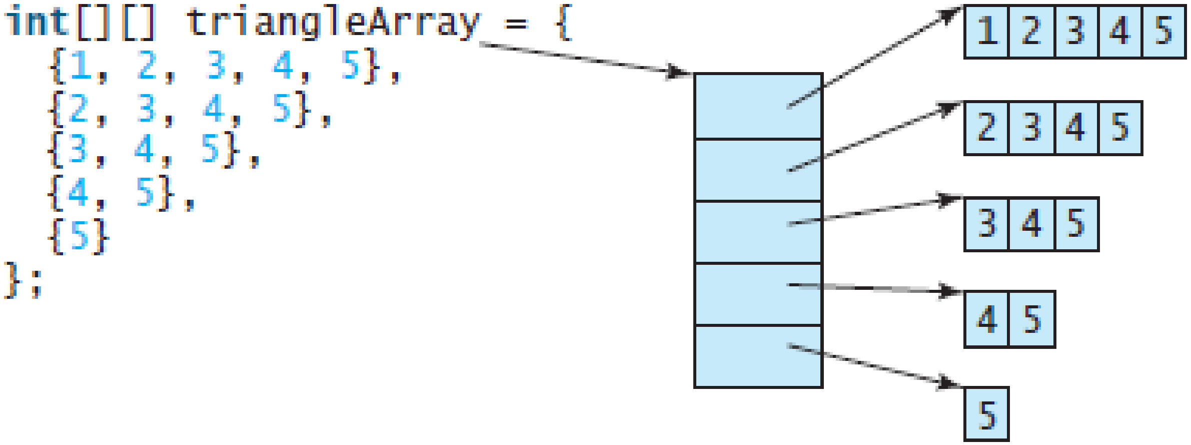
Two-Dimensional Arrays (Cont.)

- A two-dimensional array is actually an array in which each element is a one-dimensional array.



Ragged Arrays

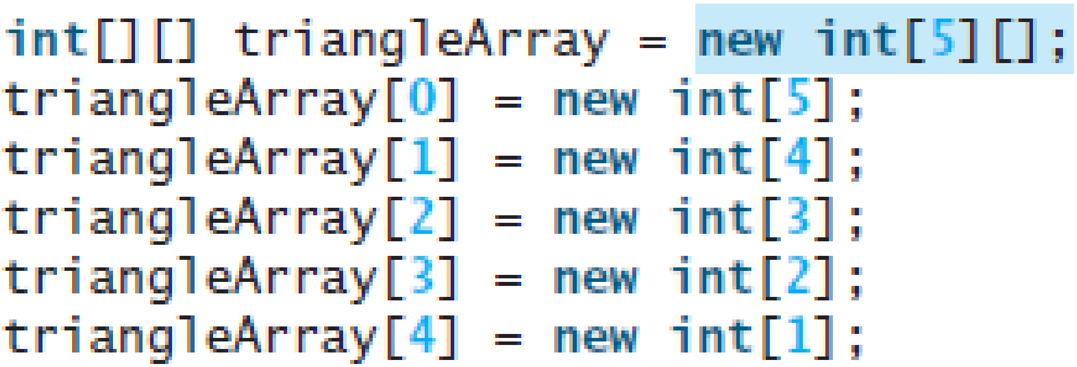
- Each row in a two-dimensional array is itself an array.
- Thus, the rows can have different lengths.
  - An array of this kind is known as a *ragged array*.



```
int[][] triangleArray = {
  {1, 2, 3, 4, 5},
  {2, 3, 4, 5},
  {3, 4, 5},
  {4, 5},
  {5}
};
```

Ragged Arrays (Cont.)

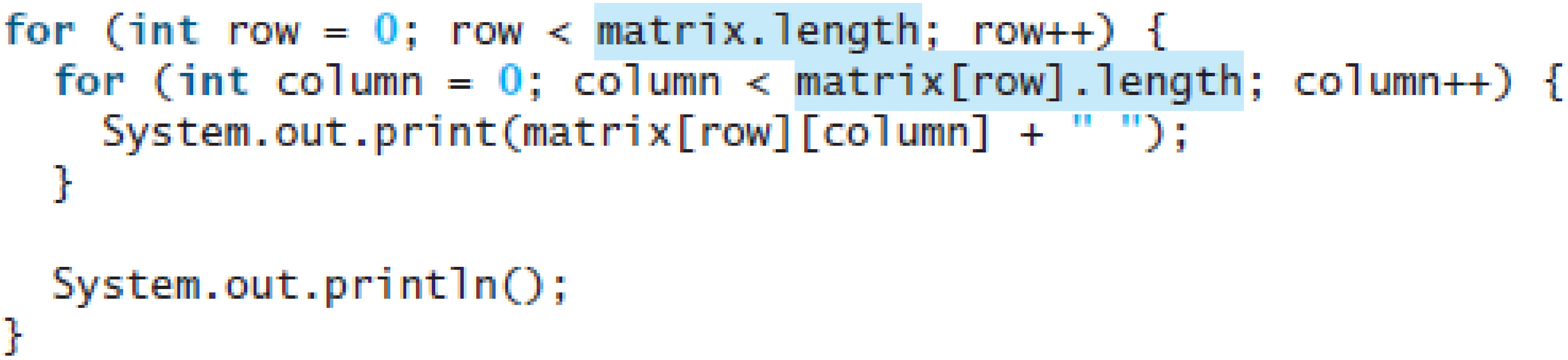
- You can also create a two-dimensional array if you know the sizes of its rows but do not know the values, using the following format:



```
int[][] triangleArray = new int[5][];
triangleArray[0] = new int[5];
triangleArray[1] = new int[4];
triangleArray[2] = new int[3];
triangleArray[3] = new int[2];
triangleArray[4] = new int[1];
```

Processing Two-Dimensional Arrays

- Nested for loops are often used to process a two-dimensional array
- Suppose an array *matrix* is created as follows:
  - *int[][] matrix = new int[10][10];*
- To print the elements of the array *matrix* each row on a line:



```
for (int row = 0; row < matrix.length; row++) {
  for (int column = 0; column < matrix[row].length; column++) {
    System.out.print(matrix[row][column] + " ");
  }

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

Object-Oriented Problem Solving

Objects & Classes (Part I)

Based on Chapter 9 of "Introduction to Java Programming" by Y. Daniel Liang.

Eng. Asma Abdel Karim
Computer Engineering Department

Outline

- Defining Classes for Objects (9.2)
- Declaring and Creating Objects Reference Variables(9.5.1)
- Accessing an object’s members (9.5.2)
- Example: TestCircle.java.
- Constructing objects using constructors (9.4)
- Reference data fields and the null value (9.5.3)
- Difference between variables of primitive types and reference types. (9.5.4)
- UML class diagrams.

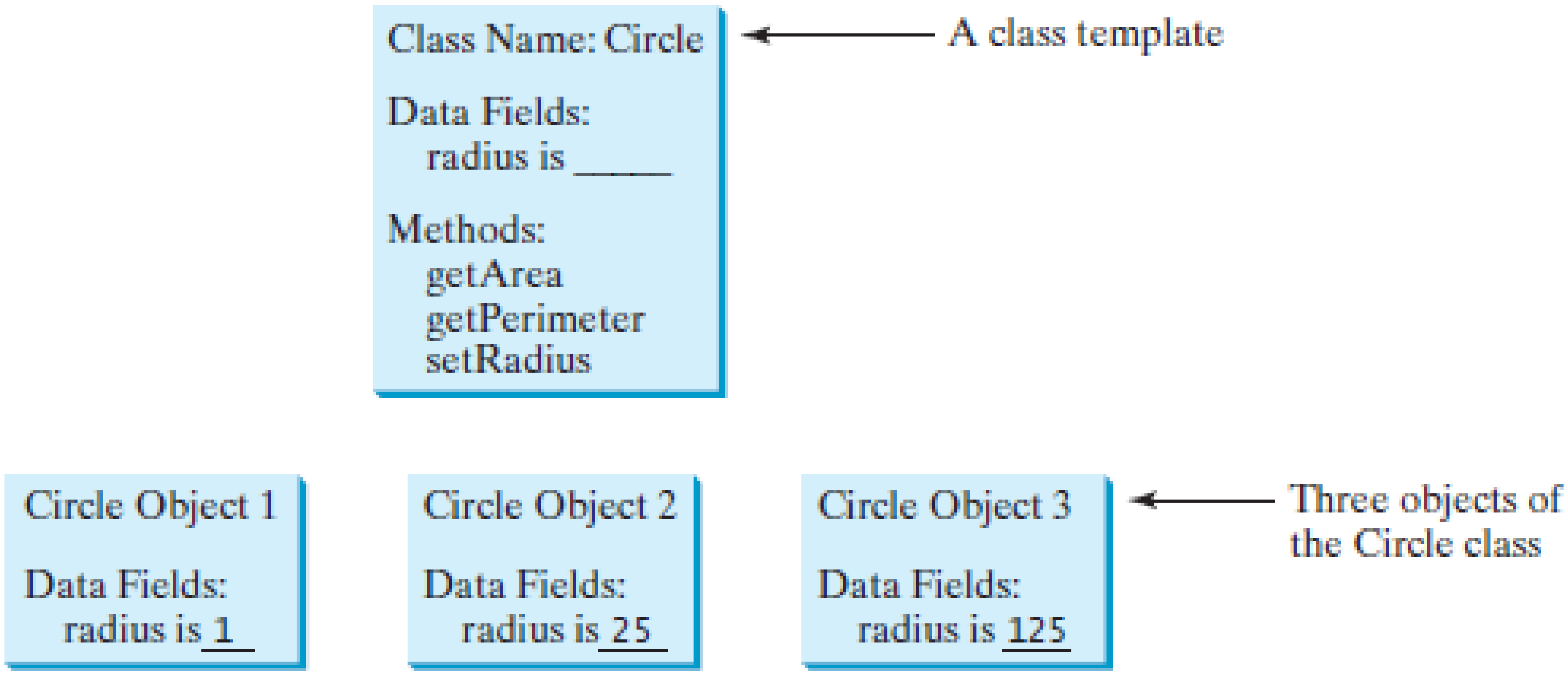
Defining Classes for Objects
What is an Object?

- *Object-oriented programming (OOP)* involves programming using *objects*.
- An *object* represents an entity that can be distinctly identified.
- An object has a unique:
  - *Identity*
  - *State*
    - Also known as its properties or attributes.
    - Represented by data fields with their current values.
  - *Behavior*
    - Also known as its actions.
    - Defined by methods: to invoke a method on an object is to ask the object to perform an action.

Defining Classes for Objects What is a Class?

- A *class* is a *template*, *blue-print*, or *contract* that defines what an objects data fields and methods will be.
- An object is an instance of a class.
  - You can create many instances of a class.
  - Creating an instance is referred to as *instantiation*.
  - The terms *object* and *instance* are often interchangeable.
- Objects of the same type are defined using a common class.
- A *Java class* uses:
  - Variables to define data fields, and
  - Methods to define actions.

Defining Classes for Objects



Class Name: Circle

Data Fields:
radius is _____

Methods:
getArea
getPerimeter
setRadius

A class template

Circle Object 1

Data Fields:
radius is 1

Circle Object 2

Data Fields:
radius is 25

Circle Object 3

Data Fields:
radius is 125

Three objects of the Circle class

Defining Classes for Objects Example: Circle Class

```
class Circle{
    double radius;
    void setRadius (double newRadius){
        radius = newRadius;
    }
}
```

Class Circle has one variable of type double called radius.

Class Circle has one void method called setRadius which takes one double parameter and assigns it to the variable radius.

- The **Circle** class is different from all of the other classes you have seen thus far.
  - It does not have a **main** method and therefore cannot be run; it is merely a definition for circle objects.

Declaring and Creating Objects Reference Variables

- A class is essentially a *programmer-defined* type.
- Objects are accessed via the object's *reference variables*, which contain references to the objects.
- The syntax to *declare* an object reference variable is:

  ClassName objectRefVar;
- Example:

  Circle myCircle;
- A class is a *reference type*: a variable of the class type can reference an instance of the class.

Declaring and Creating Objects Reference Variables (Cont.)

- To *create* an object and assign its reference to a declared object reference variable:

  objectRefVar = new ClassName ();

- Example:

  myCircle = new Circle();

- The variable *myCircle* holds a reference to a *Circle* object.
  - An object reference variable that appears to hold an object actually contains a reference to that object.

Declaring and Creating Objects Reference Variables (Cont.)

- A single statement can be used to combine

1) the declaration of an object reference variable,
2) the creation of an object, and
3) the assigning of an object reference to the variable as follows:

ClassName objectRefVar = new ClassName();

- Example:

Circle myCircle = new Circle ();

Accessing an Object's Members

- In OOP, object's members are its *data fields* and *methods*.
- An object's data can be accessed and its methods invoked using the *dot operator (.)*.
  - Also known as the *object member access operator*:
- To reference a data field in an object:
  - *objectRefVar.dataField*
- Example:
  - *myCircle.radius*
- To invoke a method on an object:
  - *objectRefVar.method(arguments)*
- Example:
  - *myCircle.setRadius(5);*

Example: TestCircle.java

```
public class TestCircle{
    public static void main (String [] args){
            Circle circle1 = new Circle ();
            circle1.setRadius(5);
            System.out.println(“The radius of circle-1 is ”+circle1.radius);
            Circle circle2 = new Circle();
            circle2.radius = 1;
            System.out.println(“The area of circle-2 is ”+circle2.getArea());
    }
}
class Circle{
    double radius;
    void setRadius (double newRadius){
            radius = newRadius;
    }
   double getArea(){
            return radius*radius*22.0/7.0;
    }
}
```

The public class must have the same name as the file name.

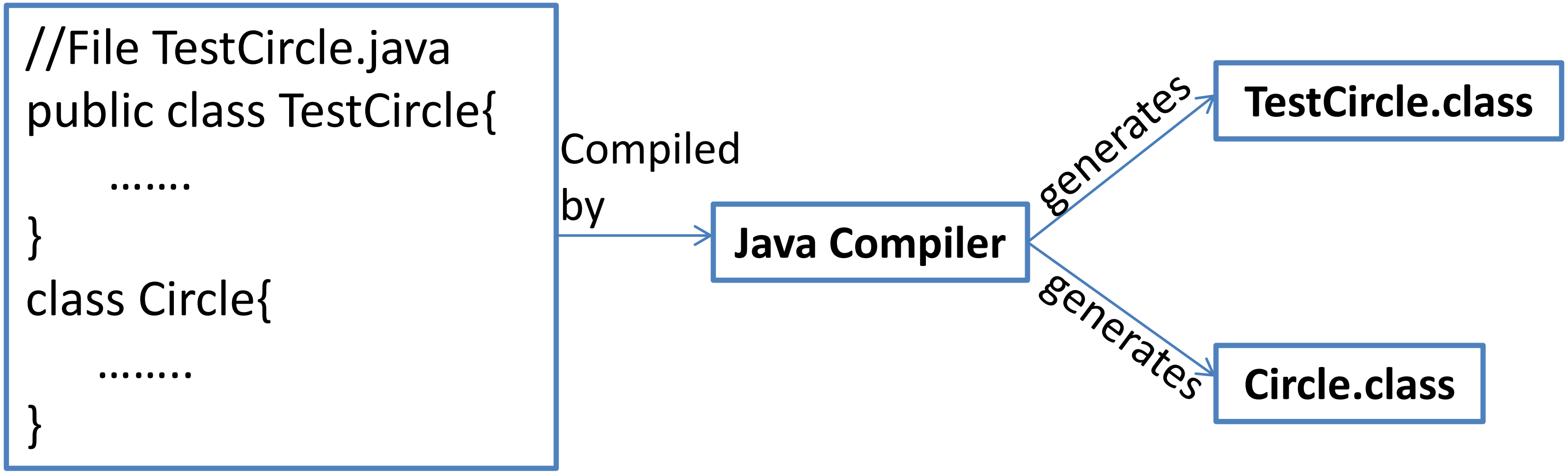
Only one class in a file can be a public class.

Example: TestCircle.java (Cont.)

```
public class TestCircle{
     public static void main (String [] args){
          Circle circle1 = new Circle ();
          circle1.setRadius(5);
          System.out.println(“The radius of circle-1 is ”+circle1.radius);
          Circle circle2 = new Circle();
          circle2.radius = 1;
          System.out.println(“The area of circle-2 is ”+circle2.getArea());
     }
}
class Circle{
     double radius;
     void setRadius (double newRadius){
          radius = newRadius;
     }
    double getArea(){
          return radius*radius*22.0/7.0;
     }
}
```

Remember: this is where the program starts execution.

Example: TestCircle.java (Cont.)



```
//File TestCircle.java
public class TestCircle{
      …….
}
class Circle{
      ……..
}
```

Compiled by

Java Compiler

generates

TestCircle.class

generates

Circle.class

Constructing Objects Using Constructors

- A *constructor* is invoked to create an object using the *new* operator.
- *Constructors* are a special kind of method.
- They have three peculiarities:
  - A constructor must have the same name as the class itself.
  - Constructors do not have a return type.
    - Not even *void*.
  - Constructors are invoked using the new operator when an object is created.
    - They play the role of initializing objects.

Constructing Objects Using Constructors (Cont.)

- A class may be defined without constructors.
- In this case, a *default constructor* is provided automatically:
  - A *default constructor* is a *public no-argument* constructor with an empty body which is implicitly defined in the class.
  - A *default constructor* is provided only if there are no other constructors explicitly defined in the class.

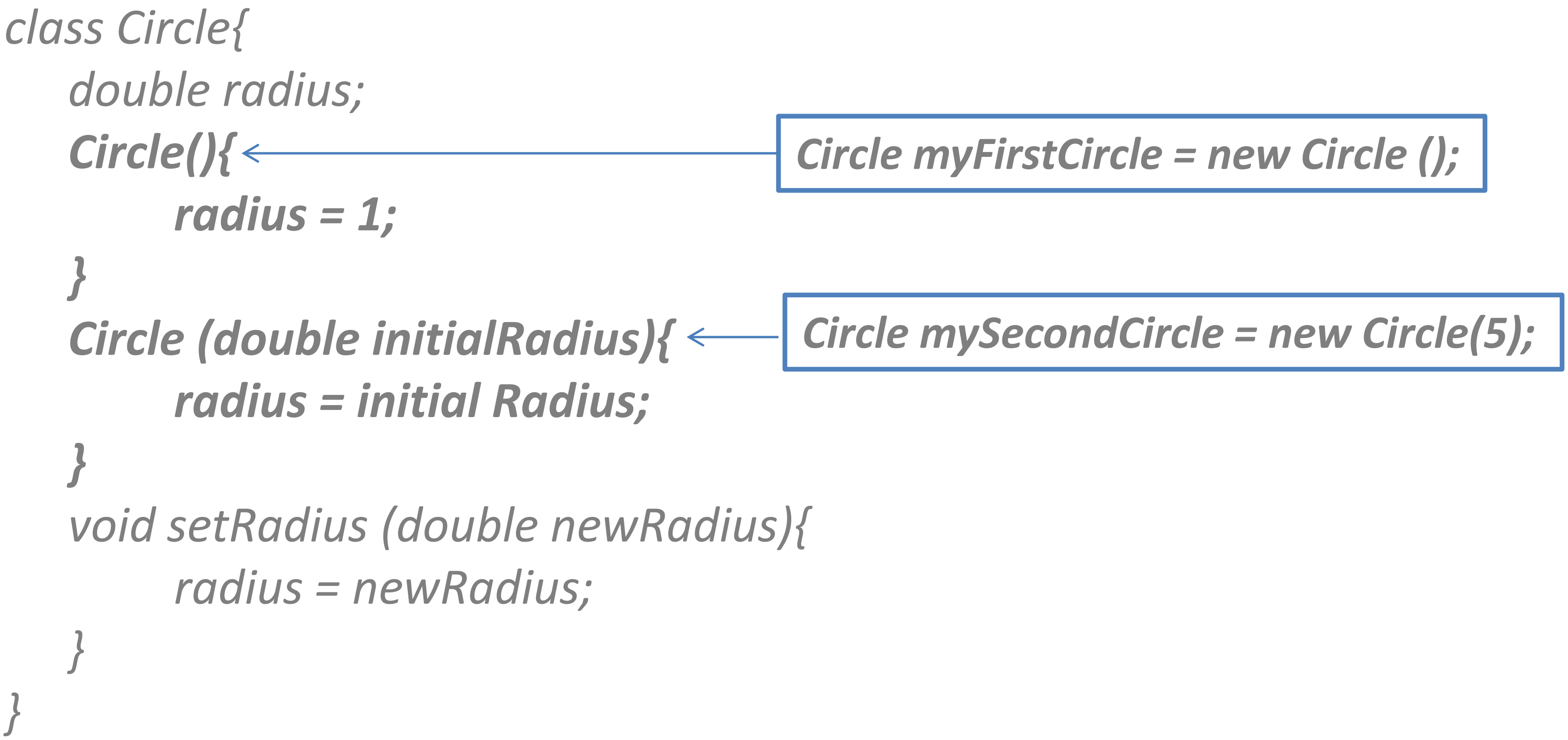
Constructing Objects Using Constructors (Cont.)

- The constructor has exactly the same name as its defining class.
- To construct an object from a class, invoke a constructor of the class using the *new* operator, as follows:
  - *new ClassName(arguments);*
- Like regular methods, constructors can be overloaded.
  - Multiple constructors can have the same name but different signatures.
  - Makes it easy to construct objects with different initial data values.

Example TestCircle.java Revisited

```
public class TestCircle{
    public static void main (String [] args){
        Circle circle1 = new Circle (5);
        System.out.println(“The radius of this circle is   “+circle1.radius);
    }
}
class Circle{
    double radius;
    Circle (double initialRadius){
        radius = initialRadius;
    }
    void setRadius (double newRadius){
        radius = newRadius;
    }
}
```

Constructors Overloading



```
class Circle{
    double radius;
    Circle(){
        radius = 1;
    }
    Circle (double initialRadius){
        radius = initial Radius;
    }
    void setRadius (double newRadius){
        radius = newRadius;
    }
}
```

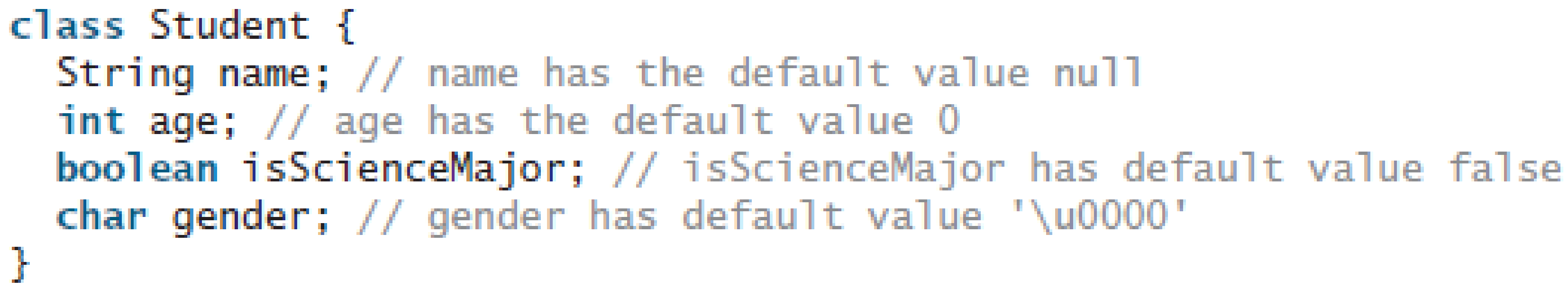
Circle myFirstCircle = new Circle ();

Circle mySecondCircle = new Circle(5);

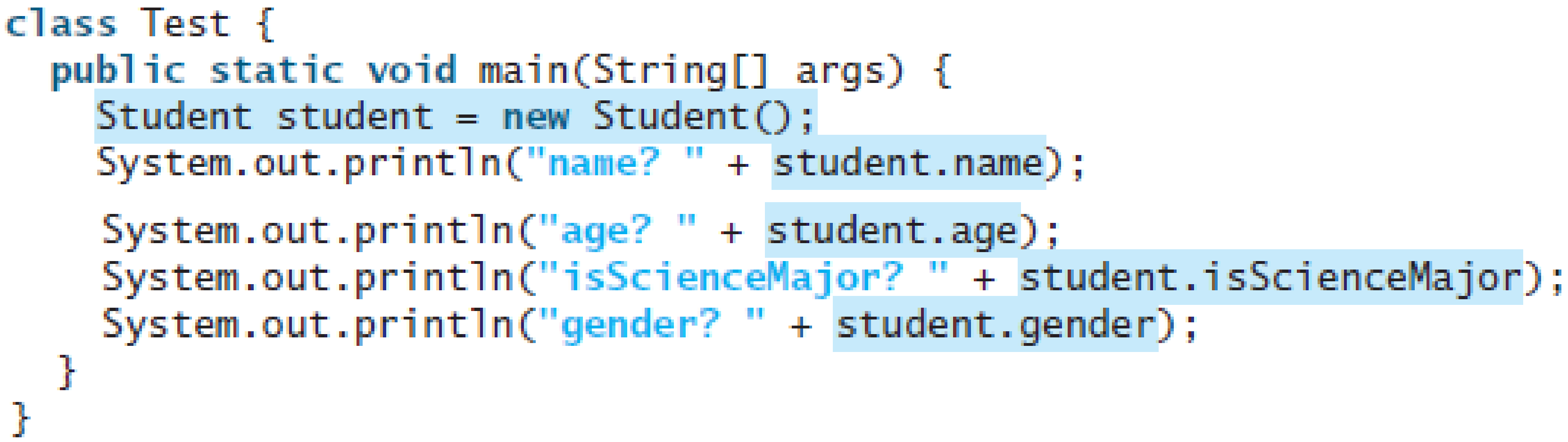
Reference Data Fields and the *null* Value

- Java assigns default values to data fields when an object is created.
  - *0* for *numeric* type.
  - *false* for a *boolean* type.
  - *\u0000* for a *char* type.
  - *Null* for a *reference* type.
    - *Null* is a special literal used for reference types.
- *NullPointerException* is a common runtime error. It occurs when you invoke a method on a reference variable with a *null* value.
- However, Java assigns no default value to a local variable inside a method.

Reference Data Fields and the *null* Value (Cont.)



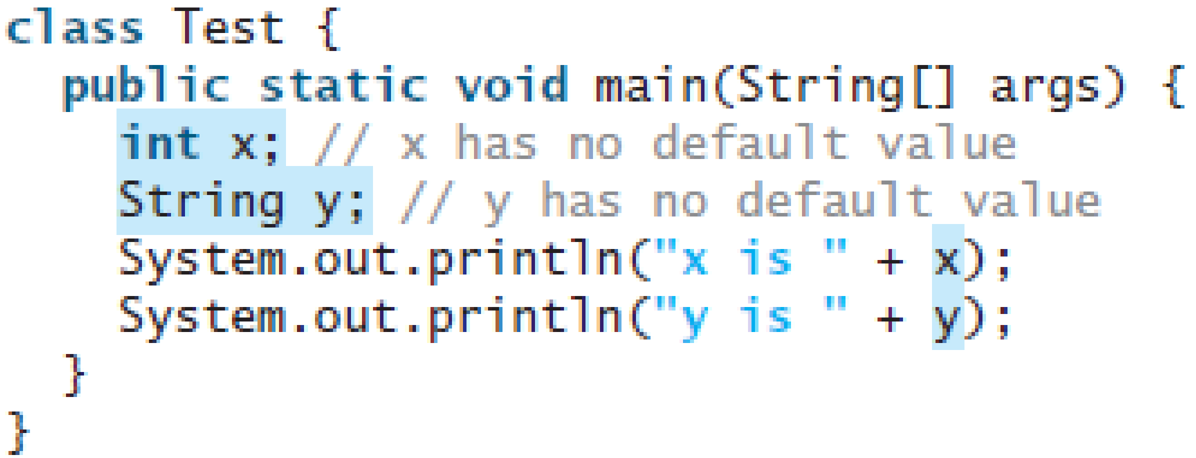
```
class Student {
  String name; // name has the default value null
  int age; // age has the default value 0
  boolean isScienceMajor; // isScienceMajor has default value false
  char gender; // gender has default value '\u0000'
}
```



```
class Test {
  public static void main(String[] args) {
    Student student = new Student();
    System.out.println("name? " + student.name);
    System.out.println("age? " + student.age);
    System.out.println("isScienceMajor? " + student.isScienceMajor);
    System.out.println("gender? " + student.gender);
  }
}
```

Reference Data Fields and the *null* Value

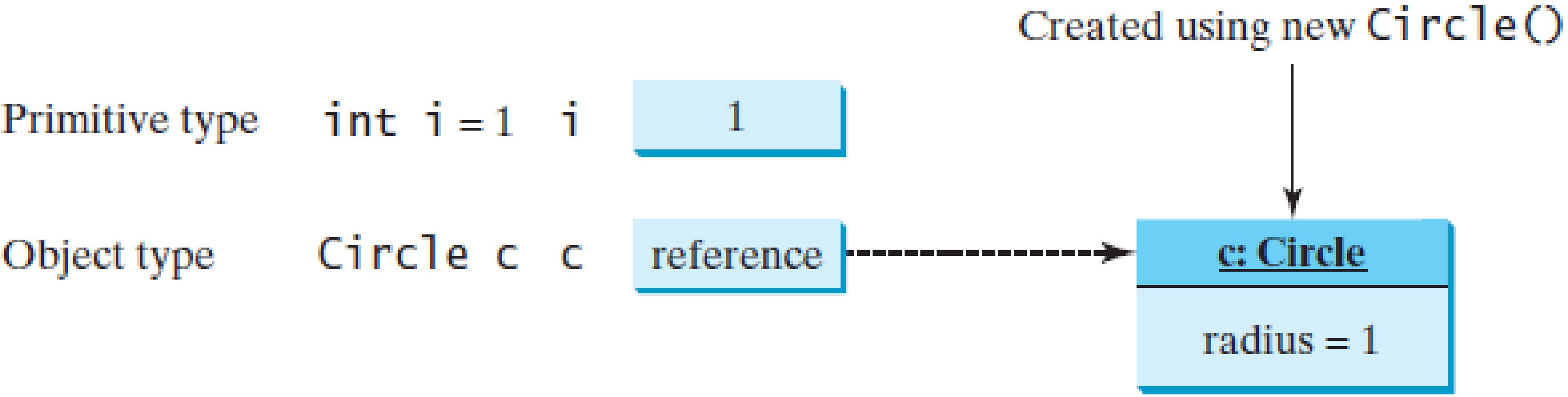
- The following code has a compile error, because the local variables *x* and *y* are not initialized:



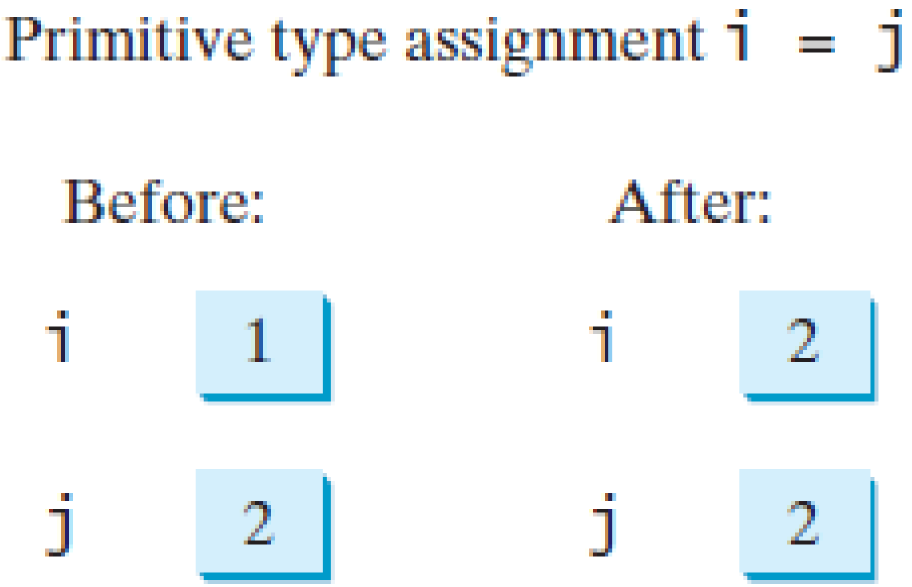
```
class Test {
  public static void main(String[] args) {
    int x; // x has no default value
    String y; // y has no default value
    System.out.println("x is " + x);
    System.out.println("y is " + y);
  }
}
```

Difference between Variables of Primitive Types and Reference Types

- Every variable represents a memory location that holds a value.
- A variable of a primitive type holds a value of the primitive type, and a variable of a reference type holds a reference to where an array or object is stored in memory.



Difference between Variables of Primitive Types and Reference Types (Cont.)



Primitive type assignment i = j

Before:

i 1

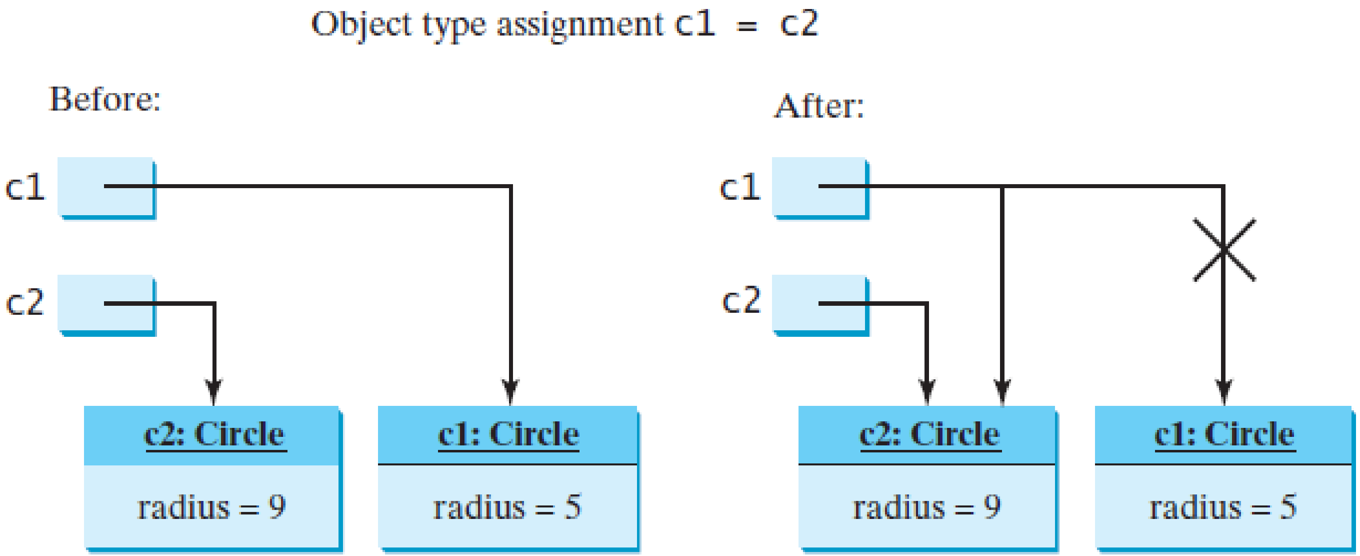
j 2

After:

i 2

j 2

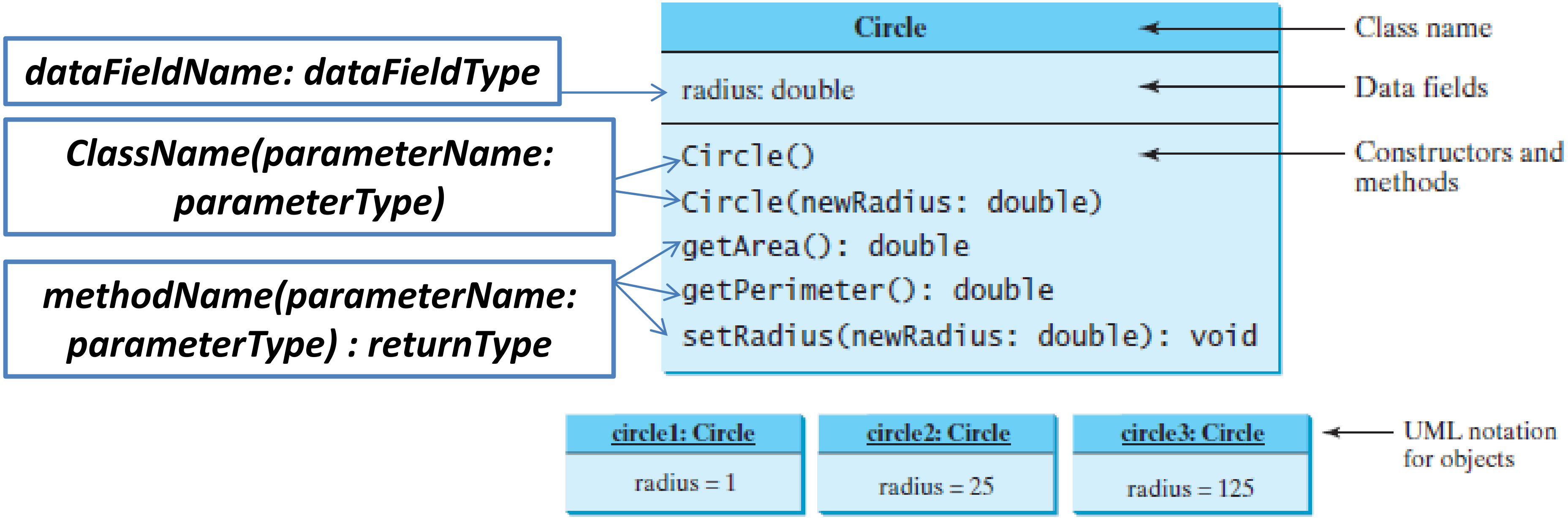
Difference between Variables of Primitive Types and Reference Types (Cont.)



- <u>After assignment:</u>
  - *c1* points to the same object referenced by *c2*.
  - The object previously referenced by *c1* is no longer useful and therefore is now known as *garbage*.
  - Garbage occupies memory space, so the Java runtime system detects garbage and automatically reclaims the space it occupies. This process is called *garbage collection*.

UML Class Diagrams

- A standardized notation to illustrate classes and objects is the *Unified Modeling Language (UML)* class diagram.



dataFieldName: dataFieldType

ClassName(parameterName: parameterType)

methodName(parameterName: parameterType) : returnType

Object-Oriented Problem Solving

Objects & Classes (Part II)

Based on Chapter 9 of "Introduction to Java Programming" by Y. Daniel Liang.

Eng. Asma Abdel Karim
Computer Engineering Department

Outline

- Static Variables, Constants, and Methods (9.7)
- Visibility Modifiers (9.8)
- Data Field Encapsulation (9.9)
- Passing Objects to Methods (9.10)
- The Scope of Variables (9.13)

Static Variables, Constants, and Methods

- All variables declared in the data fields of the previous examples are called *instance variables*.
- An *instance variable* is tied to a specific instance of the class.
  - It is not shared among objects of the same class.
  - It has independent memory storage for each instance.
- In the following example, the *radius* of the first object "*circle1*" is independent of the *radius* of the second object "*circle2*":

```
Circle circle1 = new Circle();
Circle circle2 = new Circle(5);
```

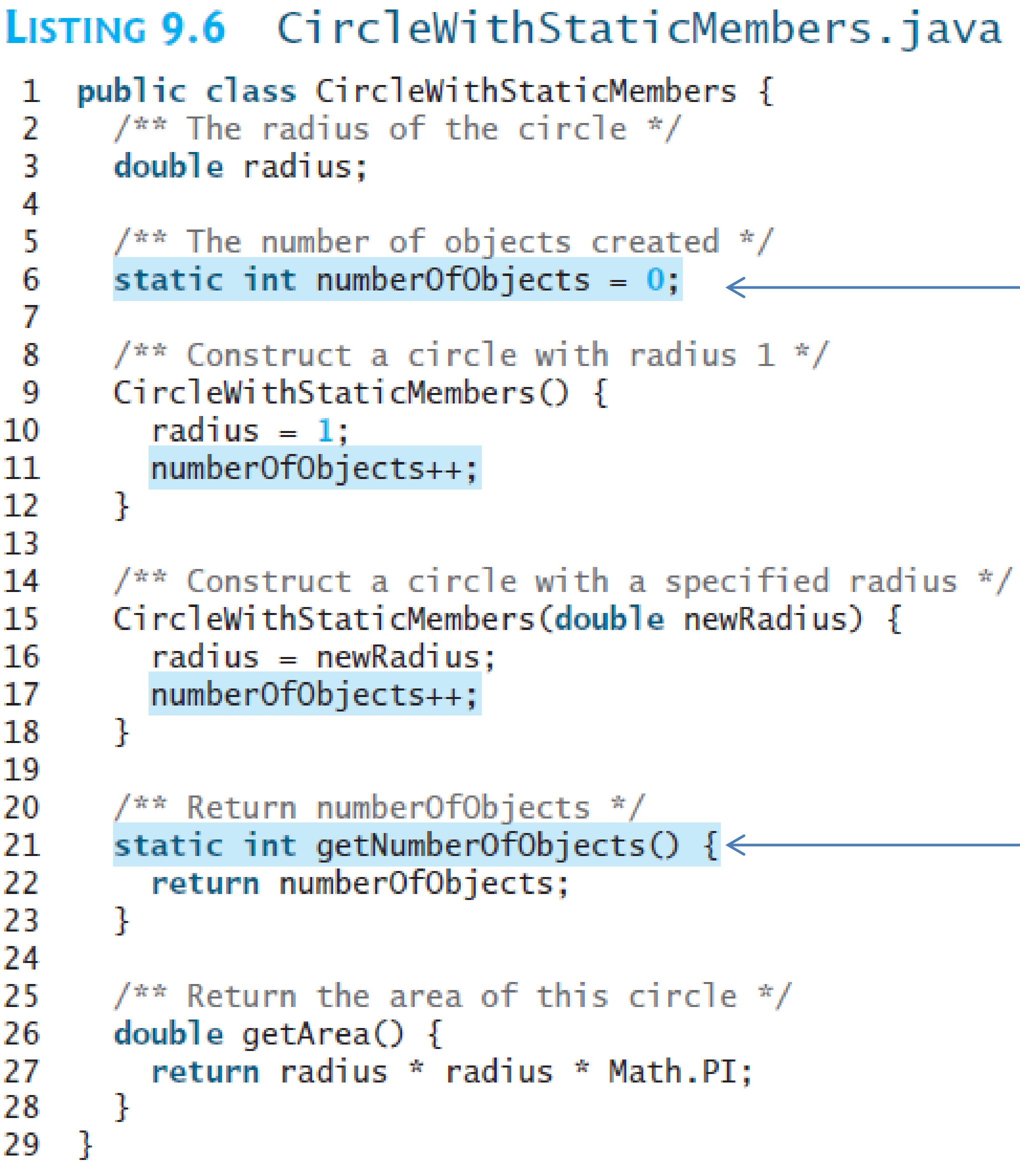
Static Variables, Constants, and Methods (Cont.)

- *Static variables*, also known as *class variables*, store values for the variables in a common memory location.
  - A *static variable* is used when it is wanted that all instances of the class to share data.
  - If one instance of the class changes the value of a static variables, all instances of the same class are affected.
- Static methods can be called without creating an instance of the class.

Static Variables, Constants, and Methods (Cont.)

- To declare a static variable or define a static method, put the modifier *static* in the variable or method declaration.
- Since constants in a class are shared by all objects of the class, they should be declared static.
  - *final static double PI = 3.14159265358979323846;*
- Static variables and methods can be accessed from a reference variable or from their class name.

Example

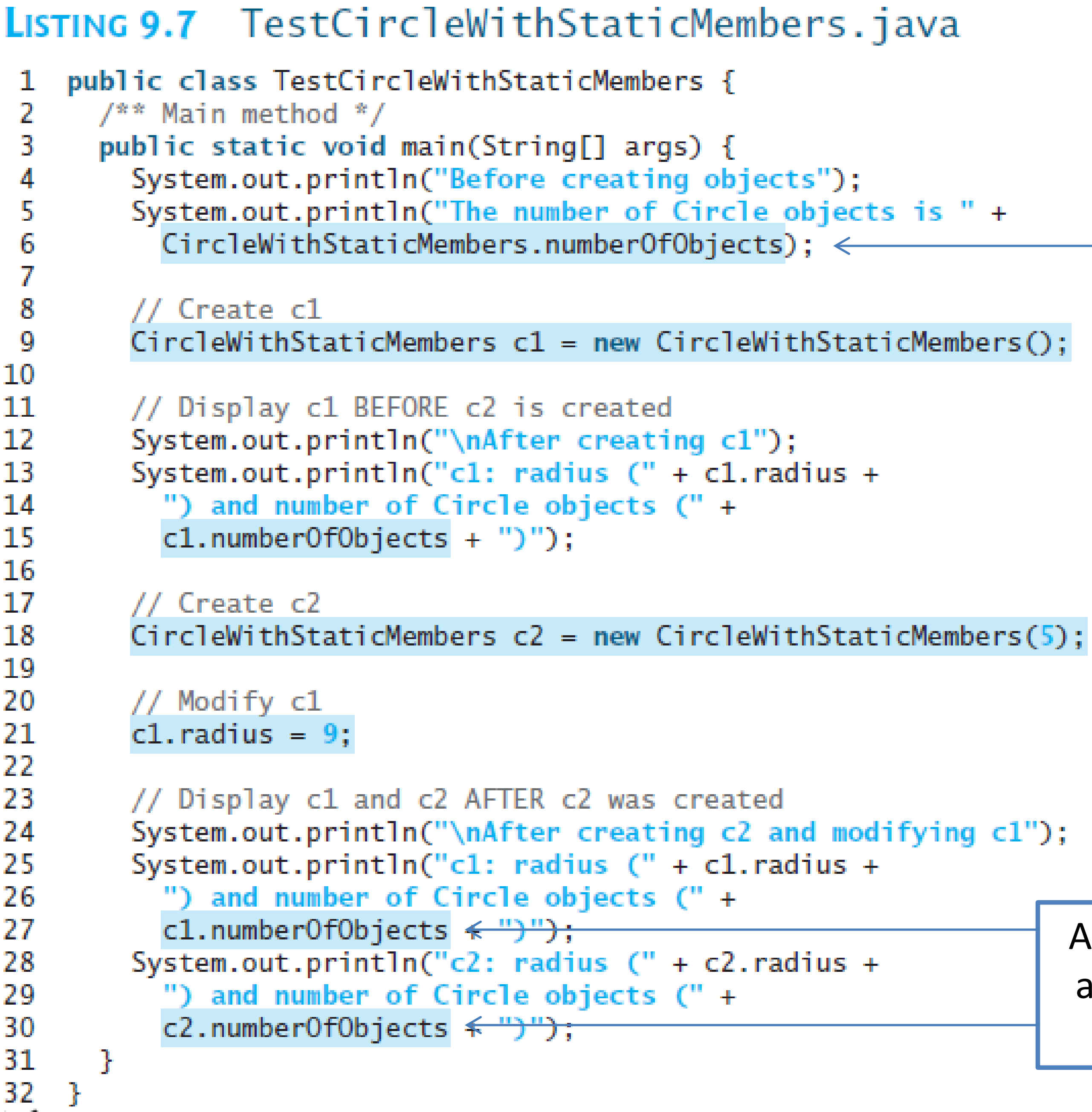


LISTING 9.6 CircleWithStaticMembers.java

```java
public class CircleWithStaticMembers {
  /** The radius of the circle */
  double radius;

  /** The number of objects created */
  static int numberOfObjects = 0;

  /** Construct a circle with radius 1 */
  CircleWithStaticMembers() {
    radius = 1;
    numberOfObjects++;
  }

  /** Construct a circle with a specified radius */
  CircleWithStaticMembers(double newRadius) {
    radius = newRadius;
    numberOfObjects++;
  }

  /** Return numberOfObjects */
  static int getNumberOfObjects() {
    return numberOfObjects;
  }

  /** Return the area of this circle */
  double getArea() {
    return radius * radius * Math.PI;
  }
}
```

static variable

static method



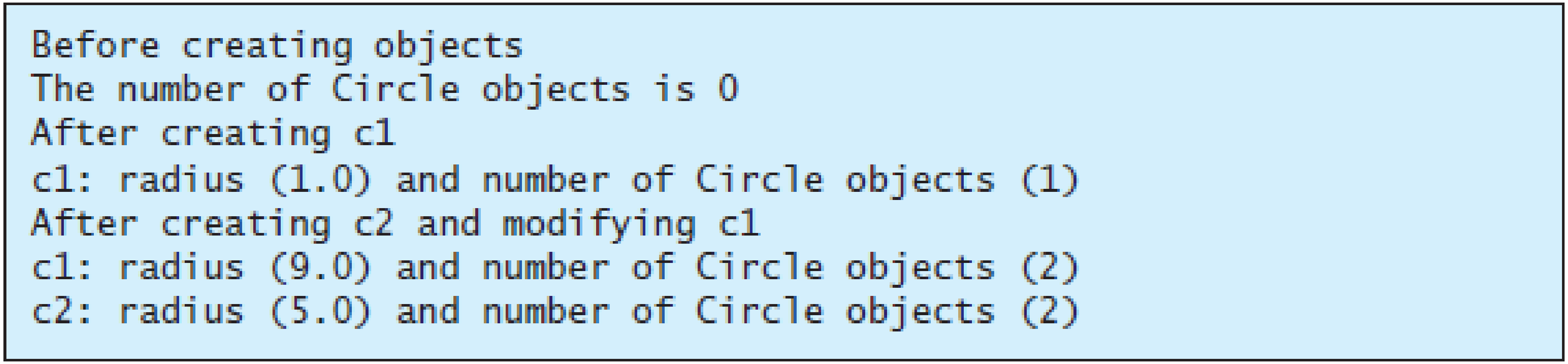
LISTING 9.7 TestCircleWithStaticMembers.java

```java
public class TestCircleWithStaticMembers {
  /** Main method */
  public static void main(String[] args) {
    System.out.println("Before creating objects");
    System.out.println("The number of Circle objects is " +
      CircleWithStaticMembers.numberOfObjects);

    // Create c1
    CircleWithStaticMembers c1 = new CircleWithStaticMembers();

    // Display c1 BEFORE c2 is created
    System.out.println("\nAfter creating c1");
    System.out.println("c1: radius (" + c1.radius +
      ") and number of Circle objects (" +
      c1.numberOfObjects + ")");

    // Create c2
    CircleWithStaticMembers c2 = new CircleWithStaticMembers(5);

    // Modify c1
    c1.radius = 9;

    // Display c1 and c2 AFTER c2 was created
    System.out.println("\nAfter creating c2 and modifying c1");
    System.out.println("c1: radius (" + c1.radius +
      ") and number of Circle objects (" +
      c1.numberOfObjects + ")");
    System.out.println("c2: radius (" + c2.radius +
      ") and number of Circle objects (" +
      c2.numberOfObjects + ")");
  }
}
```

A static variable can be accessed via its class name.

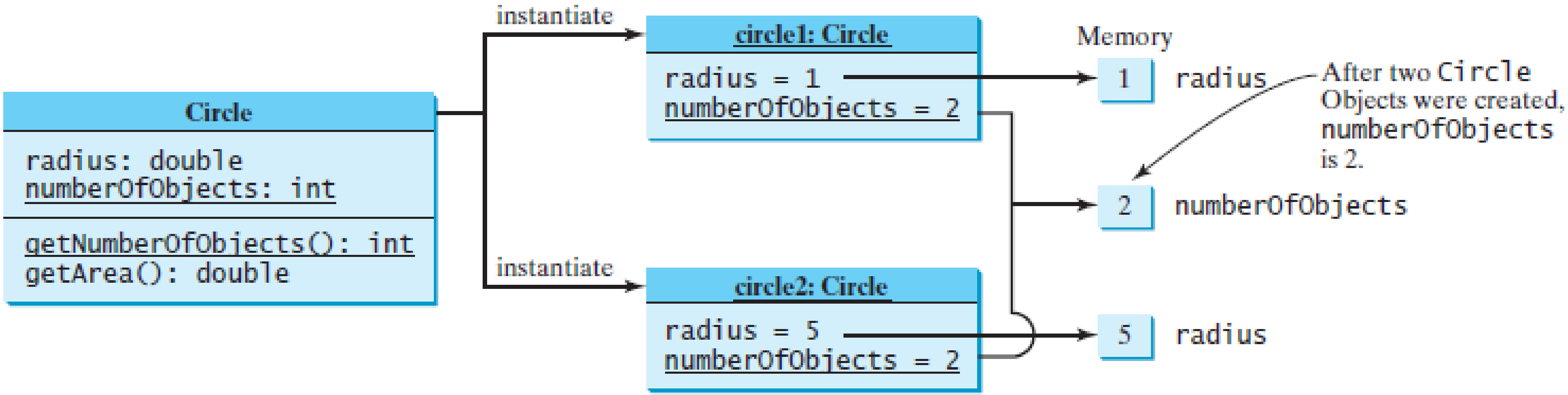
A static variable can also be accessed via objects of the class.

Example (Output)



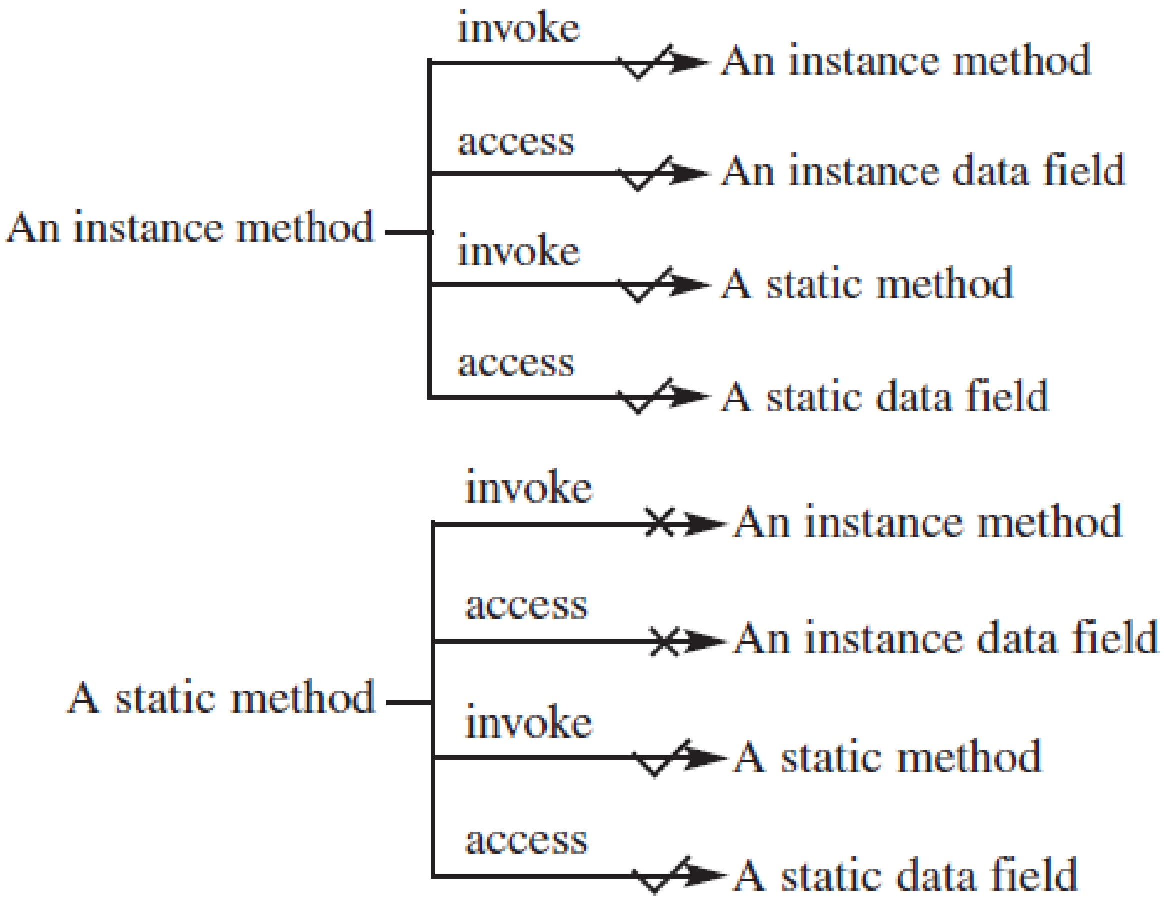
```
Before creating objects
The number of Circle objects is 0
After creating c1
c1: radius (1.0) and number of Circle objects (1)
After creating c2 and modifying c1
c1: radius (9.0) and number of Circle objects (2)
c2: radius (5.0) and number of Circle objects (2)
```

UML Class Diagram: Circle with Static Members

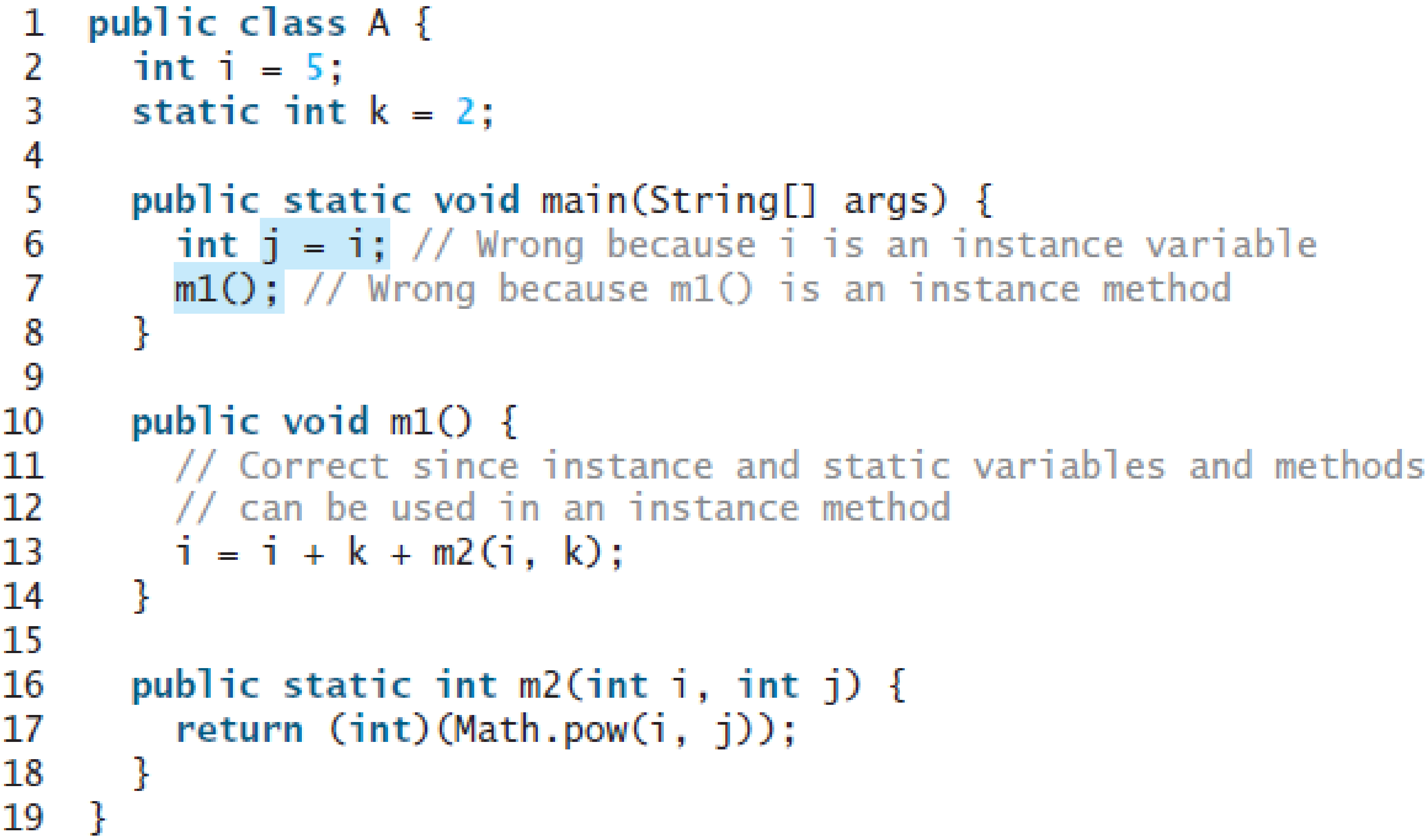


Static members are underlined in UML class diagrams.

Relationship between Static and Instance Members

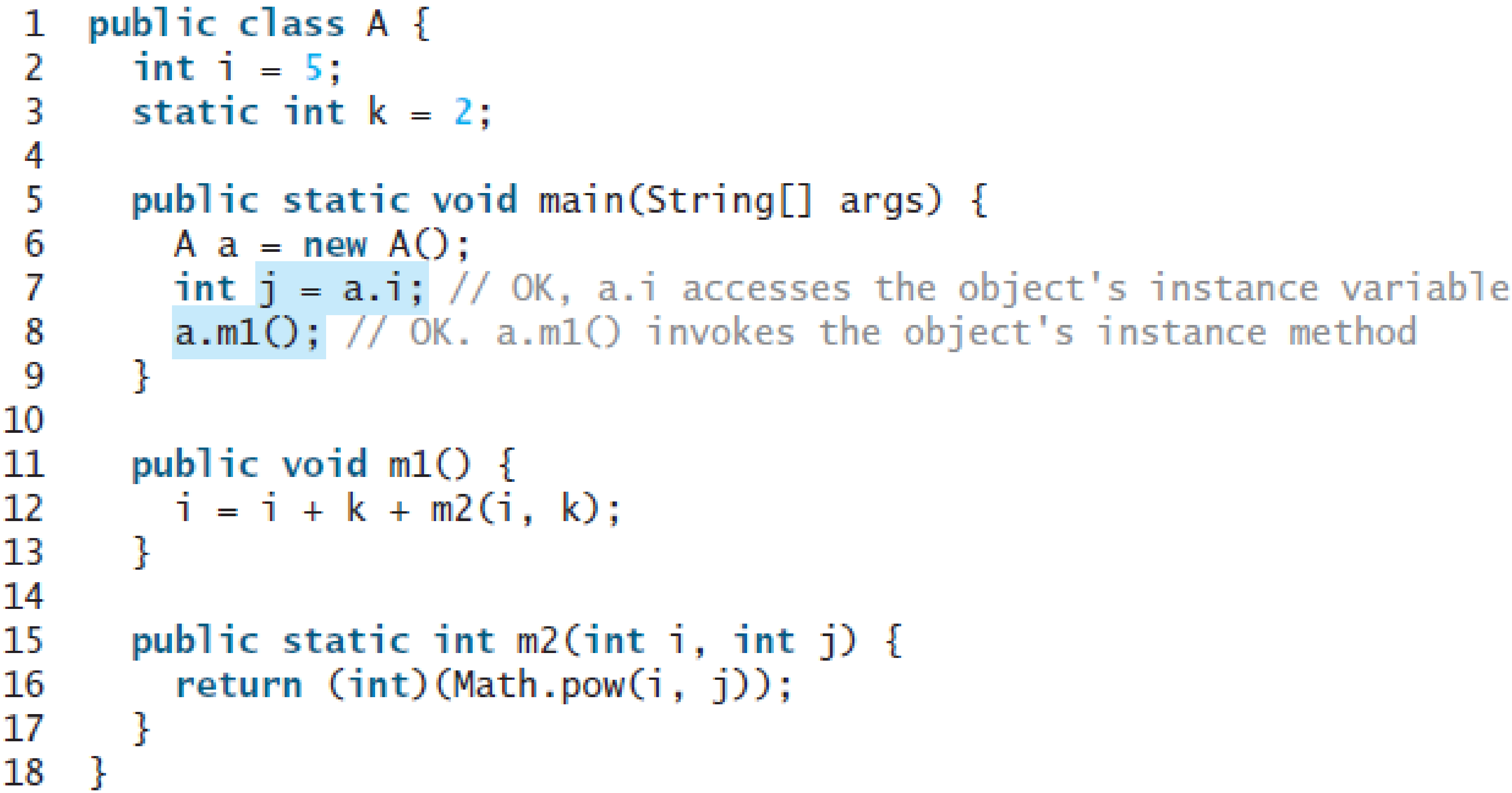


Relationship between Static and Instance Members (Example 1)



```
public class A {
  int i = 5;
  static int k = 2;

  public static void main(String[] args) {
    int j = i; // Wrong because i is an instance variable
    m1(); // Wrong because m1() is an instance method
  }

  public void m1() {
    // Correct since instance and static variables and methods
    // can be used in an instance method
    i = i + k + m2(i, k);
  }

  public static int m2(int i, int j) {
    return (int)(Math.pow(i, j));
  }
}
```

Relationship between Static and Instance Members (Example 2)



```java
public class A {
  int i = 5;
  static int k = 2;

  public static void main(String[] args) {
    A a = new A();
    int j = a.i; // OK, a.i accesses the object's instance variable
    a.m1(); // OK. a.m1() invokes the object's instance method
  }

  public void m1() {
    i = i + k + m2(i, k);
  }

  public static int m2(int i, int j) {
    return (int)(Math.pow(i, j));
  }
}
```

Instance or Static?

- How to decide whether a variable or method should be an instance one or static one?
  - A variable or method that is *dependent* on a specific instance of the class should be an *instance* variable or method.
    - Example: *radius* and *getArea* of the *Circle* class; each circle has its own radius and area.
  - A variable or method that is *not dependent* on a specific instance of the class should be a *static* variable or method.
    - Example: *numberOfObjects* of the Circle class; all circles should share this value.

Visibility Modifiers

- *Visibility modifiers* can be used to specify the *visibility* of a class and its members.
- A *visibility modifier* specifies how data fields and methods in a class can be accessed from outside the class.
  - There is no restriction on accessing data fields and methods from inside the class.

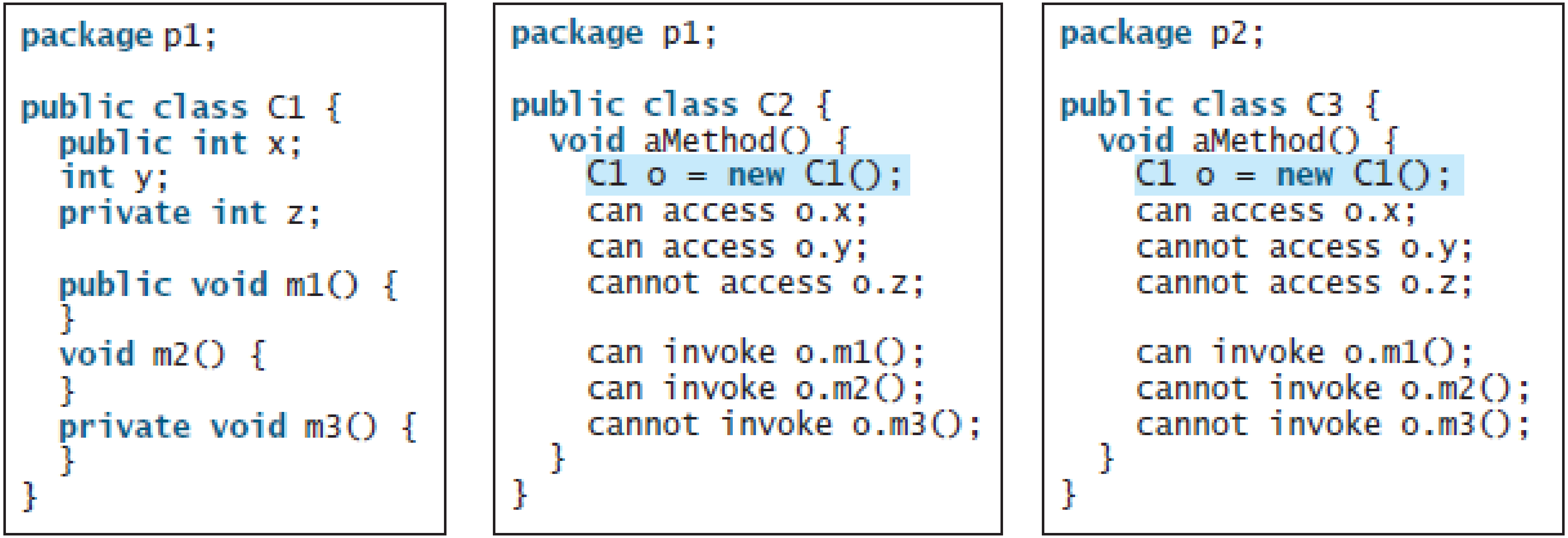
Visibility Modifiers: The Default

- If no visibility modifier is used, then by *default* the classes, methods, and data fields are accessible by any class in the same package.
  - This is known as *package-private* or *package-access*.
- *Packages* are used to organize classes. To do so, you need to add the following statement as the first statement in the program.
  - *package packageName;*
- If a class is defined without the package statement, it is said to be placed in the default package.

Visibility Modifiers: Public and Private

- The *public* modifier can be used for classes, methods and data fields to denote that they can be accessed from any other classes.
- The *private* modifier makes methods and data fields accessible only from within its own class.

Visibility Modifiers: Methods and Data Fields Example



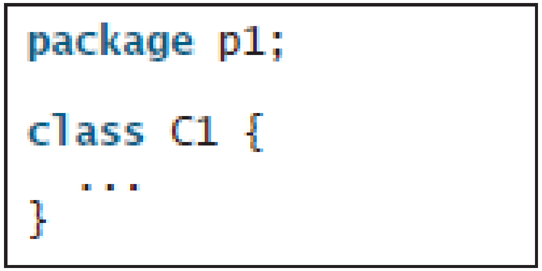
```
package p1;

public class C1 {
  public int x;
  int y;
  private int z;

  public void m1() {
  }
  void m2() {
  }
  private void m3() {
  }
}
```

```
package p1;

public class C2 {
  void aMethod() {
    C1 o = new C1();
    can access o.x;
    can access o.y;
    cannot access o.z;

    can invoke o.m1();
    can invoke o.m2();
    cannot invoke o.m3();
  }
}
```

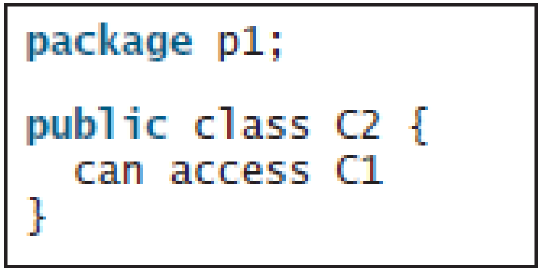
```
package p2;

public class C3 {
  void aMethod() {
    C1 o = new C1();
    can access o.x;
    cannot access o.y;
    cannot access o.z;

    can invoke o.m1();
    cannot invoke o.m2();
    cannot invoke o.m3();
  }
}
```

- The *private* modifier restricts access to its defining class.
- The *default* modifier restricts access to a package.
- The *public* modifier enables unrestricted access.

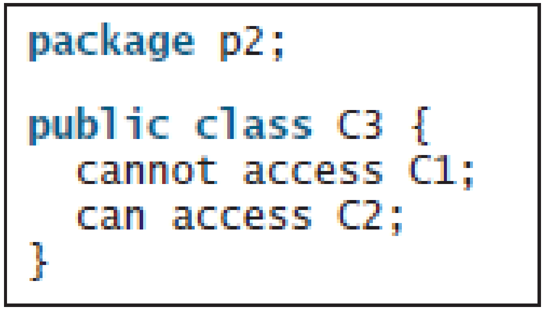
Visibility Modifiers: Classes Example



```
package p1;

class C1 {
  ...
}
```

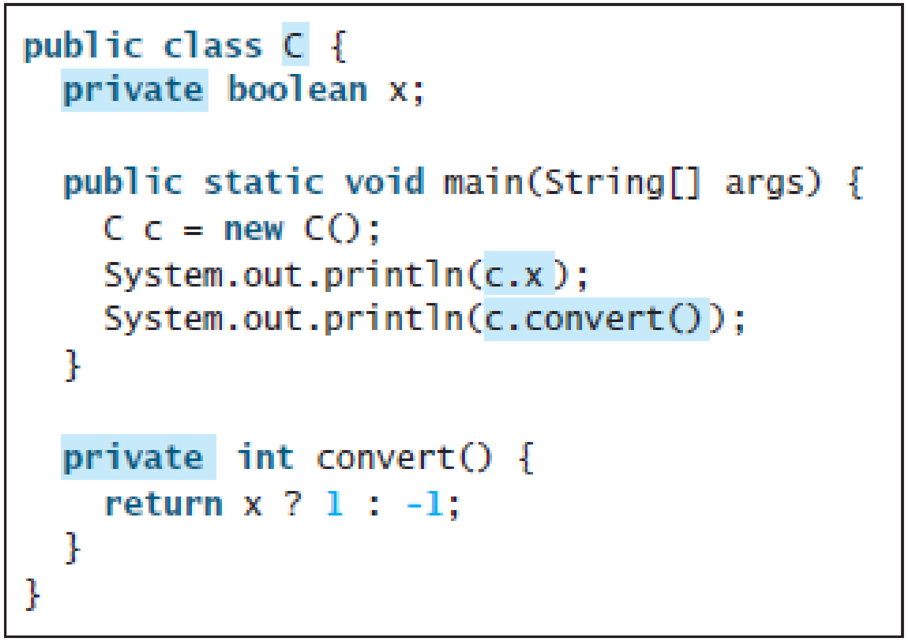


```
package p1;

public class C2 {
  can access C1
}
```



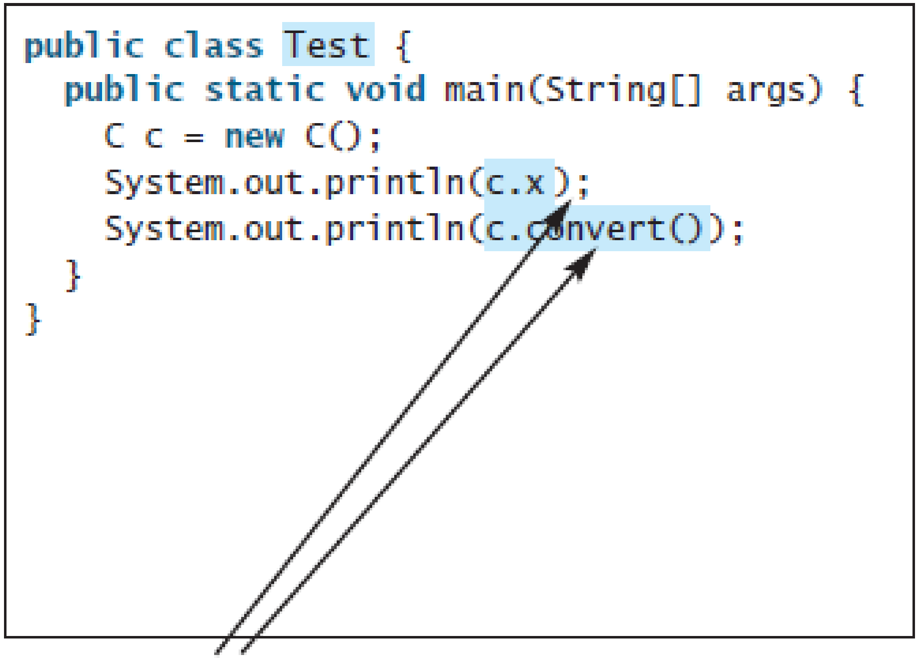
```
package p2;

public class C3 {
  cannot access C1;
  can access C2;
}
```

Visibility Modifiers: Another Example



```
public class C {
  private boolean x;

  public static void main(String[] args) {
    C c = new C();
    System.out.println(c.x);
    System.out.println(c.convert());
  }

  private int convert() {
    return x ? 1 : -1;
  }
}
```

(a) This is okay because object c is used inside the class C.



```
public class Test {
  public static void main(String[] args) {
    C c = new C();
    System.out.println(c.x);
    System.out.println(c.convert());
  }
}
```

(b) This is wrong because x and convert are private in class C.

Visibility Modifiers: Comments

- The *private* modifier applies only to the members of a class.
- The *public* modifier can apply to a class or members of a class.
- Using the modifiers *public* and *private* on local variables would cause a compile error.

Data Field Encapsulation

- It is not a good practice to allow data fields to be directly modified.
  - Data may be tampered with.
  - The class becomes difficult to maintain and vulnerable to bugs.
- To prevent direct modifications of data fields, you should declare the data fields *private*.
  - This is known as *data field encapsulation*.

Data Field Encapsulation (Cont.)

- A private data field cannot be accessed by an object from outside the class that defines the private field.
- However, a client often needs to retrieve and modify a data field.
- To make a private data field accessible:
  - Provide a *getter (accessor)* method to return its value.
  - Provide a *setter (mutator)* method set a new value to it.

Data Field Encapsulation (Cont.)

- A *getter* method has the following signature:

  public returnType getPropertyName()

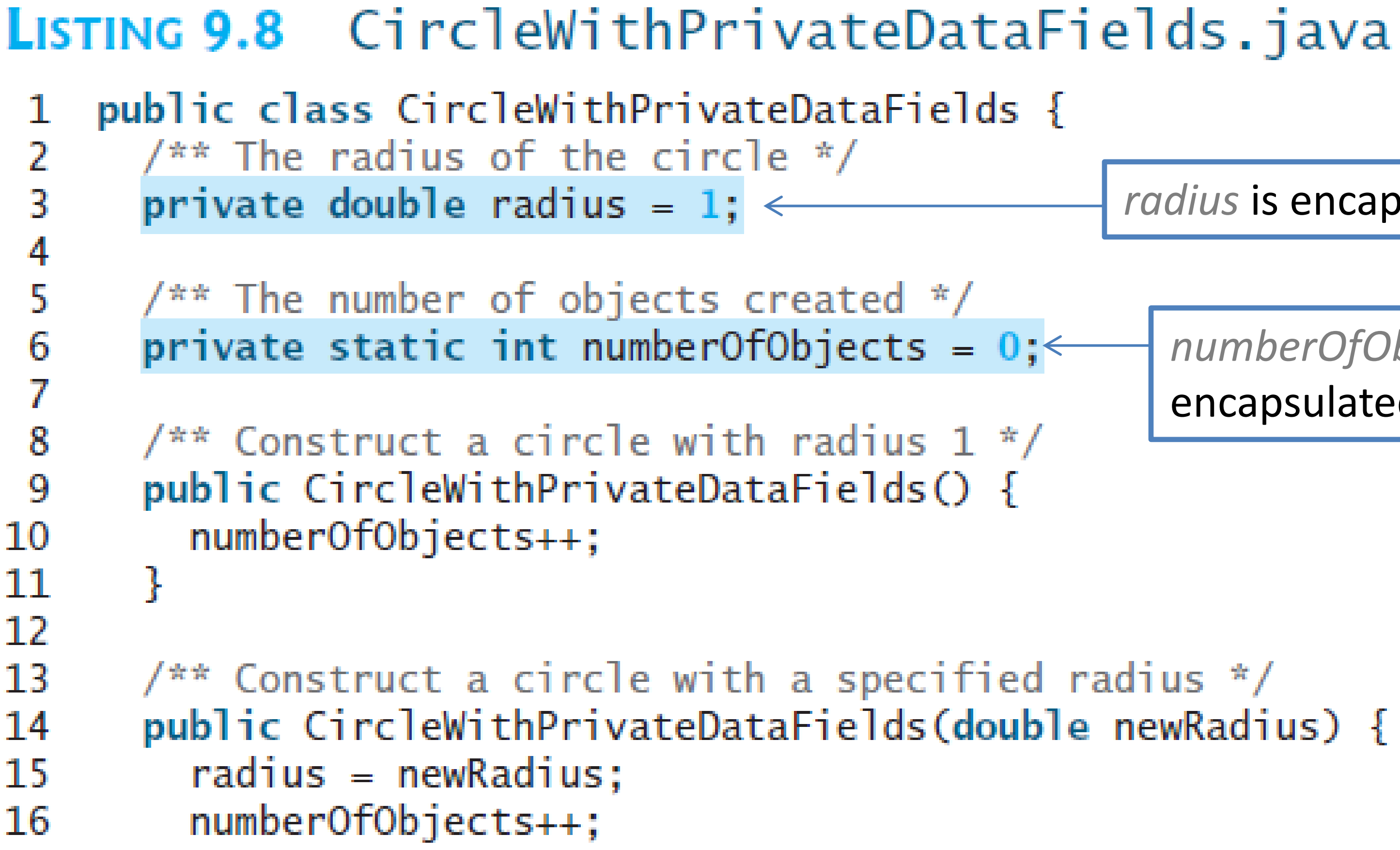
  – If the *returnType* is *boolean*, the *get* method is defined as follows by convention:

  public boolean isProperyName()

- A set method has the following signature:

  public void setPropertyName(dataType propertyValue)

Example



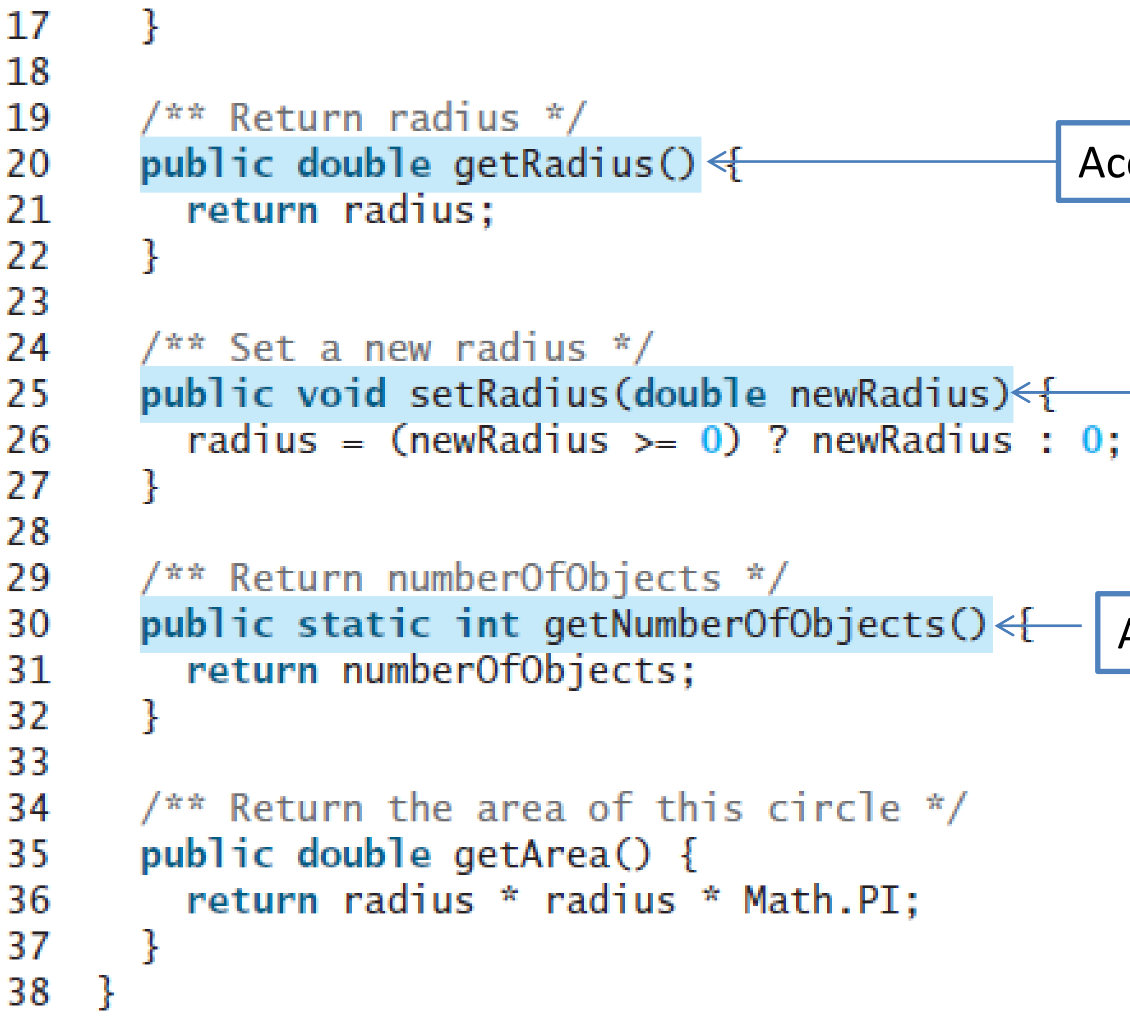
LISTING 9.8 CircleWithPrivateDataFields.java

```java
public class CircleWithPrivateDataFields {
  /** The radius of the circle */
  private double radius = 1;

  /** The number of objects created */
  private static int numberOfObjects = 0;

  /** Construct a circle with radius 1 */
  public CircleWithPrivateDataFields() {
    numberOfObjects++;
  }

  /** Construct a circle with a specified radius */
  public CircleWithPrivateDataFields(double newRadius) {
    radius = newRadius;
    numberOfObjects++;
```

radius is encapsulated

numberOfObjects is encapsulated

Example (Cont.)



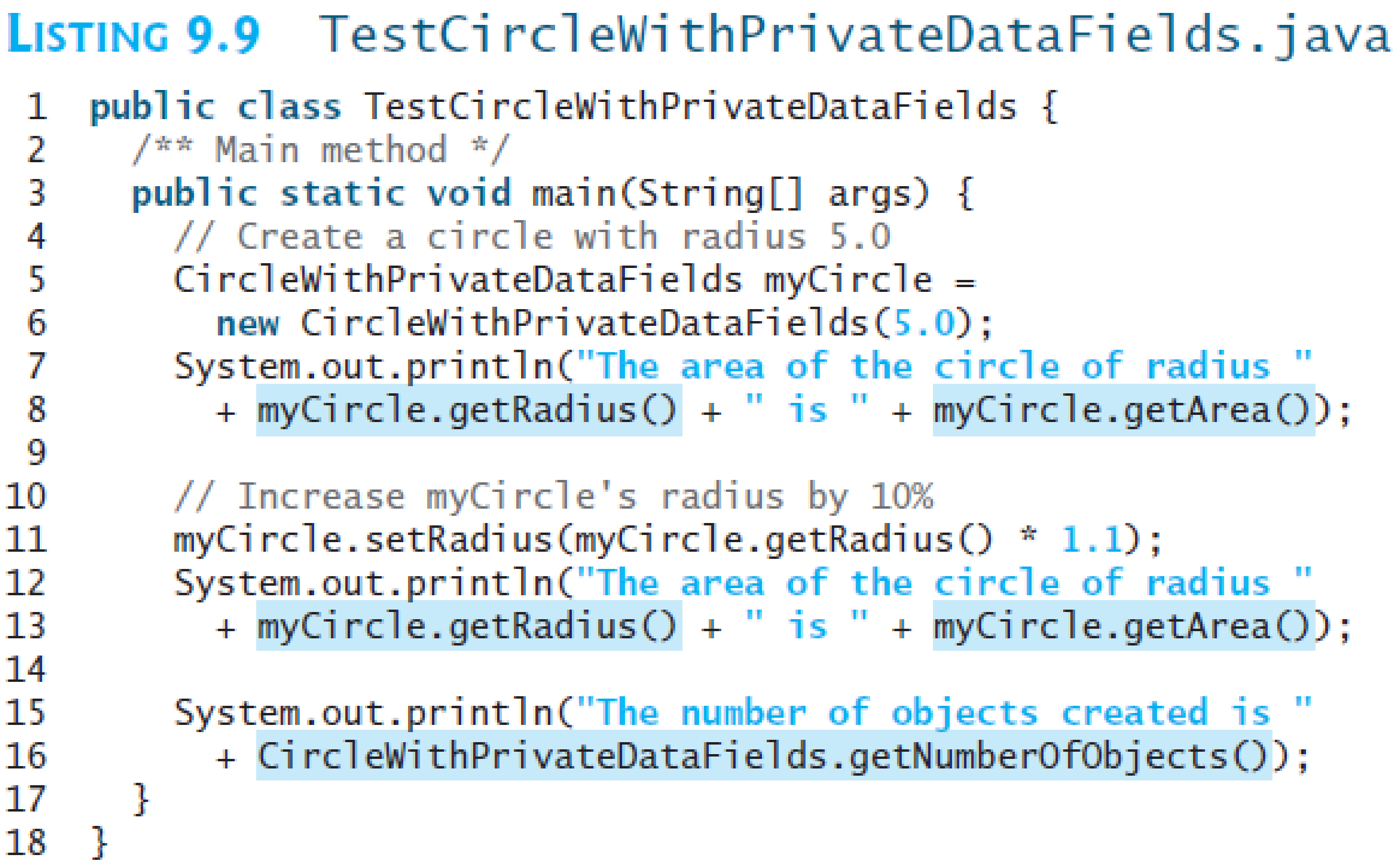
```
  }

  /** Return radius */
  public double getRadius() {
    return radius;
  }

  /** Set a new radius */
  public void setRadius(double newRadius) {
    radius = (newRadius >= 0) ? newRadius : 0;
  }

  /** Return numberOfObjects */
  public static int getNumberOfObjects() {
    return numberOfObjects;
  }

  /** Return the area of this circle */
  public double getArea() {
    return radius * radius * Math.PI;
  }
}
```

Accessor method

Mutator method

Accessor method

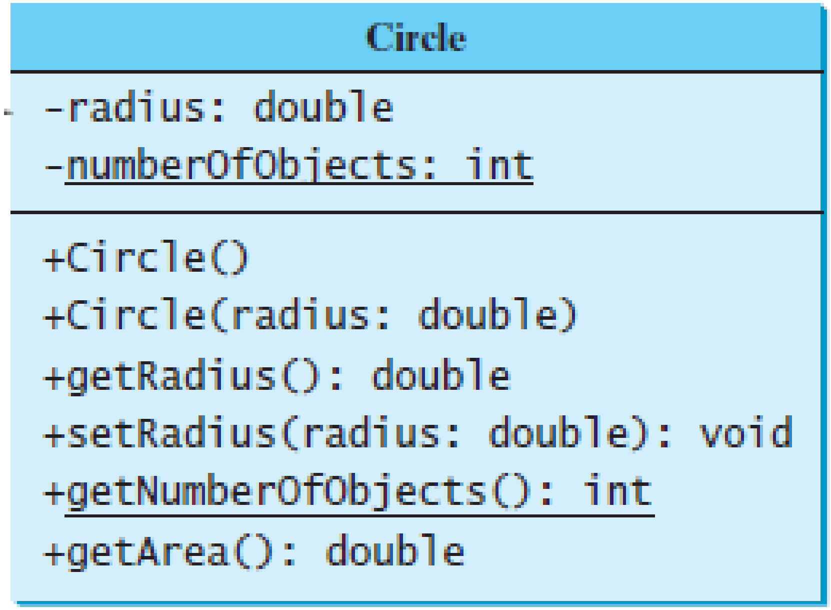
Example (Cont.)



Listing 9.9 TestCircleWithPrivateDataFields.java

```java
public class TestCircleWithPrivateDataFields {
  /** Main method */
  public static void main(String[] args) {
    // Create a circle with radius 5.0
    CircleWithPrivateDataFields myCircle =
      new CircleWithPrivateDataFields(5.0);
    System.out.println("The area of the circle of radius "
      + myCircle.getRadius() + " is " + myCircle.getArea());

    // Increase myCircle's radius by 10%
    myCircle.setRadius(myCircle.getRadius() * 1.1);
    System.out.println("The area of the circle of radius "
      + myCircle.getRadius() + " is " + myCircle.getArea());

    System.out.println("The number of objects created is "
      + CircleWithPrivateDataFields.getNumberOfObjects());
  }
}
```

UML Class Diagram: Circle with Private Data Fields



Circle

-radius: double

-numberOfObjects: int

+Circle()

+Circle(radius: double)

+getRadius(): double

+setRadius(radius: double): void

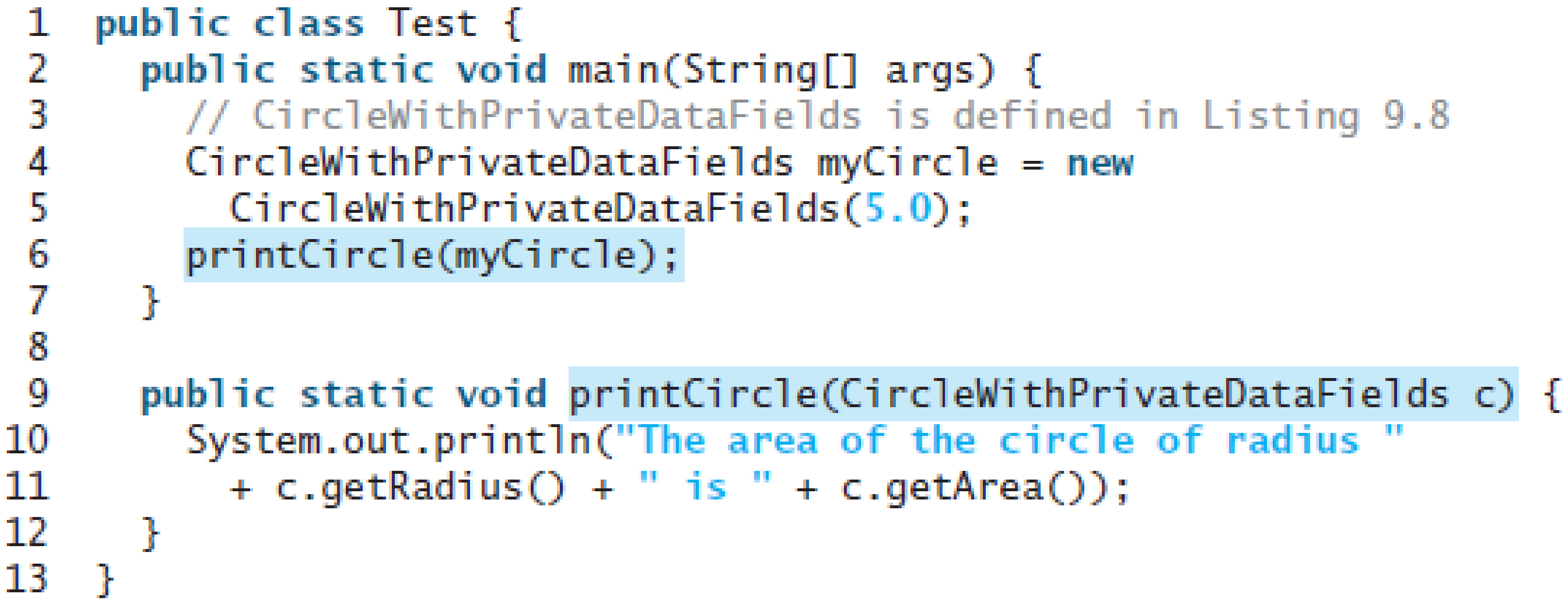
+getNumberOfObjects(): int

+getArea(): double

- The (-) sign indicates a private modifier.
- The (+) sign indicates a public modifier.

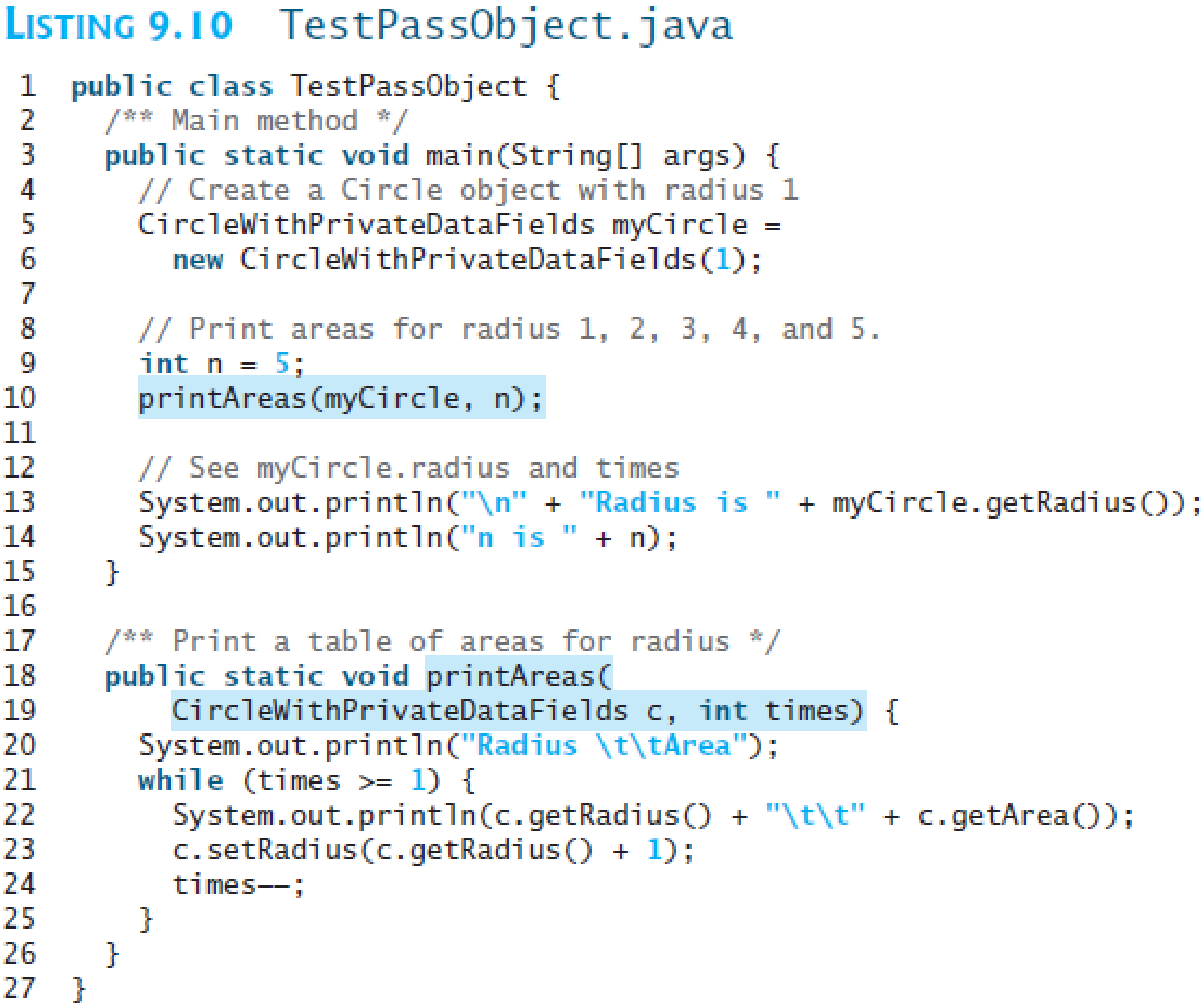
Passing Objects to Methods

- Passing an object to a method is to pass the reference of the object.
- The following code passes the *myCircle* object as an argument to the *printCircle* method:



```
public class Test {
  public static void main(String[] args) {
    // CircleWithPrivateDataFields is defined in Listing 9.8
    CircleWithPrivateDataFields myCircle = new
      CircleWithPrivateDataFields(5.0);
    printCircle(myCircle);
  }

  public static void printCircle(CircleWithPrivateDataFields c) {
    System.out.println("The area of the circle of radius "
      + c.getRadius() + " is " + c.getArea());
  }
}
```

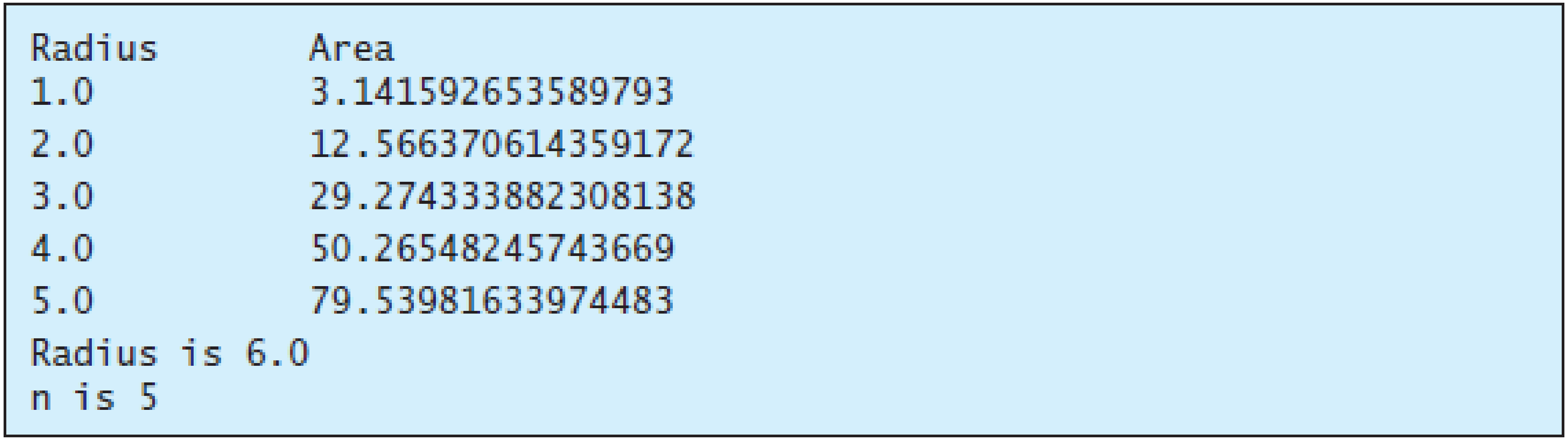
Passing Objects to Methods (Example)



LISTING 9.10 TestPassObject.java

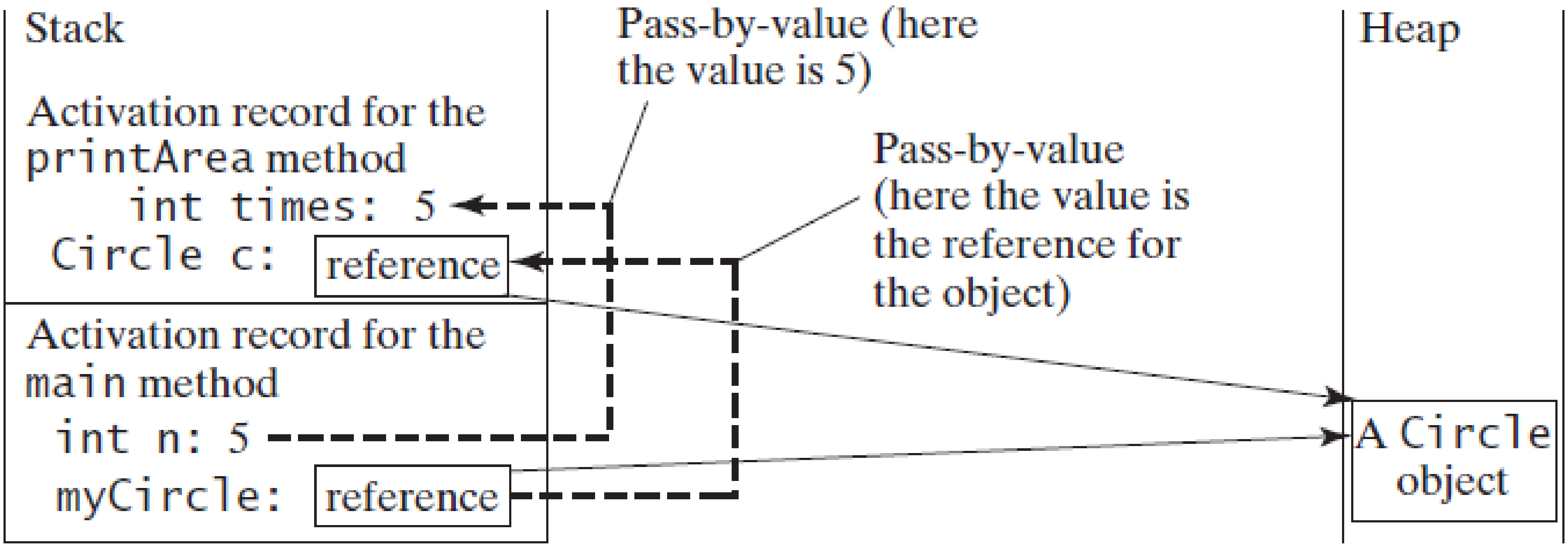
```java
public class TestPassObject {
  /** Main method */
  public static void main(String[] args) {
    // Create a Circle object with radius 1
    CircleWithPrivateDataFields myCircle =
      new CircleWithPrivateDataFields(1);

    // Print areas for radius 1, 2, 3, 4, and 5.
    int n = 5;
    printAreas(myCircle, n);

    // See myCircle.radius and times
    System.out.println("\n" + "Radius is " + myCircle.getRadius());
    System.out.println("n is " + n);
  }

  /** Print a table of areas for radius */
  public static void printAreas(
      CircleWithPrivateDataFields c, int times) {
    System.out.println("Radius \t\tArea");
    while (times >= 1) {
      System.out.println(c.getRadius() + "\t\t" + c.getArea());
      c.setRadius(c.getRadius() + 1);
      times--;
    }
  }
}
```

Passing Objects to Methods Example Output



```
Radius       Area
1.0          3.141592653589793
2.0          12.566370614359172
3.0          29.274333882308138
4.0          50.26548245743669
5.0          79.53981633974483
Radius is 6.0
n is 5
```

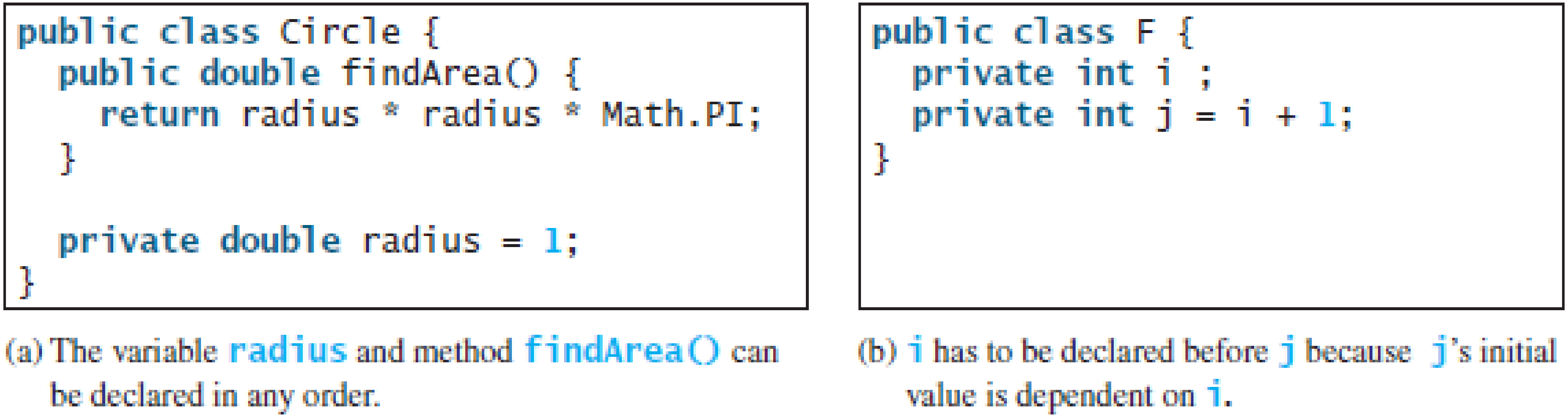
Passing Objects to Methods Example Explanation


Stack
Pass-by-value (here the value is 5)
Heap
Activation record for the printArea method
int times: 5
Circle c:
reference
Pass-by-value (here the value is the reference for the object)
Activation record for the main method
int n: 5
myCircle:
reference
A Circle object

The Scope of Variables

- The scope of a *class's variables* or *data fields* is the *entire class*, regardless of where the variables are declared.
- A class's variables and methods can appear in any order in the class.
  - The exception is when a data field is initialized based on a reference to another data field.

The Scope of Variables (Cont.)



```
public class Circle {
  public double findArea() {
    return radius * radius * Math.PI;
  }

  private double radius = 1;
}
```

(a) The variable radius and method findArea() can be declared in any order.

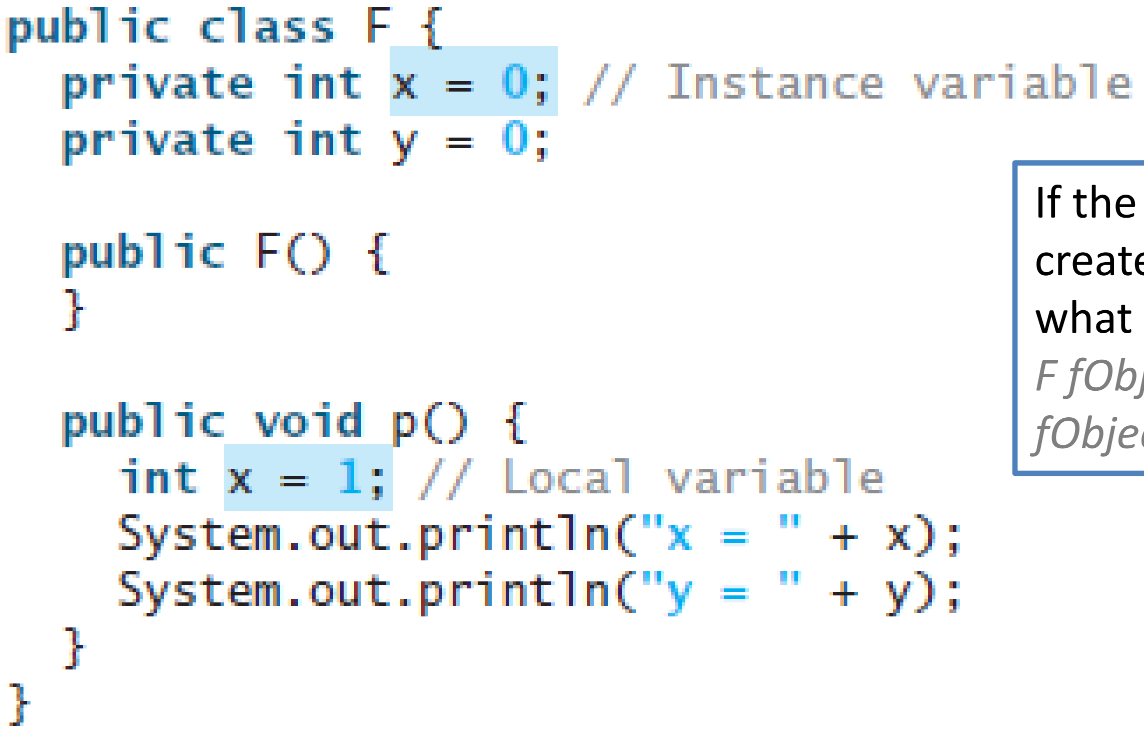
```
public class F {
  private int i ;
  private int j = i + 1;
}
```

(b) i has to be declared before j because j's initial value is dependent on i.

The Scope of Variables (Cont.)

- You can declare a class's variable only once.
  - But you can declare the same variable name in a method many times in different nonnesting blocks.
- If a local variable has the same name as a class's variable, the local variable takes precedence and the class's variable with the same name is *hidden*.

The Scope of Variables (Cont.)



```
public class F {
  private int x = 0; // Instance variable
  private int y = 0;

  public F() {
  }

  public void p() {
    int x = 1; // Local variable
    System.out.println("x = " + x);
    System.out.println("y = " + y);
  }
}
```

If the following statements are created in the *main* method, what is the output?

F fObject = new F();

fObject.print();

Object-Oriented Problem Solving

Objects & Classes (Part III)

Based on Chapters 9 & 10 of “Introduction to Java Programming” by Y. Daniel Liang.

Eng. Asma Abdel Karim
Computer Engineering Department

Outline

- Array of Objects (9.11)
- Immutable Objects and Classes (9.12)
- The *this* reference (9.14)
- Method Abstraction and Stepwise Refinement (6.11)
- Class Abstraction and Encapsulation (10.2)
- Thinking in Objects (10.3)
- Class Relationships (10.4)
- Processing Primitive Data Type Values as Objects. (10.7 & 10.8)
- The BigInteger and BigDecimal Classes (10.9)

Array of Objects

- An array can hold objects as well as primitive type values.
- The following statement declares and creates an array of ten *Circle* objects:

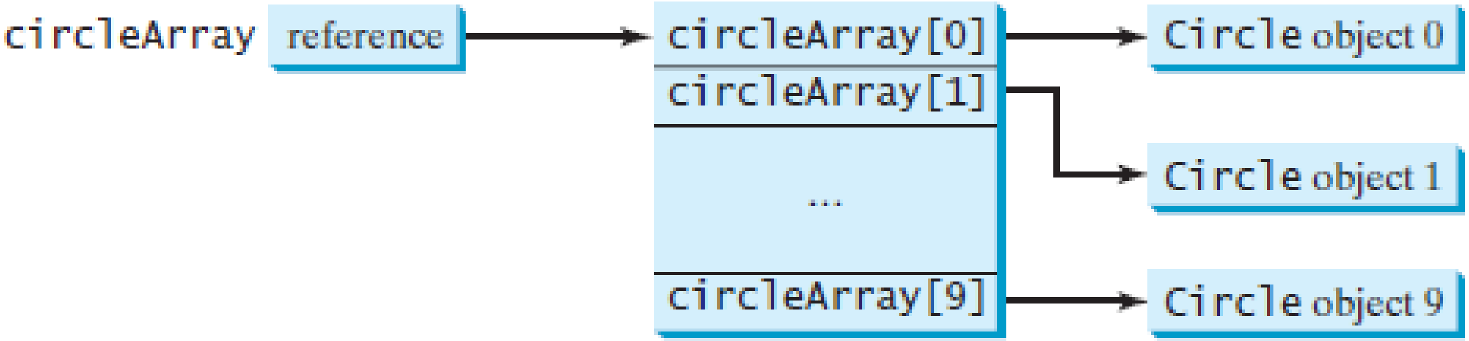
```
Circle[] circleArray = new Circle[10];
```

- To initialize *circleArray*, you can use a *for* loop like this one:

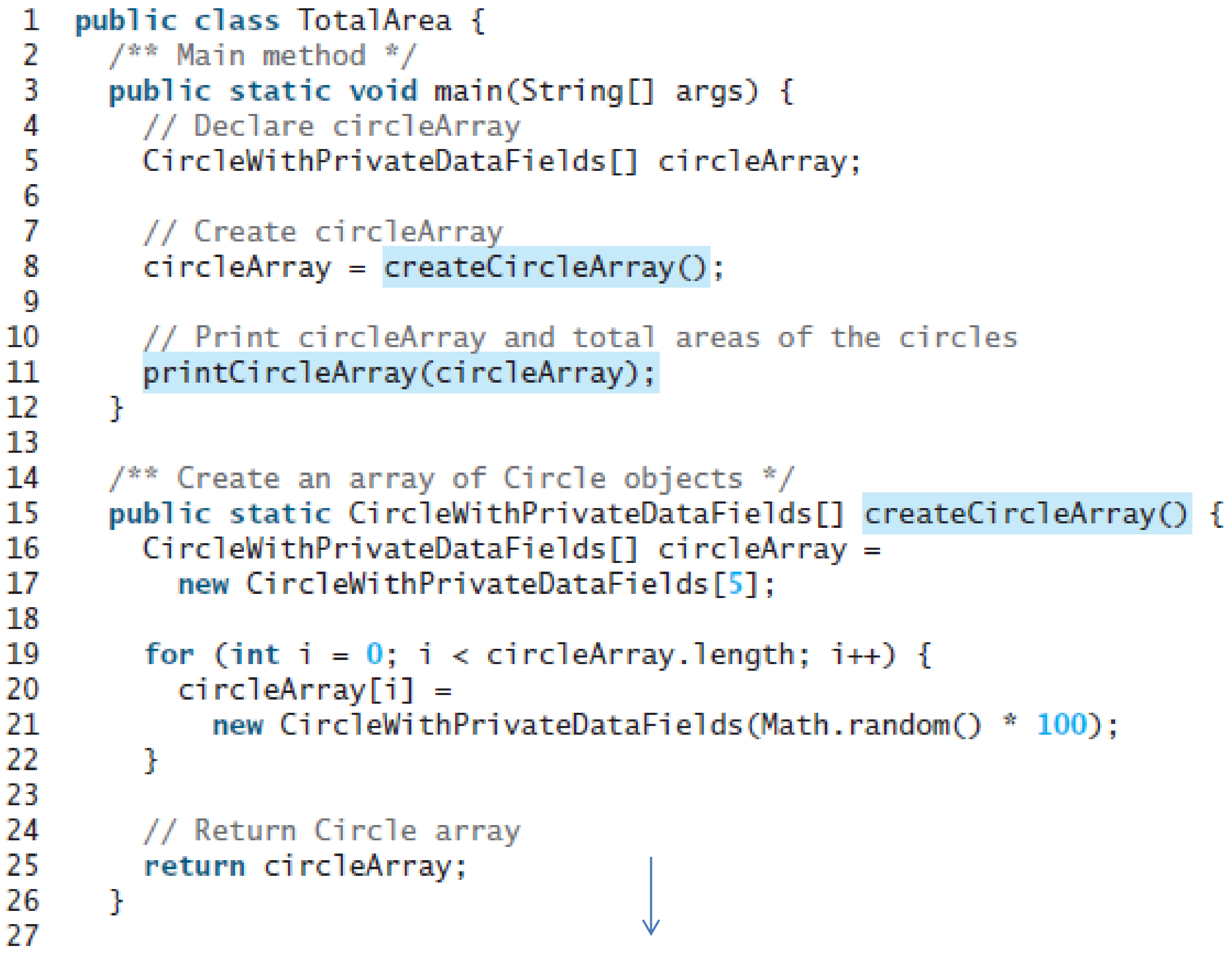
```
for (int i = 0; i < circleArray.length; i++) {
     circleArray[i] = new Circle();
}
```

Array of Objects (Cont.)

- An array of objects is actually an array of reference variables. So, invoking *circleArray[1].getArea()* involves two levels of referencing.
  - *circleArray* references the entire array;
  - *circleArray[1]* references a **Circle** object.
- When an array of objects is created using the *new* operator, each element in the array is a reference variable with a default value of *null*.

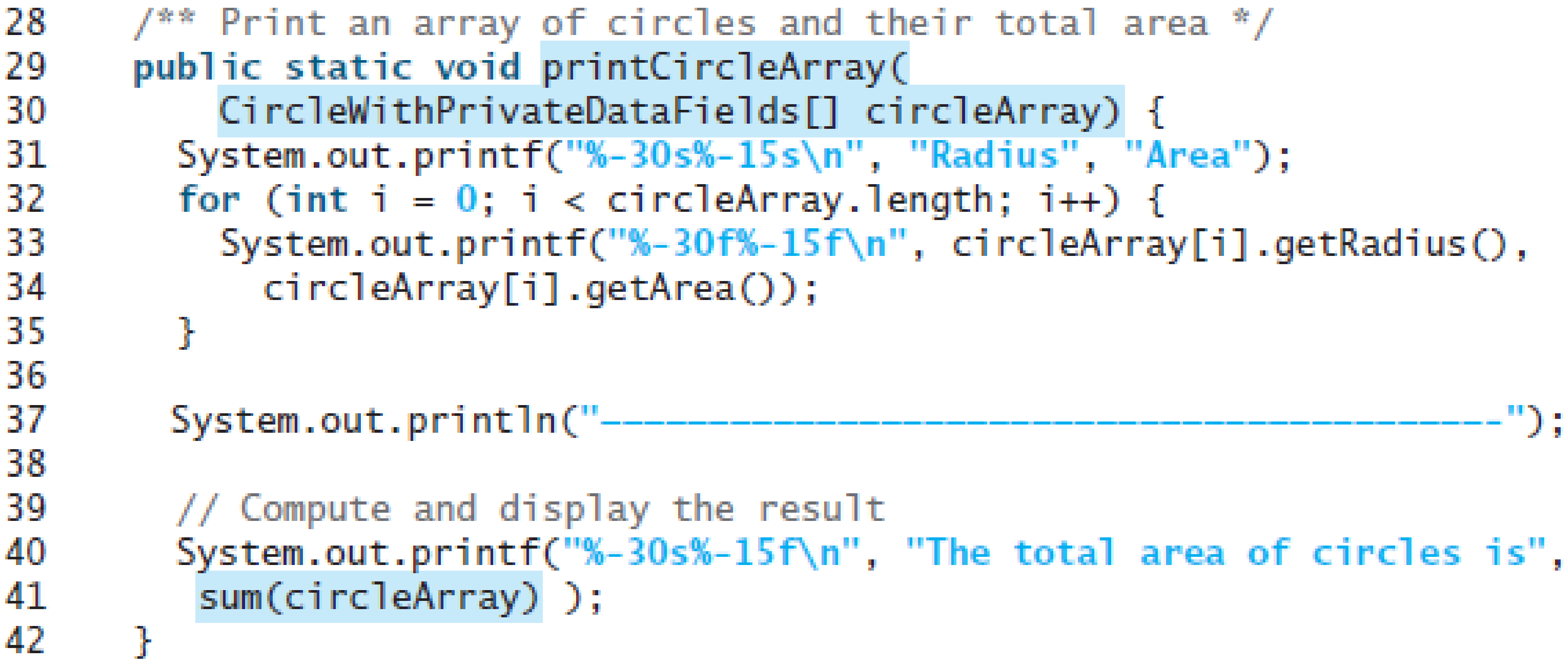


Array of Objects (Example)



```java
public class TotalArea {
  /** Main method */
  public static void main(String[] args) {
    // Declare circleArray
    CircleWithPrivateDataFields[] circleArray;

    // Create circleArray
    circleArray = createCircleArray();

    // Print circleArray and total areas of the circles
    printCircleArray(circleArray);
  }

  /** Create an array of Circle objects */
  public static CircleWithPrivateDataFields[] createCircleArray() {
    CircleWithPrivateDataFields[] circleArray =
      new CircleWithPrivateDataFields[5];

    for (int i = 0; i < circleArray.length; i++) {
      circleArray[i] =
        new CircleWithPrivateDataFields(Math.random() * 100);
    }

    // Return Circle array
    return circleArray;
  }

```

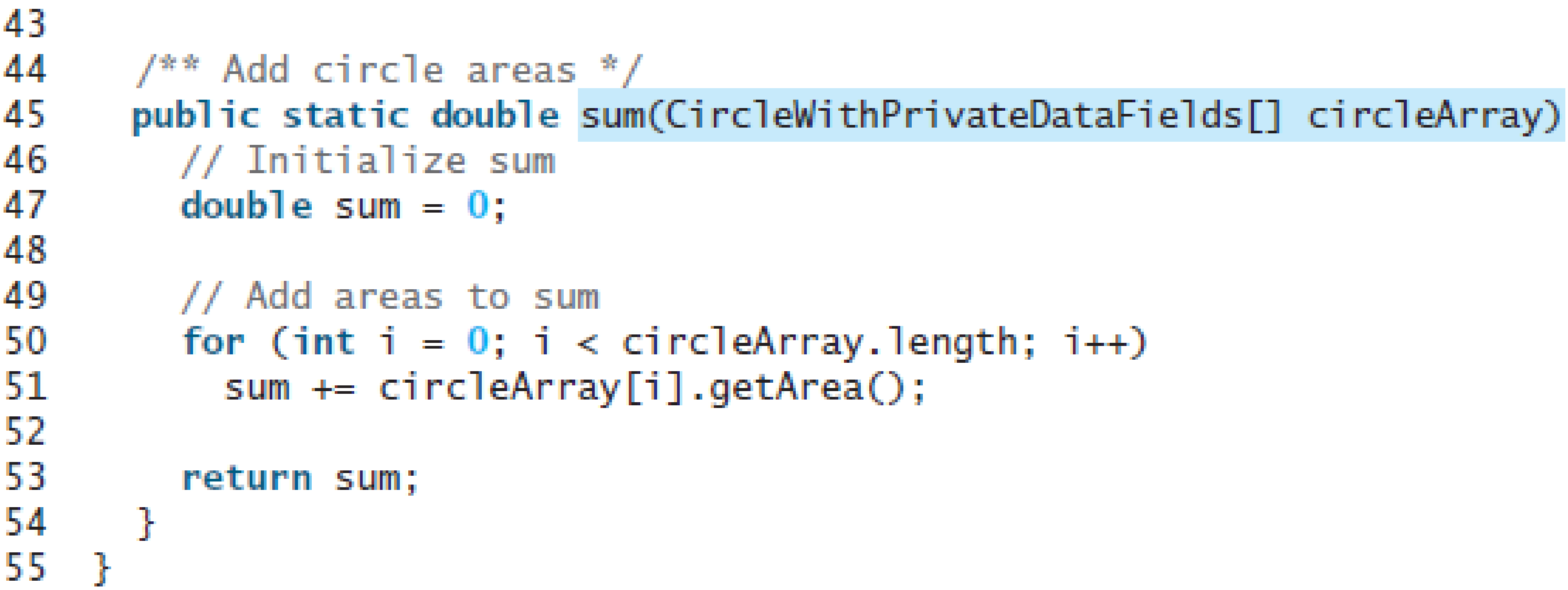
Array of Objects (Example Cont.)



```
/** Print an array of circles and their total area */
public static void printCircleArray(
    CircleWithPrivateDataFields[] circleArray) {
  System.out.printf("%-30s%-15s\n", "Radius", "Area");
  for (int i = 0; i < circleArray.length; i++) {
    System.out.printf("%-30f%-15f\n", circleArray[i].getRadius(),
      circleArray[i].getArea());
  }

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

  // Compute and display the result
  System.out.printf("%-30s%-15f\n", "The total area of circles is",
   sum(circleArray) );
}
```

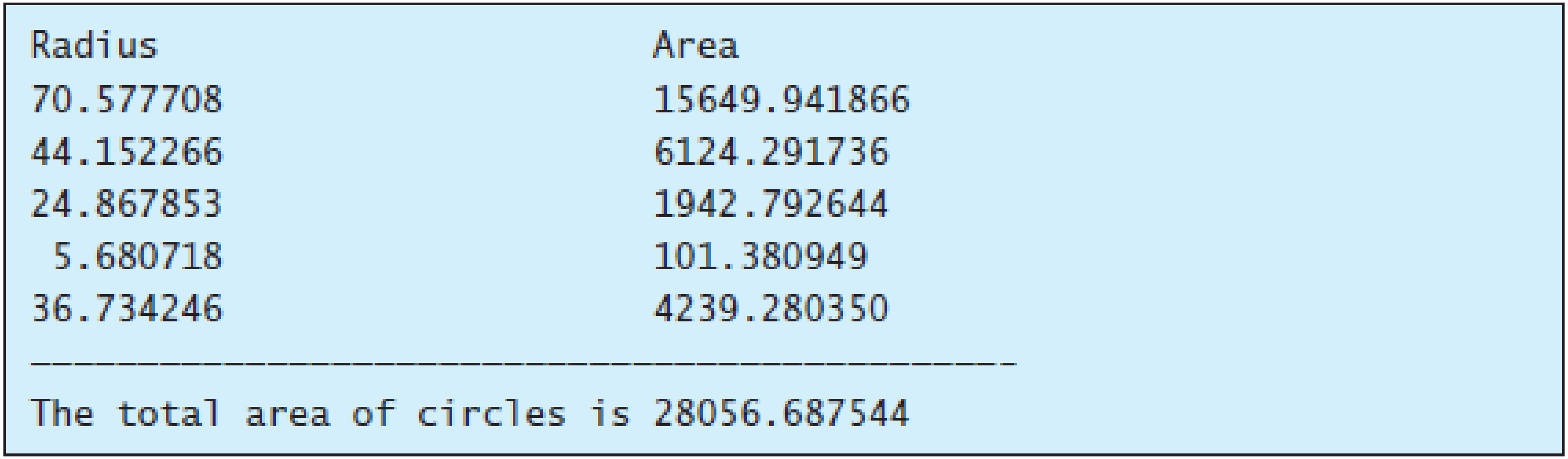
Eng. Asma Abdel Karim
Computer Engineering Department

Array of Objects (Example Cont.)



```java

  /** Add circle areas */
  public static double sum(CircleWithPrivateDataFields[] circleArray) {
    // Initialize sum
    double sum = 0;

    // Add areas to sum
    for (int i = 0; i < circleArray.length; i++)
      sum += circleArray[i].getArea();

    return sum;
  }
}
```

Array of Objects (Example Output)



```
Radius                       Area
70.577708                    15649.941866
44.152266                    6124.291736
24.867853                    1942.792644
 5.680718                    101.380949
36.734246                    4239.280350
----------------------------------------------
The total area of circles is 28056.687544
```

Immutable Objects and Classes

- Normally, you create an object and allow its contents to be changed later.
- However, occasionally it is desirable to create an object whose contents cannot be changed once the object has been created.
  - Such an object is called *immutable object* and its class is called *immutable class*.

Immutable Objects and Classes (Cont.)

- For a class to be immutable, it must meet the following requirements:
  - All data fields must be private.
  - There can’t be any mutator methods for data fields.
  - No accessor methods can return a reference to a data field that is mutable.

Immutable Objects and Classes (Example)

```
public class Student{
    private int id;
    private String name;
    private double [] grades = new double[3];

    public Student (int ssn, String newName){
        id = ssn;
        name = newName;
    }

    public int getId(){ return id; }

    public String getName(){ return name; }

    public double [] getGrades(){
        return grades;
    }
}
```

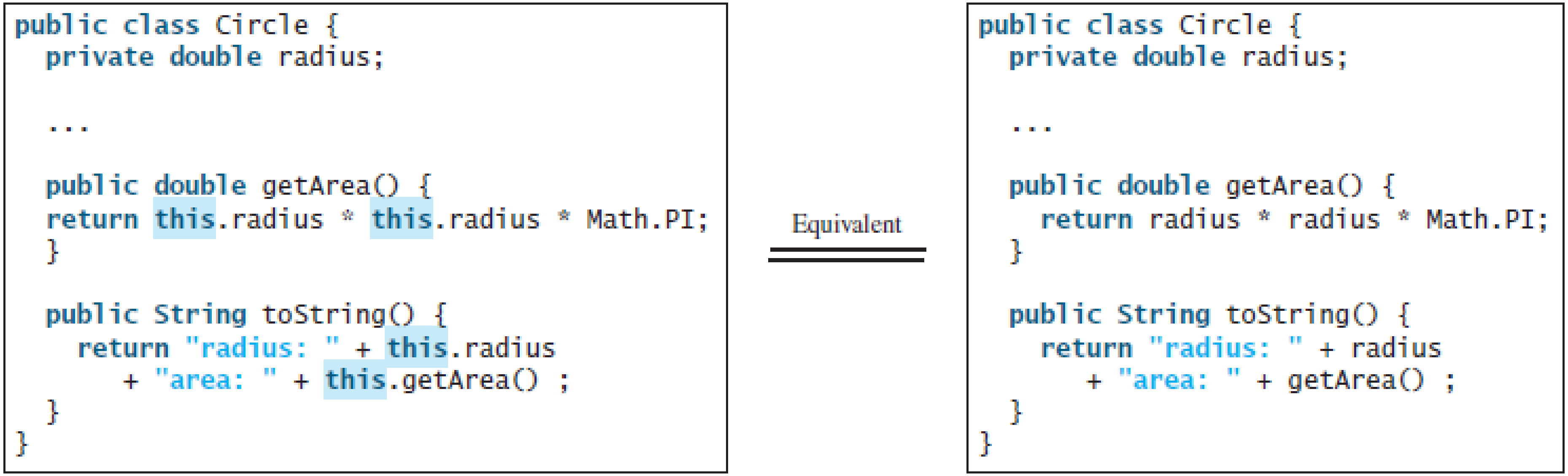
This method actually returns a reference to the array *grades*, which means it can be changed once returned.

Immutable Objects and Classes: Example (Cont.)

```
public class test {
    public static void main(String [] args){
        Student student = new Student (112233,  “John”);
        double [] G = student.getGrades();
        G[0] = 90.0;
        G[1] = 95.5;
        G[2] = 92.9;
    }
}
```

The *this* Reference

- The keyword *this* refers to the object itself.
- The *this keyword* is the name of a reference that an object can use to refer to itself



```
public class Circle {
  private double radius;

  ...

  public double getArea() {
  return this.radius * this.radius * Math.PI;
  }

  public String toString() {
    return "radius: " + this.radius
       + "area: " + this.getArea() ;
  }
}
```

(a)

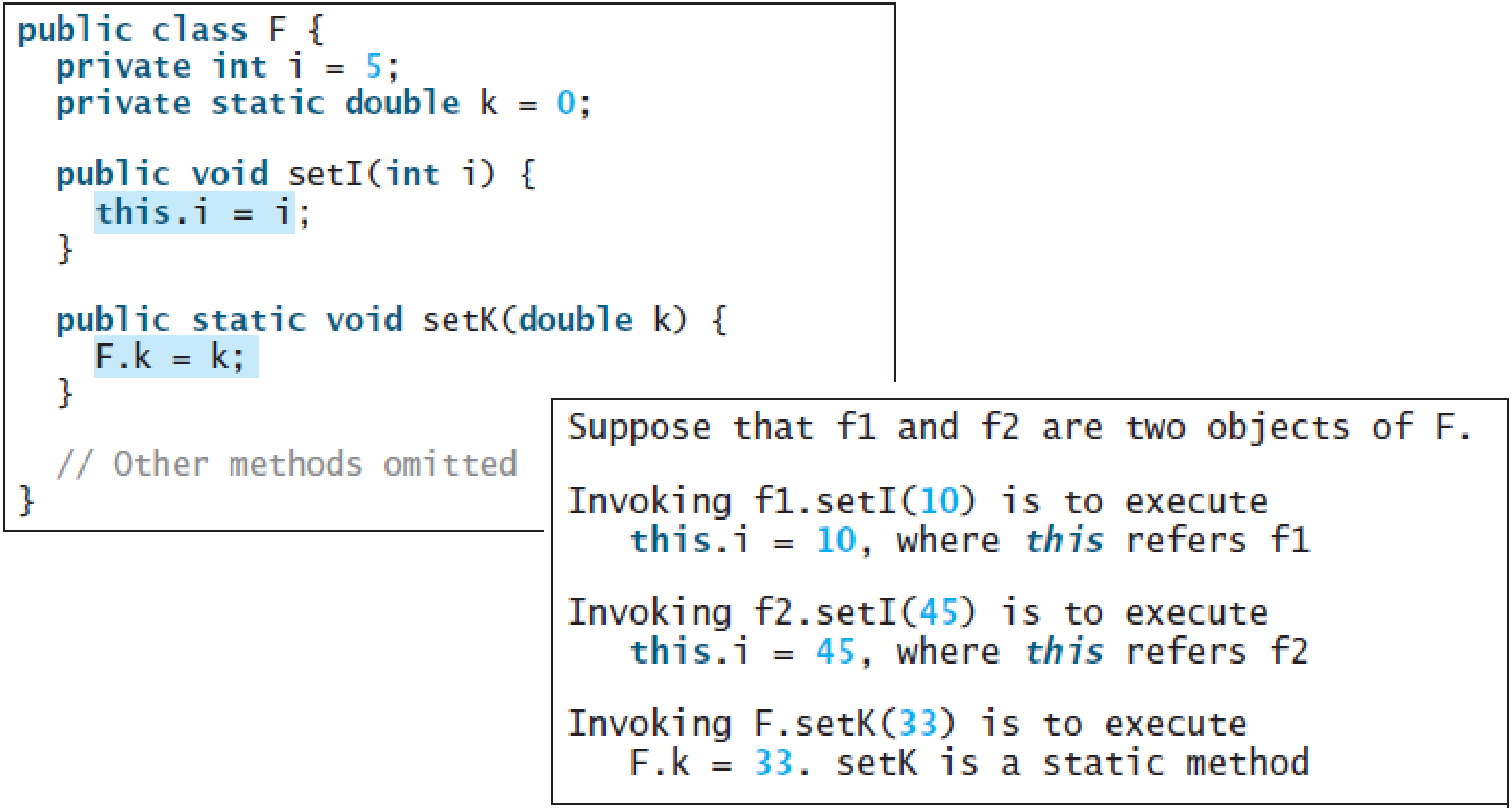
```
public class Circle {
  private double radius;

  ...

  public double getArea() {
    return radius * radius * Math.PI;
  }

  public String toString() {
    return "radius: " + radius
       + "area: " + getArea() ;
  }
}
```

(b)

Using *this* to Reference Hidden Data Fields

- The *this* keyword can be used to reference a class’s hidden data fields.
- A hidden *static variable* can be accessed simply by using the *ClassName.staticVariable*.
- A hidden *instance variable* can be accessed by using the keyword *this*.

Using *this* to Reference Hidden Data Fields: Example

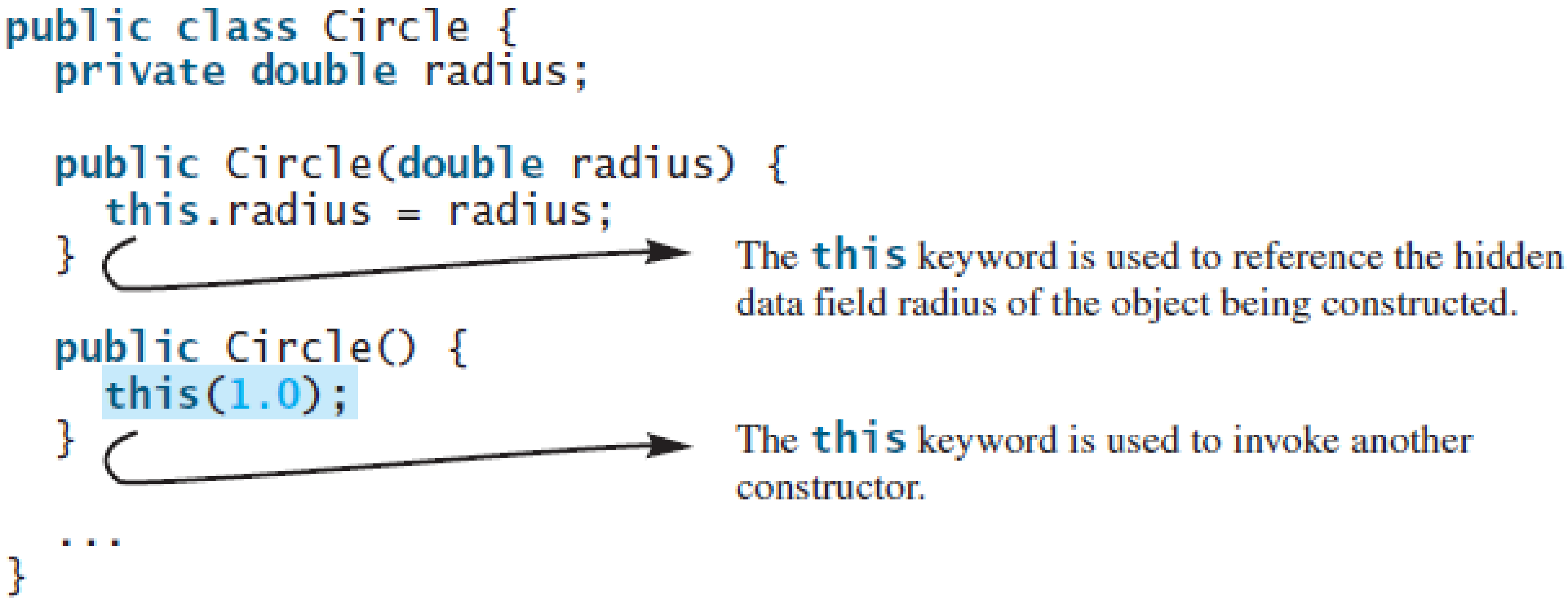


```java
public class F {
  private int i = 5;
  private static double k = 0;

  public void setI(int i) {
    this.i = i;
  }

  public static void setK(double k) {
    F.k = k;
  }

  // Other methods omitted
}
```

```
Suppose that f1 and f2 are two objects of F.

Invoking f1.setI(10) is to execute
   this.i = 10, where this refers f1

Invoking f2.setI(45) is to execute
   this.i = 45, where this refers f2

Invoking F.setK(33) is to execute
   F.k = 33. setK is a static method
```

Using *this* to Invoke a Constructor

- The *this* keyword can be used to invoke another constructor of the same class.



```
public class Circle {
  private double radius;

  public Circle(double radius) {
    this.radius = radius;
  }

  public Circle() {
    this(1.0);
  }

  ...
}
```

The `this` keyword is used to reference the hidden data field radius of the object being constructed.

The `this` keyword is used to invoke another constructor.

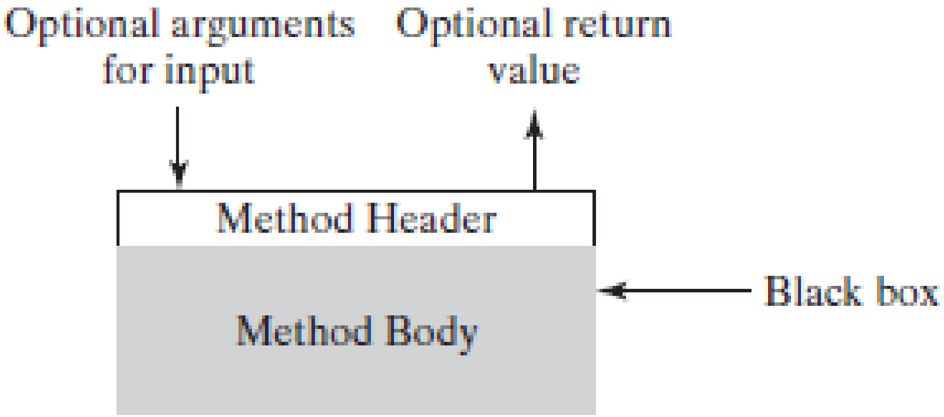
Using *this* to Invoke a Constructor Notes

- Java requires that the *this(arg-list)* statement appear first in the constructor before any other executable statements.
- If a class has multiple constructors, it is better to implement them using *this(arg-list)* as much as possible.
  - In general, a constructor with no or fewer arguments can invoke a constructor with more arguments using *this(arg-list)*.
  - This syntax often simplifies coding and makes the class easier to read and to maintain.

Method Abstraction and Stepwise Refinement

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

Method Abstraction and Stepwise Refinement (Cont.)



- You have already used the *System.out.print* method to display a string and the *max* method to find the maximum number.
- You know how to write the code to invoke these methods in your program, but as a user of these methods, you are not required to know how they are implemented.

Method Abstraction and Stepwise Refinement (Cont.)

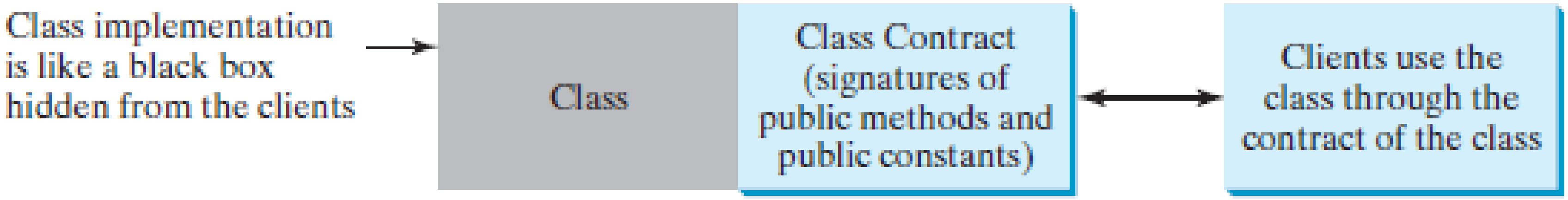
- The concept of method *abstraction* can be applied to the process of developing programs.
- When writing a large program, you can use the *divide-and-conquer* strategy, also known as *stepwise refinement*, to decompose it into subproblems.
  - The subproblems can be further decomposed into smaller, more manageable problems.

Class Abstraction and Encapsulation

- *Class abstraction* separates class implementation from how the class is used.
  - The creator of a class describes the functions of the class and lets the user know how the class can be used.
  - The collection of methods and fields that are accessible from outside the class, together with the description of how these members are expected to behave, serves as the *class's contract*.

Class Abstraction and Encapsulation (Cont.)

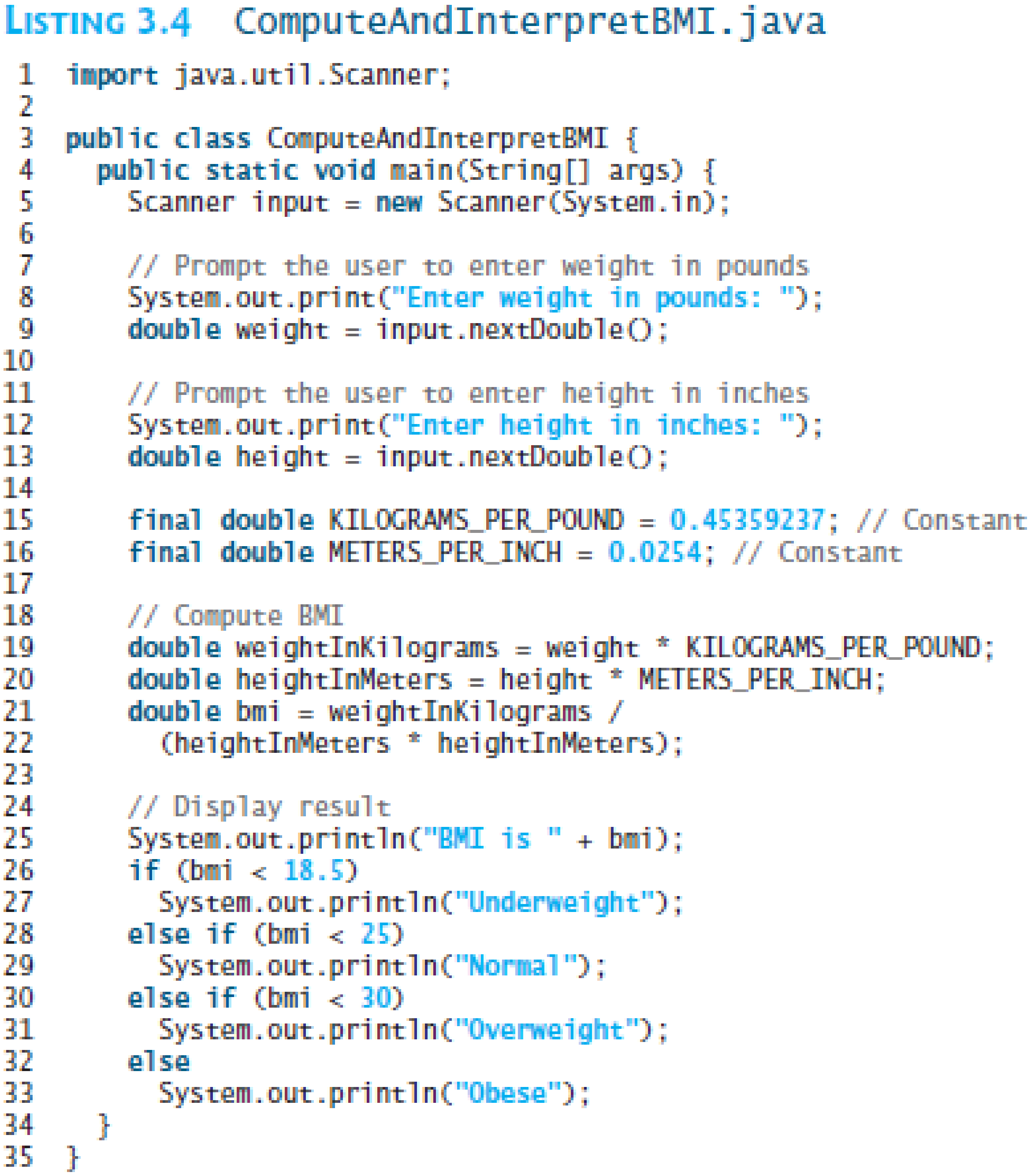
- The user of the class does not need to know how the class is implemented.
  - – The details of implementation are encapsulated and hidden from the user.
  - – This is called *class encapsulation*.
  - – For this reason, a class is also known as an *abstract data type (ADT)*.


Class implementation is like a black box hidden from the clients
Class
Class Contract (signatures of public methods and public constants)
Clients use the class through the contract of the class

Thinking in Objects

- The procedural paradigm focuses on designing methods.
- The object-oriented paradigm couples data and methods together into objects.
  - Software design using the object-oriented paradigm focuses on objects and operations on objects.
  - The object-oriented approach combines the power of the procedural paradigm with an added dimension that integrates data with operations into objects.
- In procedural programming, data and operations on the data are separate, and this methodology requires passing data to methods.
- Object-oriented programming places data and the operations that pertain to them in an object.

Procedural



LISTING 3.4 ComputeAndInterpretBMI.java

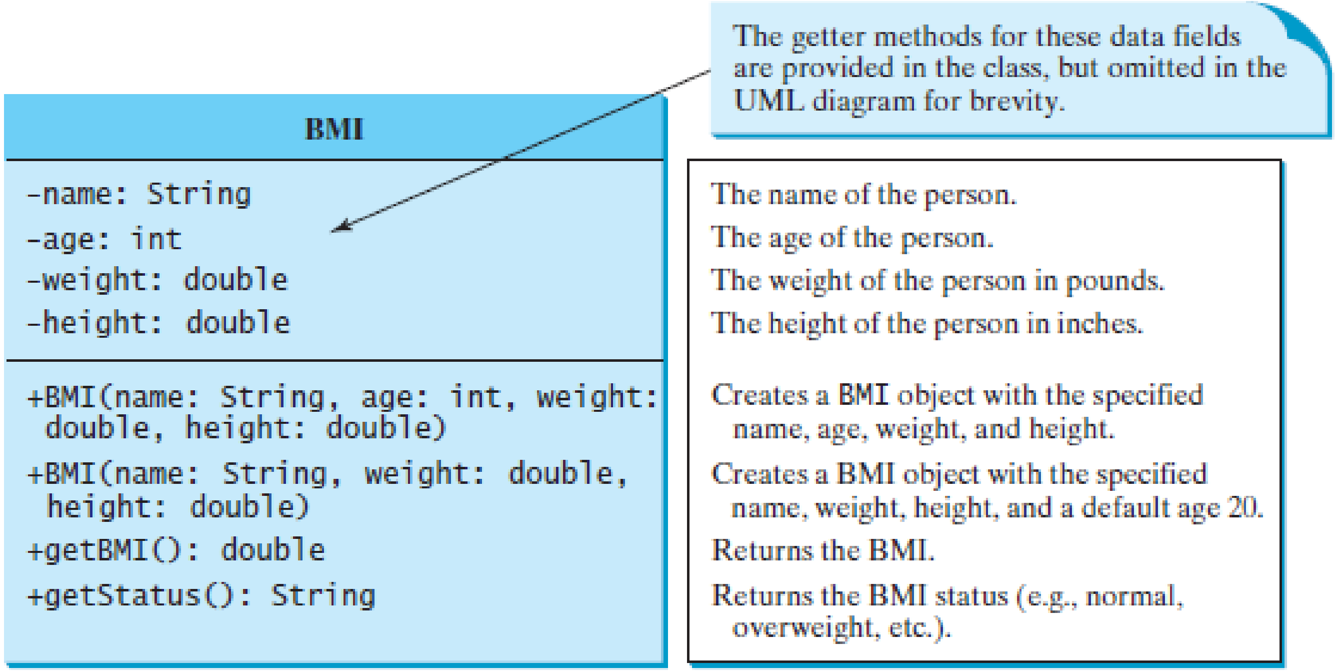
```java
import java.util.Scanner;

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

    // Prompt the user to enter weight in pounds
    System.out.print("Enter weight in pounds: ");
    double weight = input.nextDouble();

    // Prompt the user to enter height in inches
    System.out.print("Enter height in inches: ");
    double height = input.nextDouble();

    final double KILOGRAMS_PER_POUND = 0.45359237; // Constant
    final double METERS_PER_INCH = 0.0254; // Constant

    // Compute BMI
    double weightInKilograms = weight * KILOGRAMS_PER_POUND;
    double heightInMeters = height * METERS_PER_INCH;
    double bmi = weightInKilograms /
      (heightInMeters * heightInMeters);

    // Display result
    System.out.println("BMI is " + bmi);
    if (bmi < 18.5)
      System.out.println("Underweight");
    else if (bmi < 25)
      System.out.println("Normal");
    else if (bmi < 30)
      System.out.println("Overweight");
    else
      System.out.println("Obese");
  }
}
```

Thinking in Objects (Cont.)

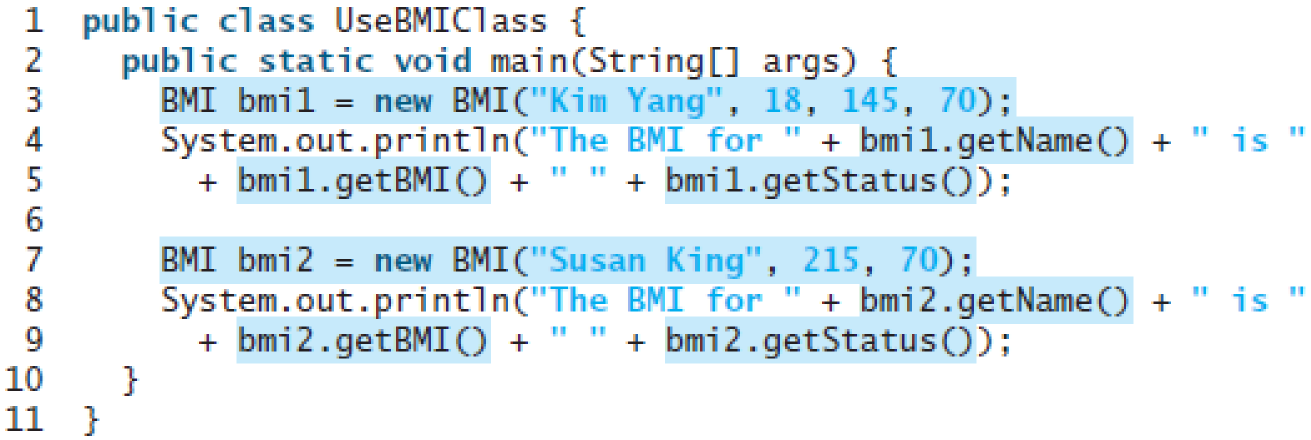
- The code cannot be reused in other programs, because the code is in the *main* method.
- To make it reusable, define a static method to compute body mass index as follows:

  public static double getBMI(double weight, double height)
- This method is useful for computing body mass index for a specified weight and height.
- However, it has limitations:
  - Suppose you need to associate the weight and height with a person's name and birth date.
  - You could declare separate variables to store these values, but these values would not be tightly coupled.
- The ideal way to couple them is to create an object that contains them all.
- Since these values are tied to individual objects, they should be stored in instance data fields.

Thinking in Objects (Cont.)



The getter methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.
BMI
-name: String
-age: int
-weight: double
-height: double
+BMI(name: String, age: int, weight: double, height: double)
+BMI(name: String, weight: double, height: double)
+getBMI(): double
+getStatus(): String
The name of the person.
The age of the person.
The weight of the person in pounds.
The height of the person in inches.
Creates a BMI object with the specified name, age, weight, and height.
Creates a BMI object with the specified name, weight, height, and a default age 20.
Returns the BMI.
Returns the BMI status (e.g., normal, overweight, etc.).



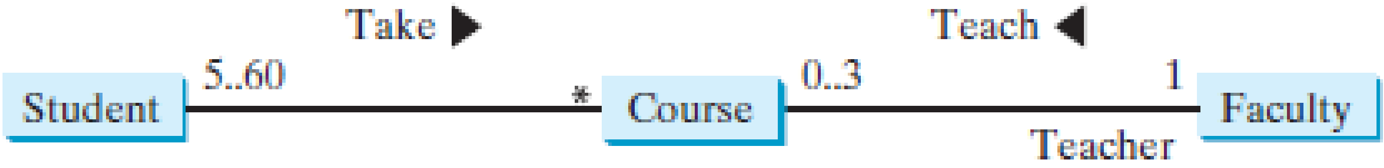
```java
public class UseBMIClass {
  public static void main(String[] args) {
    BMI bmi1 = new BMI("Kim Yang", 18, 145, 70);
    System.out.println("The BMI for " + bmi1.getName() + " is "
      + bmi1.getBMI() + " " + bmi1.getStatus());

    BMI bmi2 = new BMI("Susan King", 215, 70);
    System.out.println("The BMI for " + bmi2.getName() + " is "
      + bmi2.getBMI() + " " + bmi2.getStatus());
  }
}
```

Class Relationships

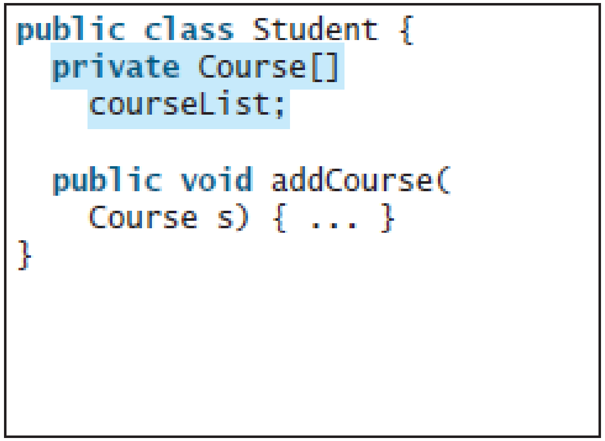
- To design classes, you need to explore the relationships among classes.
- The common relationships among classes are:
  - Association,
  - Aggregation and Composition, and
  - Inheritance.

Class Relationships
Association

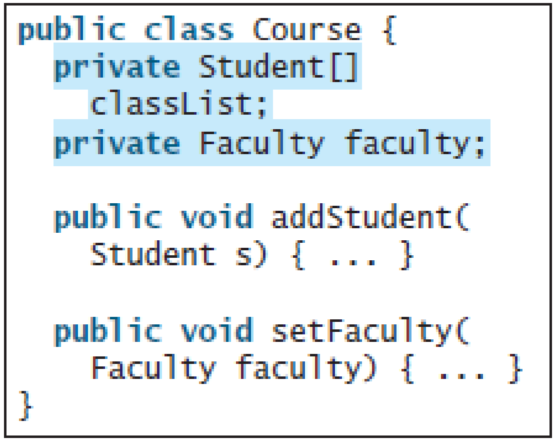
- *Association* is a general binary relationship that describes an activity between two classes.
- For example:
  - A student taking a course is an association between the *Student* class and the *Course* class.
  - A faculty member teaching a course is an association between the *Faculty* class and the *Course* class.



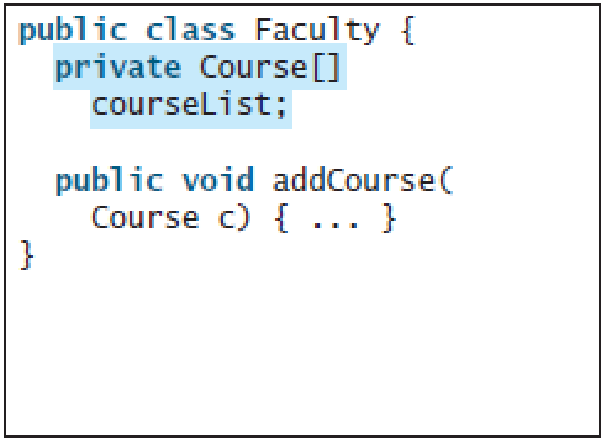
Class Relationships Association (Cont.)



```
public class Student {
  private Course[]
    courseList;

  public void addCourse(
    Course s) { ... }
}
```



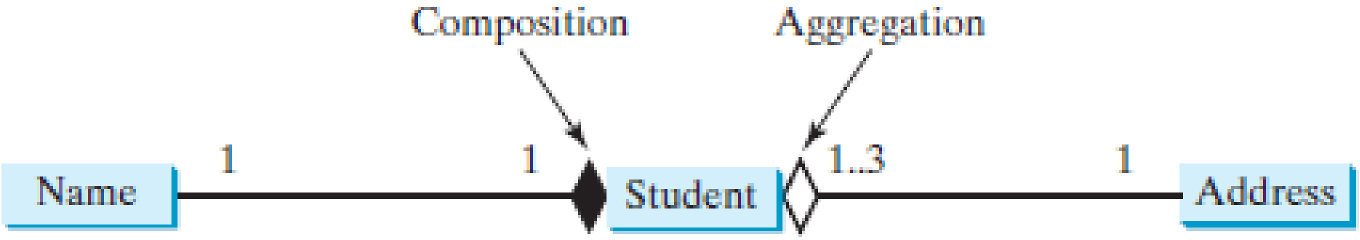
```
public class Course {
  private Student[]
    classList;
  private Faculty faculty;

  public void addStudent(
    Student s) { ... }

  public void setFaculty(
    Faculty faculty) { ... }
}
```



```
public class Faculty {
  private Course[]
    courseList;

  public void addCourse(
    Course c) { ... }
}
```

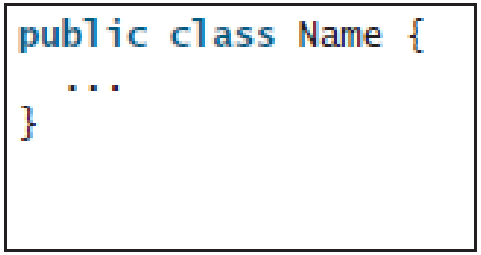
Class Relationships
Aggregation and Composition

- *Aggregation* is a special form of association that represents an ownership relationship between two objects.
- Aggregation models *has-a* relationships.
- An object can be owned by several other aggregating objects.
- If an object is exclusively owned by an aggregating object, the relationship between the object and its aggregating object is referred to as a *composition*.
- For example, "a student has a name" is a composition relationship between the *Student* class and the *Name* class, whereas "a student has an address" is an aggregation relationship between the *Student* class and the *Address* class, since an address can be shared by several students.



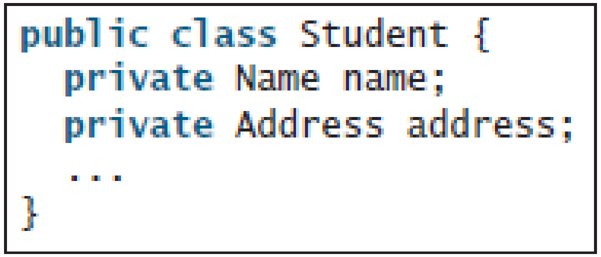
Class Relationships
Aggregation and Composition (Cont.)

- An *aggregation* relationship is usually represented as a data field in the aggregating class.
- Since aggregation and composition relationships are represented using classes in the same way, we will not differentiate them and call both compositions for simplicity.



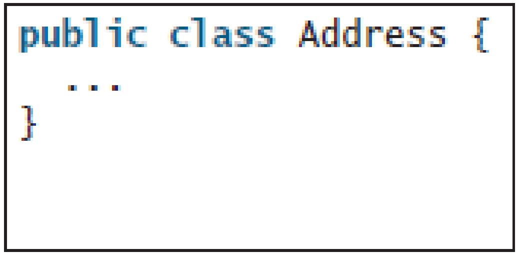
```
public class Name {
  ...
}
```

Aggregated class



```
public class Student {
  private Name name;
  private Address address;
  ...
}
```

Aggregating class

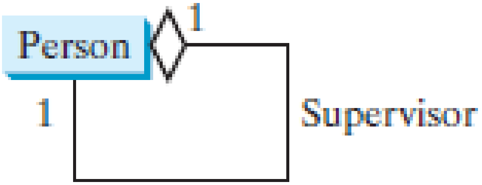


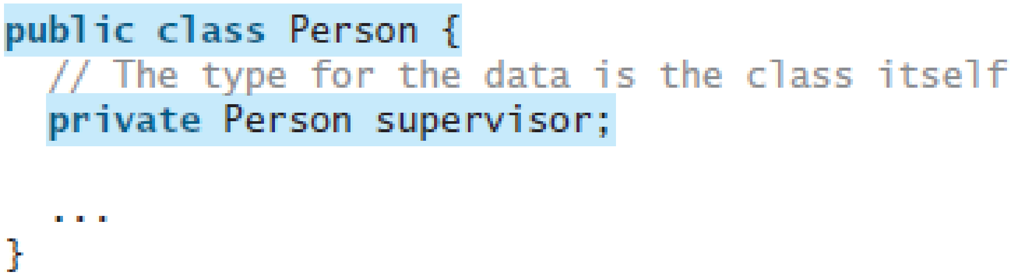
```
public class Address {
  ...
}
```

Aggregated class

Class Relationships
Aggregation and Composition (Cont.)

- Aggregation may exist between objects of the same class.
- In the relationship “a person has a supervisor,” a supervisor can be represented as a data field in the *Person* class.

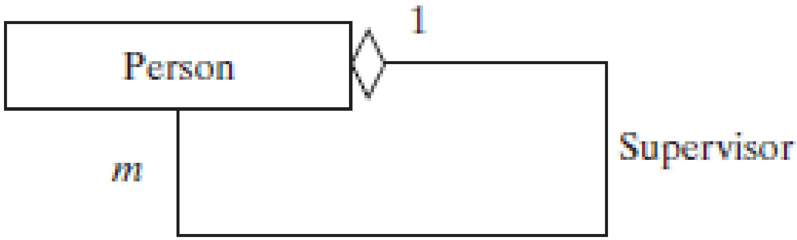




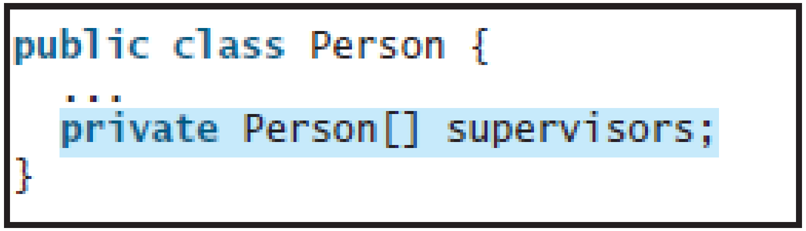
```
public class Person {
  // The type for the data is the class itself
  private Person supervisor;

  ...
}
```

Class Relationships
Aggregation and Composition (Cont.)

- If a person can have several supervisors, you may use an array to store supervisors.



(a)



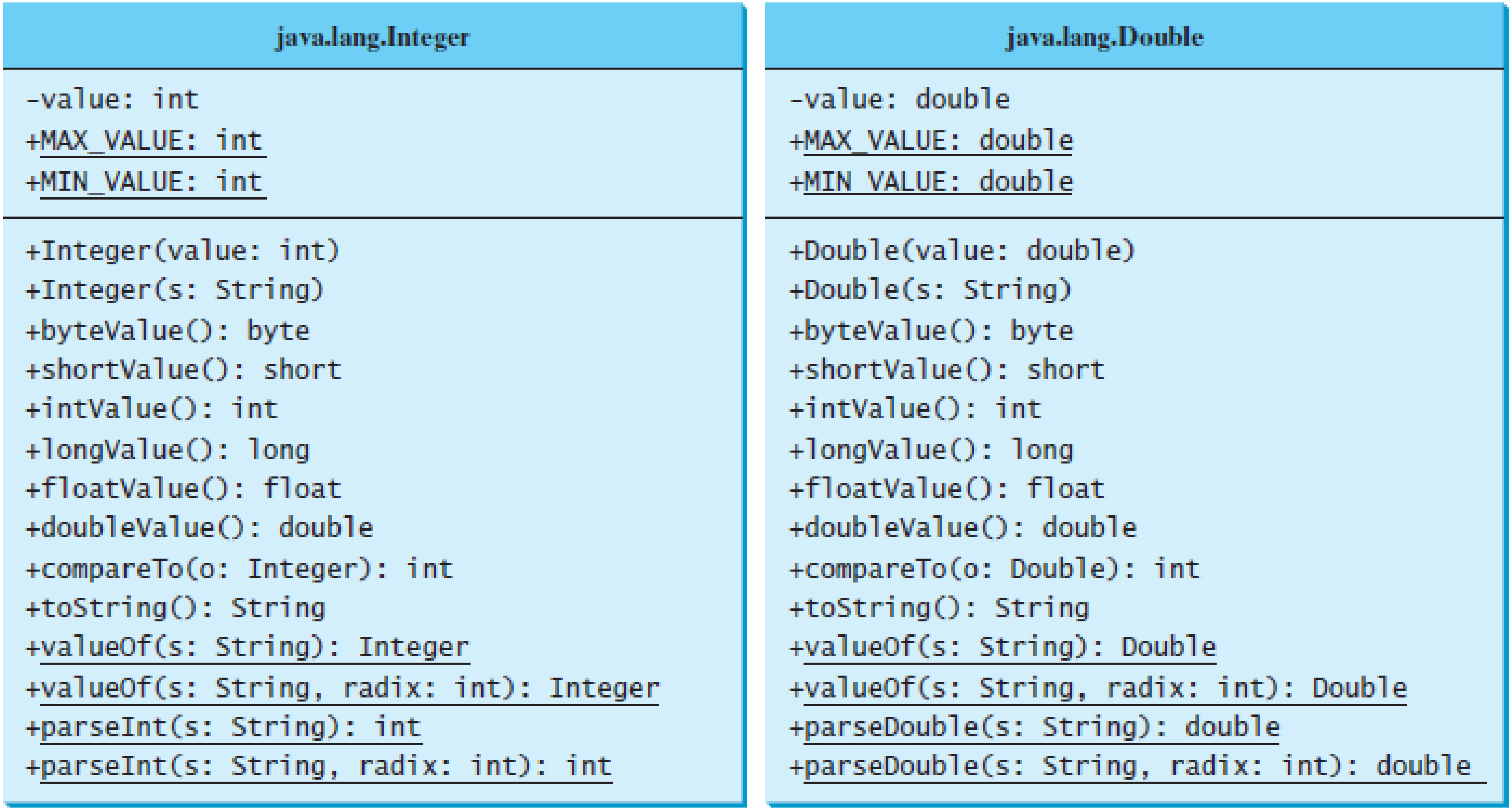
```
public class Person {
  ...
  private Person[] supervisors;
}
```

(b)

Processing Primitive Data Type Values as Objects

- Owing to performance considerations, primitive data type values are not objects in Java.
  - Because of the overhead of processing objects, the language's performance would be adversely affected if primitive data type values were treated as objects.
- However, many Java methods require the use of objects as arguments. Java offers a convenient way to incorporate, or *wrap*, a primitive data type into an object
- Wrapping *int* into the *Integer* class, wrapping *double* into the *Double* class, and wrapping *char* into the *Character* class.
- By using a wrapper class, you can process primitive data type values as objects.
  - Java provides *Boolean*, *Character*, *Double*, *Float*, *Byte*, *Short*, Integer, and *Long* wrapper classes in the *java.lang* package for primitive data types.

Processing Primitive Data Type Values as Objects (Cont.)



java.lang.Integer

-value: int
+MAX_VALUE: int
+MIN_VALUE: int

+Integer(value: int)
+Integer(s: String)
+byteValue(): byte
+shortValue(): short
+intValue(): int
+longValue(): long
+floatValue(): float
+doubleValue(): double
+compareTo(o: Integer): int
+toString(): String
+valueOf(s: String): Integer
+valueOf(s: String, radix: int): Integer
+parseInt(s: String): int
+parseInt(s: String, radix: int): int

java.lang.Double

-value: double
+MAX_VALUE: double
+MIN_VALUE: double

+Double(value: double)
+Double(s: String)
+byteValue(): byte
+shortValue(): short
+intValue(): int
+longValue(): long
+floatValue(): float
+doubleValue(): double
+compareTo(o: Double): int
+toString(): String
+valueOf(s: String): Double
+valueOf(s: String, radix: int): Double
+parseDouble(s: String): double
+parseDouble(s: String, radix: int): double

Processing Primitive Data Type Values as Objects (Cont.)

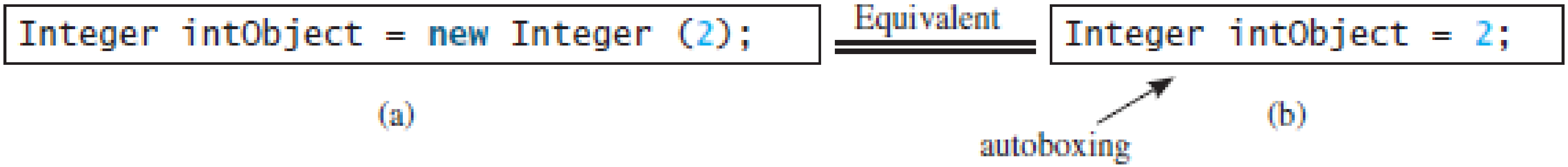
- You can construct a wrapper object either from a primitive data type value or from a string representing the numeric value.
  - For example, *new Double(5.0)*, *new Double("5.0")*, *new Integer(5)*, and *new Integer("5")*.
- The wrapper classes do not have no-arg constructors.
- The instances of all wrapper classes are immutable; this means that, once the objects are created, their internal values cannot be changed.
- Each numeric wrapper class has the constants *MAX_VALUE* and *MIN_VALUE*.
- Each numeric wrapper class contains the methods *doubleValue()*, *floatValue()*, *intValue()*, *longValue()*, and *shortValue()* for returning a *double*, *float*, *int*, *long*, or short value for the wrapper object.
  - *new Double(12.4).intValue()* returns *12*;
  - *new Integer(12).doubleValue()* returns *12.0*;

Processing Primitive Data Type Values as Objects (Cont.)

- The numeric wrapper classes have the static method, *valueOf (String s)*.
  - This method creates a new object initialized to the value represented by the specified string.
  - *Double doubleObject = Double.valueOf("**12.4**");*
  - *Integer integerObject = Integer.valueOf("**12**");*
- The static method *parseInt* is used to parse a numeric string into an *int* value and the *parseDouble* method in the *Double* class to parse a numeric string into a *double* value.
- Each numeric wrapper class has two overloaded parsing methods to parse a numeric string into an appropriate numeric value based on **10** (decimal) or any specified radix (e.g., **2** for binary, **8** for octal, and **16** for hexadecimal).
  - *Integer.parseInt("11", 2)* returns **3**;
  - *Integer.parseInt("12", 8)* returns **10**;
  - *Integer.parseInt("13", 10)* returns **13**;
  - *Integer.parseInt("1A", 16)* returns **26**;

Processing Primitive Data Type Values as Objects (Cont.)

- Converting a primitive value to a wrapper object is called *boxing*.
- The reverse conversion is called *unboxing*.
- Java allows primitive types and wrapper classes to be converted automatically.
  - The compiler will automatically box a primitive value that appears in a context requiring an object, and will unbox an object that appears in a context requiring a primitive value.
  - This is called *autoboxing* and *autounboxing*.

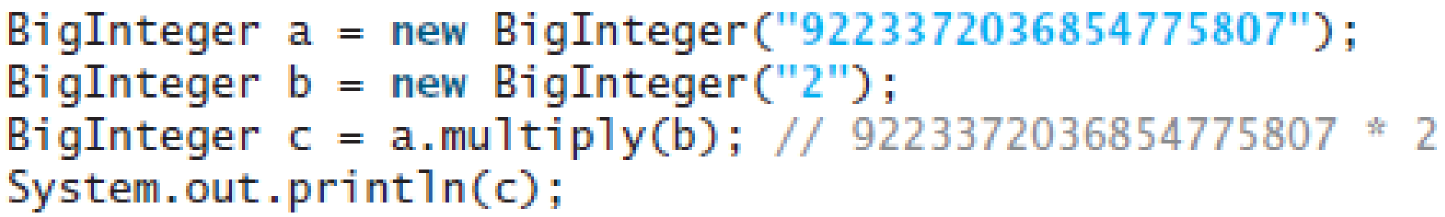


```
Integer intObject = new Integer (2);
```

```
Integer intObject = 2;
```

The *BigInteger* and *BigDecimal* Classes

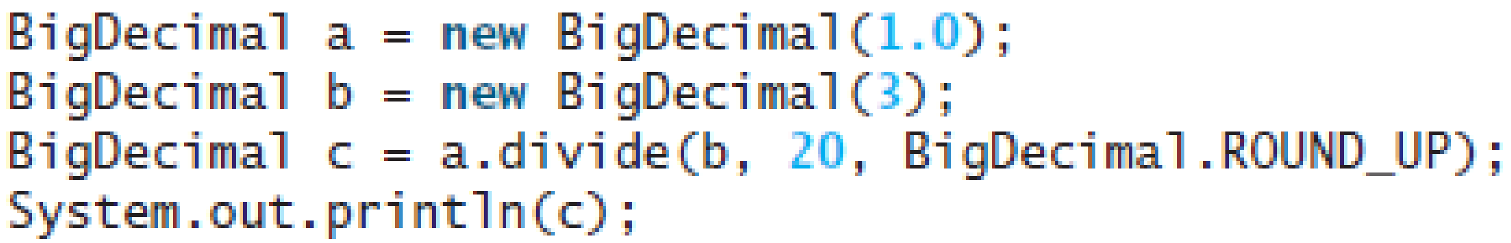
- The *BigInteger* and BigDecimal classes can be used to represent integers or decimal numbers of any size and precision.
  - If you need to compute with very large integers or high-precision floating-point values, you can use the *BigInteger* and *BigDecimal* classes in the *java.math* package.
  - Both are *immutable*.
- You can use *new BigInteger(String)* and *new BigDecimal(String)* to create an instance of *BigInteger* and *BigDecimal*.
- You can use the *add*, *subtract*, *multiply*, *divide*, and *remainder* methods to perform arithmetic operations.
- The largest integer of the *long* type is *Long.MAX_VALUE* (i.e., **9223372036854775807**). An instance of *BigInteger* can represent an integer of any size.



```
BigInteger a = new BigInteger("9223372036854775807");
BigInteger b = new BigInteger("2");
BigInteger c = a.multiply(b); // 9223372036854775807 * 2
System.out.println(c);
```

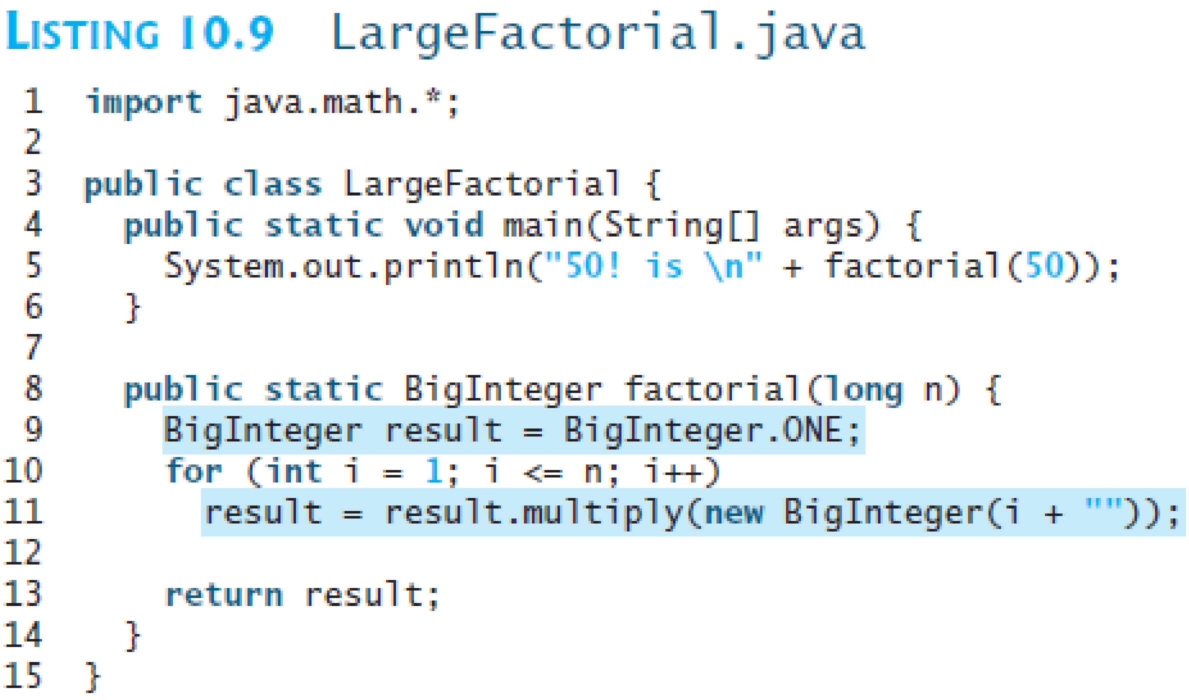
The *BigInteger* and *BigDecimal* Classes (Cont.)

- There is no limit to the precision of a *BigDecimal* object.
- The *divide* method may throw an *ArithmeticException* if the result cannot be terminated.
- However, you can use the overloaded *divide(BigDecimal d, int scale, int roundingMode)* method to specify a scale and a rounding mode to avoid this exception, where *scale* is the maximum number of digits after the decimal point.
- For example, the following code creates two *BigDecimal* objects and performs division with scale **20** and rounding mode *BigDecimal.ROUND_UP*.
  - The output is **0.3333333333333333334**.



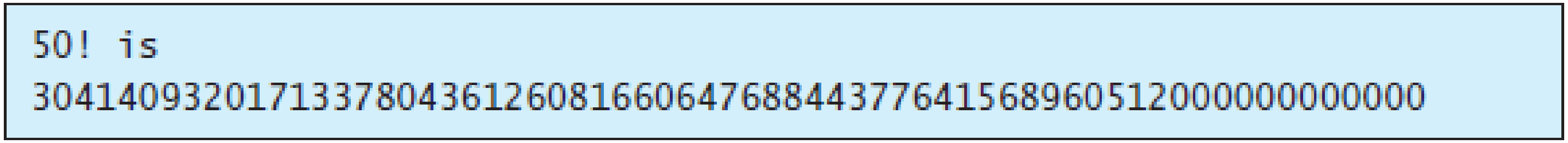
```
BigDecimal a = new BigDecimal(1.0);
BigDecimal b = new BigDecimal(3);
BigDecimal c = a.divide(b, 20, BigDecimal.ROUND_UP);
System.out.println(c);
```

The *BigInteger* and *BigDecimal* Classes (Cont.)



LISTING 10.9 LargeFactorial.java

```java
import java.math.*;

public class LargeFactorial {
  public static void main(String[] args) {
    System.out.println("50! is \n" + factorial(50));
  }

  public static BigInteger factorial(long n) {
    BigInteger result = BigInteger.ONE;
    for (int i = 1; i <= n; i++)
      result = result.multiply(new BigInteger(i + ""));

    return result;
  }
}
```



```
50! is
30414093201713378043612608166064768844377641568960512000000000000
```

Object-Oriented Problem Solving

Inheritance & Polymorphism

Based on Chapter 11 of "Introduction to Java Programming" by Y. Daniel Liang.

Eng. Asma Abdel Karim
Computer Engineering Department

Outline

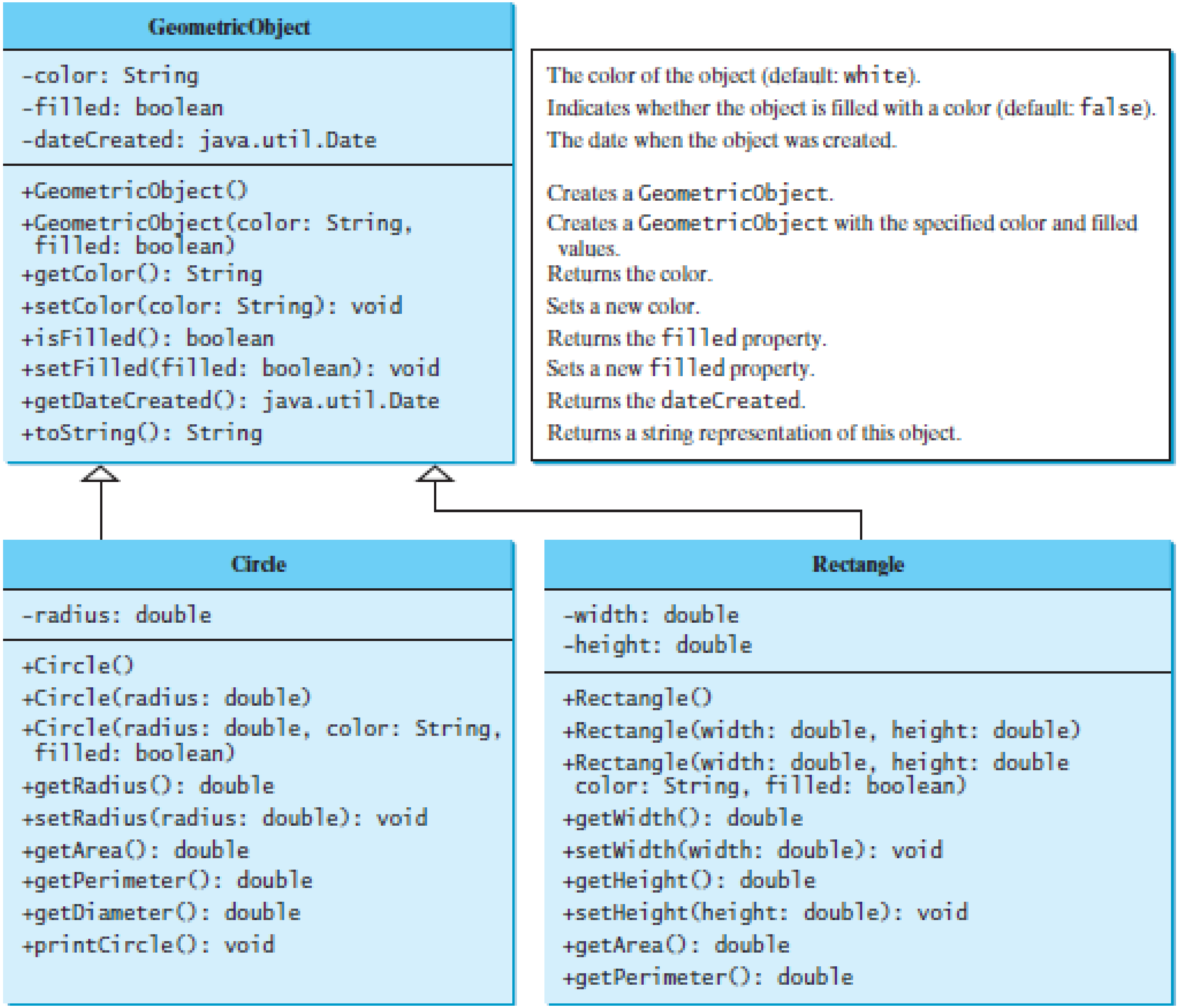
- Superclasses and Subclasses (11.2)
- Using the Super keyword (11.3)
- Overriding Methods (11.4)
- Overriding vs. Overloading (11.5)
- The Object Class and its toString() (11.6)
- Polymorphism (11.7)
- Dynamic Binding (11.8)
- Casting Objects (11.9)
- The Object's equals Method (11.10)
- The Protected Data and Methods (11.14)
- Preventing Extending and Overriding (11.15)

Superclasses and Subclasses

- Classes are used to model objects of the same type.
- Different classes may have some common properties and behaviors.
- *Inheritance* allows you to:
  - Define a generalized class that includes the common properties and behavior.
  - Define specialized classes that extend the generalized class.
    - *Inherit* the properties and methods from the general class.
    - Add new properties and methods.

Superclasses and Subclasses (Cont.)

- In Java terminology, a class *C1* extended from another class *C2* is called a *subclass*, and *C2* is called a *superclass*.
  - A superclass is also referred to as a *parent class* or a *base class*, and a subclass as a *child class*, an *extended class*, or a *derived class*.
- A subclass:
  - inherits accessible data fields and methods from its superclass and,
  - may also add new data fields and methods.



GeometricObject

```
-color: String
-filled: boolean
-dateCreated: java.util.Date

+GeometricObject()
+GeometricObject(color: String,
  filled: boolean)
+getColor(): String
+setColor(color: String): void
+isFilled(): boolean
+setFilled(filled: boolean): void
+getDateCreated(): java.util.Date
+toString(): String
```

The color of the object (default: white).
Indicates whether the object is filled with a color (default: false).
The date when the object was created.

Creates a GeometricObject.
Creates a GeometricObject with the specified color and filled values.
Returns the color.
Sets a new color.
Returns the filled property.
Sets a new filled property.
Returns the dateCreated.
Returns a string representation of this object.

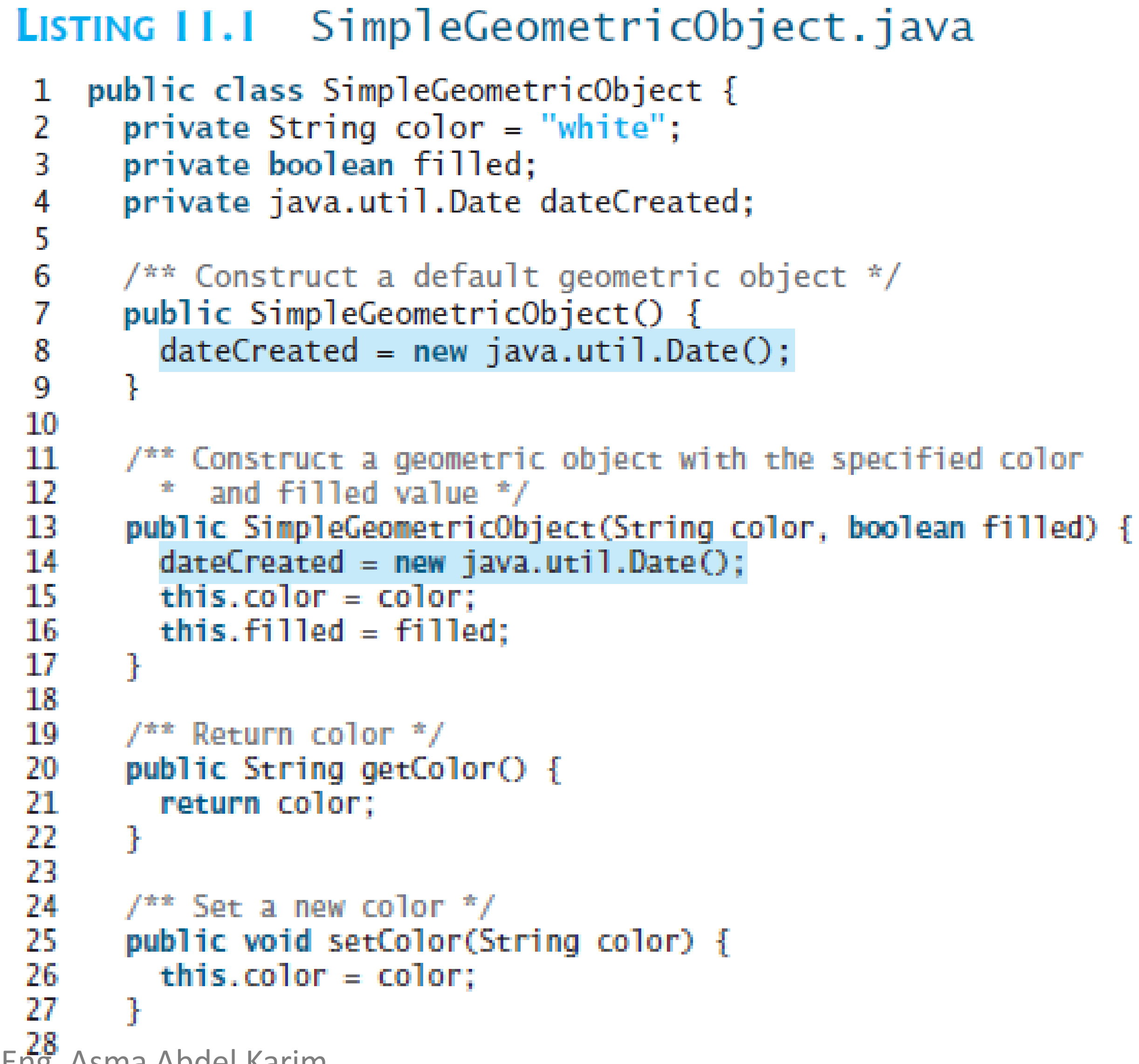
Circle

```
-radius: double

+Circle()
+Circle(radius: double)
+Circle(radius: double, color: String,
  filled: boolean)
+getRadius(): double
+setRadius(radius: double): void
+getArea(): double
+getPerimeter(): double
+getDiameter(): double
+printCircle(): void
```

Rectangle

```
-width: double
-height: double

+Rectangle()
+Rectangle(width: double, height: double)
+Rectangle(width: double, height: double
  color: String, filled: boolean)
+getWidth(): double
+setWidth(width: double): void
+getHeight(): double
+setHeight(height: double): void
+getArea(): double
+getPerimeter(): double
```

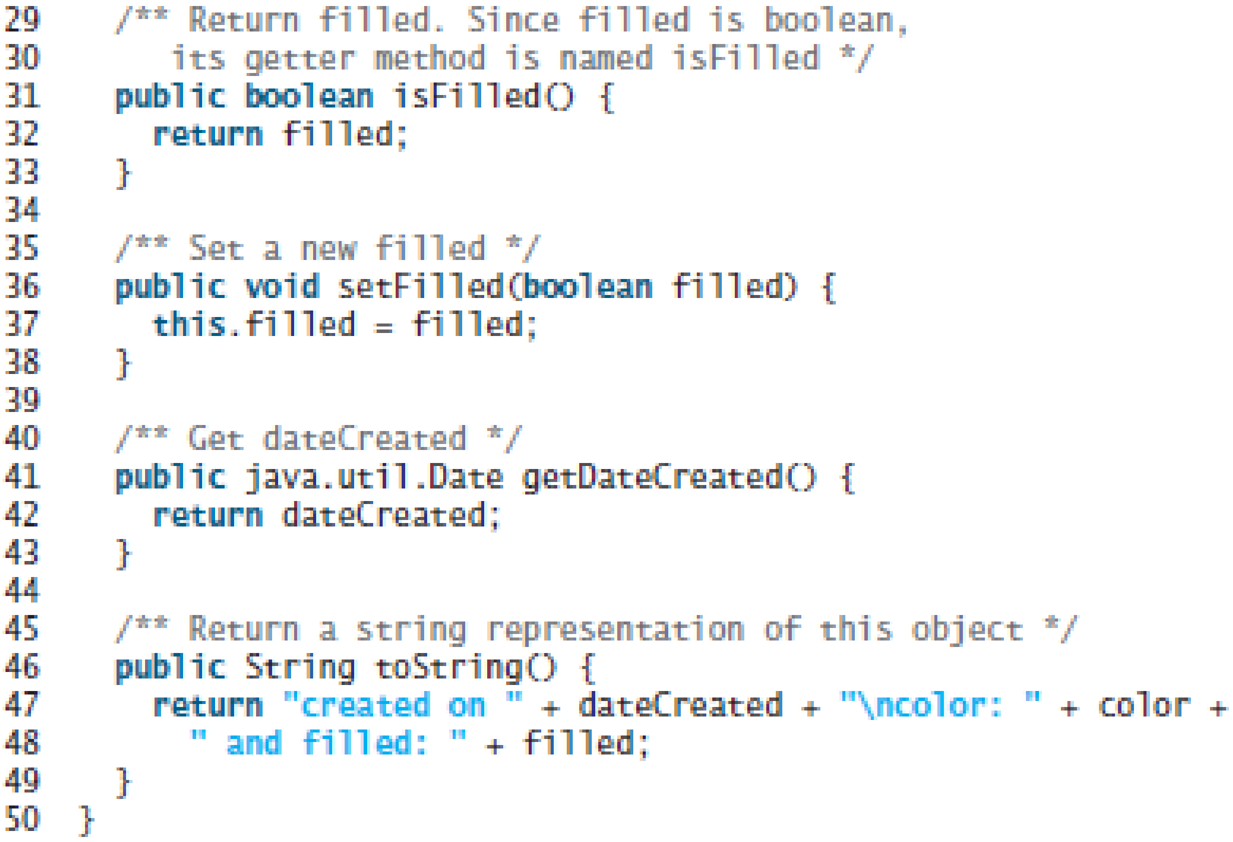
GeometricObject Class



LISTING 11.1 SimpleGeometricObject.java

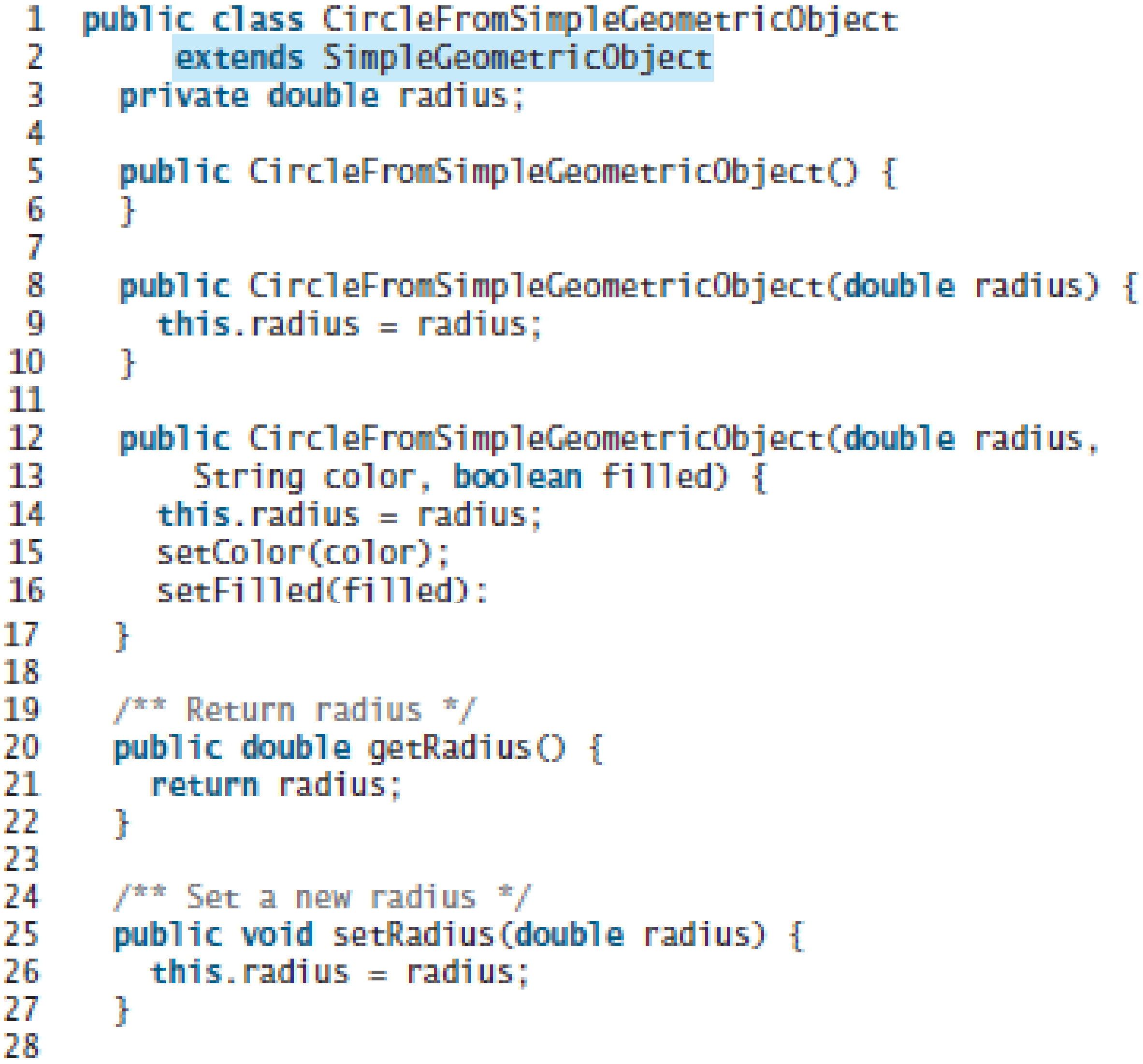
```java
public class SimpleGeometricObject {
  private String color = "white";
  private boolean filled;
  private java.util.Date dateCreated;

  /** Construct a default geometric object */
  public SimpleGeometricObject() {
    dateCreated = new java.util.Date();
  }

  /** Construct a geometric object with the specified color
    *  and filled value */
  public SimpleGeometricObject(String color, boolean filled) {
    dateCreated = new java.util.Date();
    this.color = color;
    this.filled = filled;
  }

  /** Return color */
  public String getColor() {
    return color;
  }

  /** Set a new color */
  public void setColor(String color) {
    this.color = color;
  }

```

GeometricObject Class (Cont.)



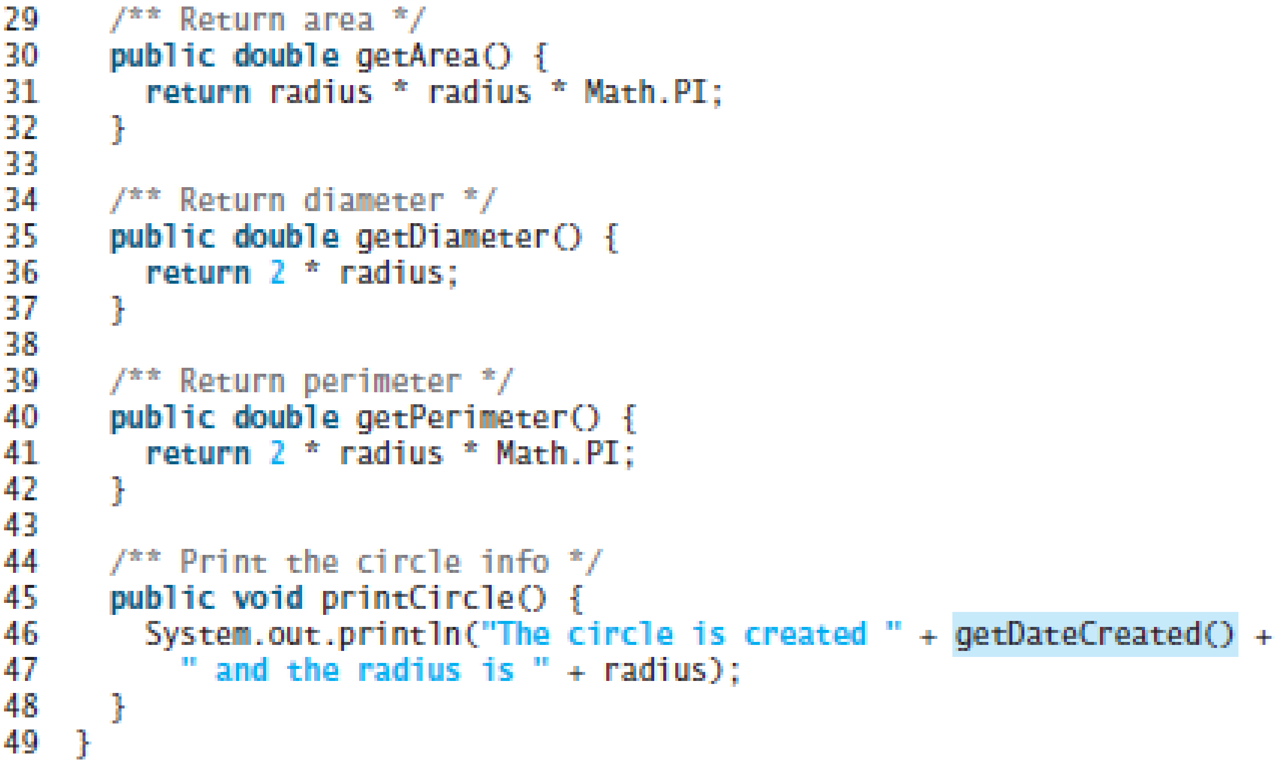
```java
  /** Return filled. Since filled is boolean,
     its getter method is named isFilled */
  public boolean isFilled() {
    return filled;
  }

  /** Set a new filled */
  public void setFilled(boolean filled) {
    this.filled = filled;
  }

  /** Get dateCreated */
  public java.util.Date getDateCreated() {
    return dateCreated;
  }

  /** Return a string representation of this object */
  public String toString() {
    return "created on " + dateCreated + "\ncolor: " + color +
      " and filled: " + filled;
  }
}
```

Circle Class



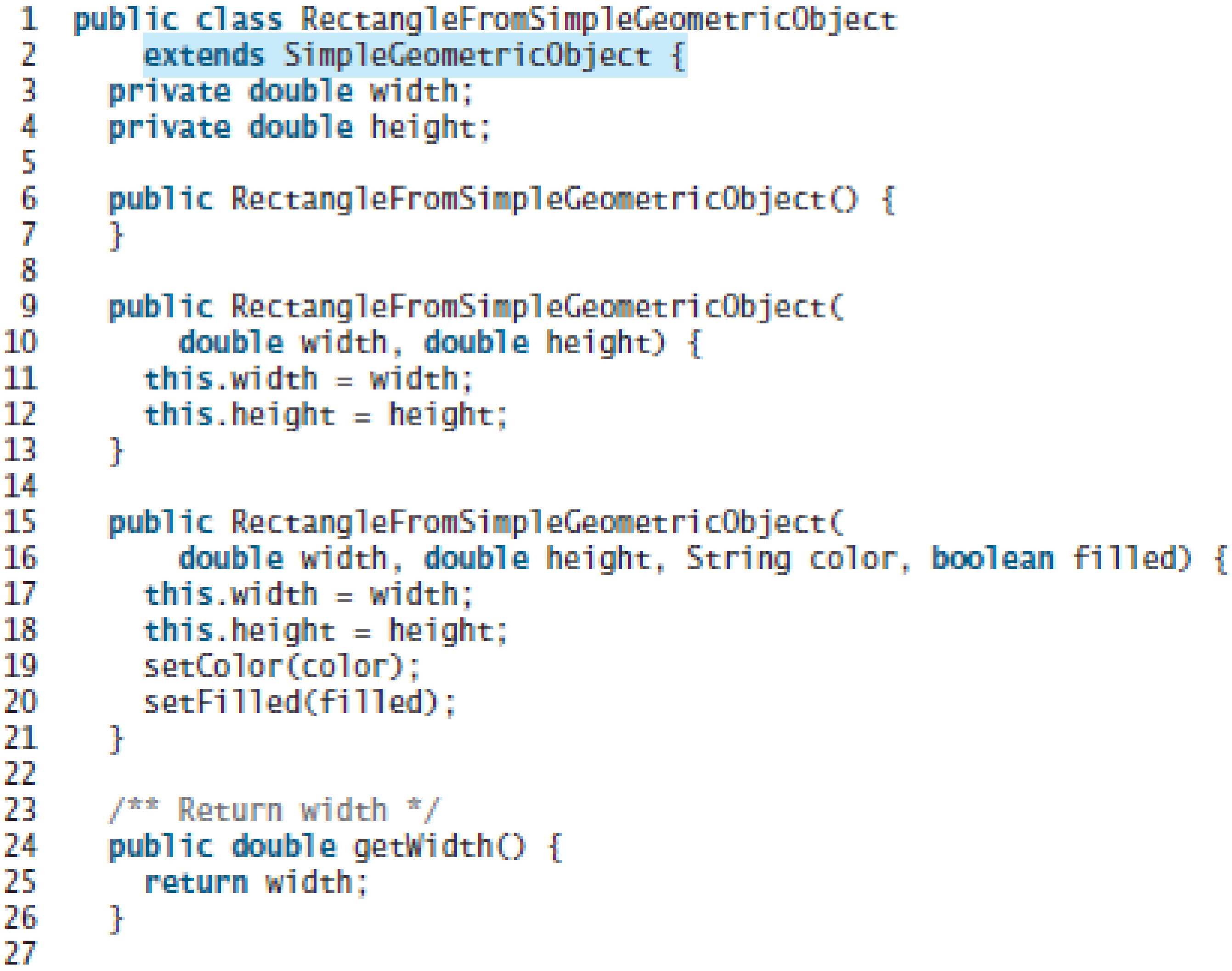
```
public class CircleFromSimpleGeometricObject
    extends SimpleGeometricObject {
  private double radius;

  public CircleFromSimpleGeometricObject() {
  }

  public CircleFromSimpleGeometricObject(double radius) {
    this.radius = radius;
  }

  public CircleFromSimpleGeometricObject(double radius,
      String color, boolean filled) {
    this.radius = radius;
    setColor(color);
    setFilled(filled);
  }

  /** Return radius */
  public double getRadius() {
    return radius;
  }

  /** Set a new radius */
  public void setRadius(double radius) {
    this.radius = radius;
  }

```

Circle Class (Cont.)



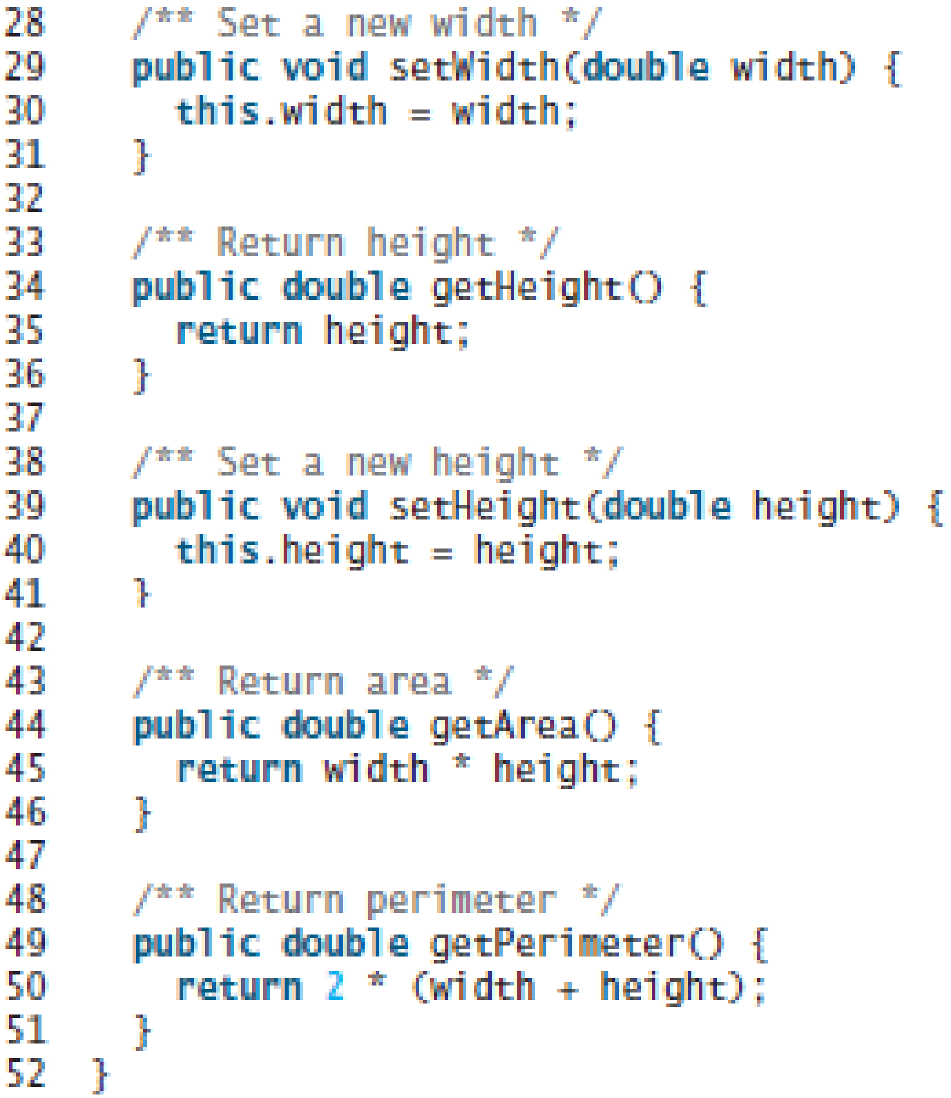
```
  /** Return area */
  public double getArea() {
    return radius * radius * Math.PI;
  }

  /** Return diameter */
  public double getDiameter() {
    return 2 * radius;
  }

  /** Return perimeter */
  public double getPerimeter() {
    return 2 * radius * Math.PI;
  }

  /** Print the circle info */
  public void printCircle() {
    System.out.println("The circle is created " + getDateCreated() +
      " and the radius is " + radius);
  }
}
```

Rectangle Class



```
public class RectangleFromSimpleGeometricObject
    extends SimpleGeometricObject {
  private double width;
  private double height;

  public RectangleFromSimpleGeometricObject() {
  }

  public RectangleFromSimpleGeometricObject(
      double width, double height) {
    this.width = width;
    this.height = height;
  }

  public RectangleFromSimpleGeometricObject(
      double width, double height, String color, boolean filled) {
    this.width = width;
    this.height = height;
    setColor(color);
    setFilled(filled);
  }

  /** Return width */
  public double getWidth() {
    return width;
  }

```

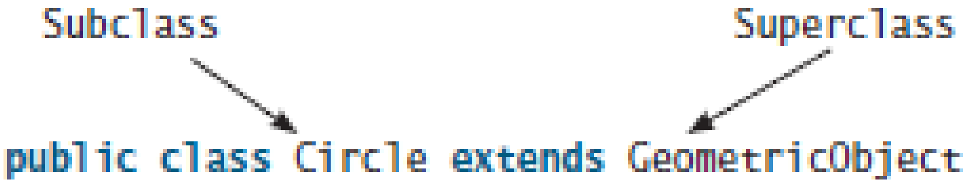
Rectangle Class (Cont.)



```
  /** Set a new width */
  public void setWidth(double width) {
    this.width = width;
  }

  /** Return height */
  public double getHeight() {
    return height;
  }

  /** Set a new height */
  public void setHeight(double height) {
    this.height = height;
  }

  /** Return area */
  public double getArea() {
    return width * height;
  }

  /** Return perimeter */
  public double getPerimeter() {
    return 2 * (width + height);
  }
}
```

Comments

- The *Circle* class extends the *GeometricObject* using the following syntax:

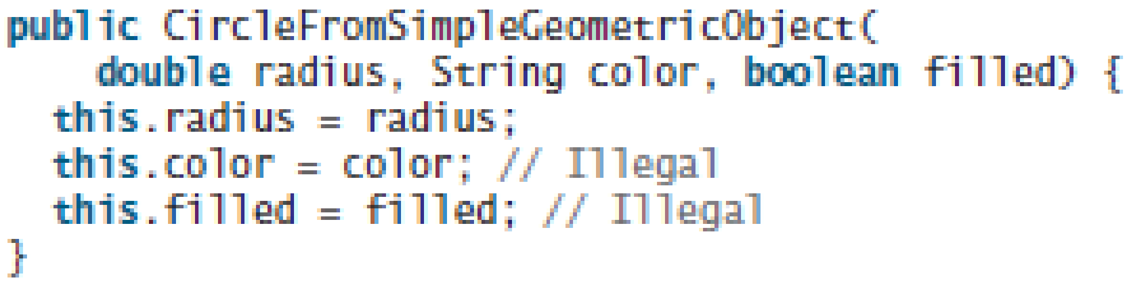


```
public class Circle extends GeometricObject
```

- The keyword *extends* tells the compiler that the *Circle* class extends the *GeometricObject* class, thus inheriting the methods *getColor*, *setColor*, *isFilled*, *setFilled*, and *toString*.
- The overloaded constructor *Circle(double radius, String color, boolean filled)* is implemented by invoking the *setColor* and *setFilled* methods to set the *color* and *filled* properties.
  - These two public methods are defined in the superclass *GeometricObject* and are inherited in *Circle*, so they can be used in the *Circle* class.

Comments (Cont.)

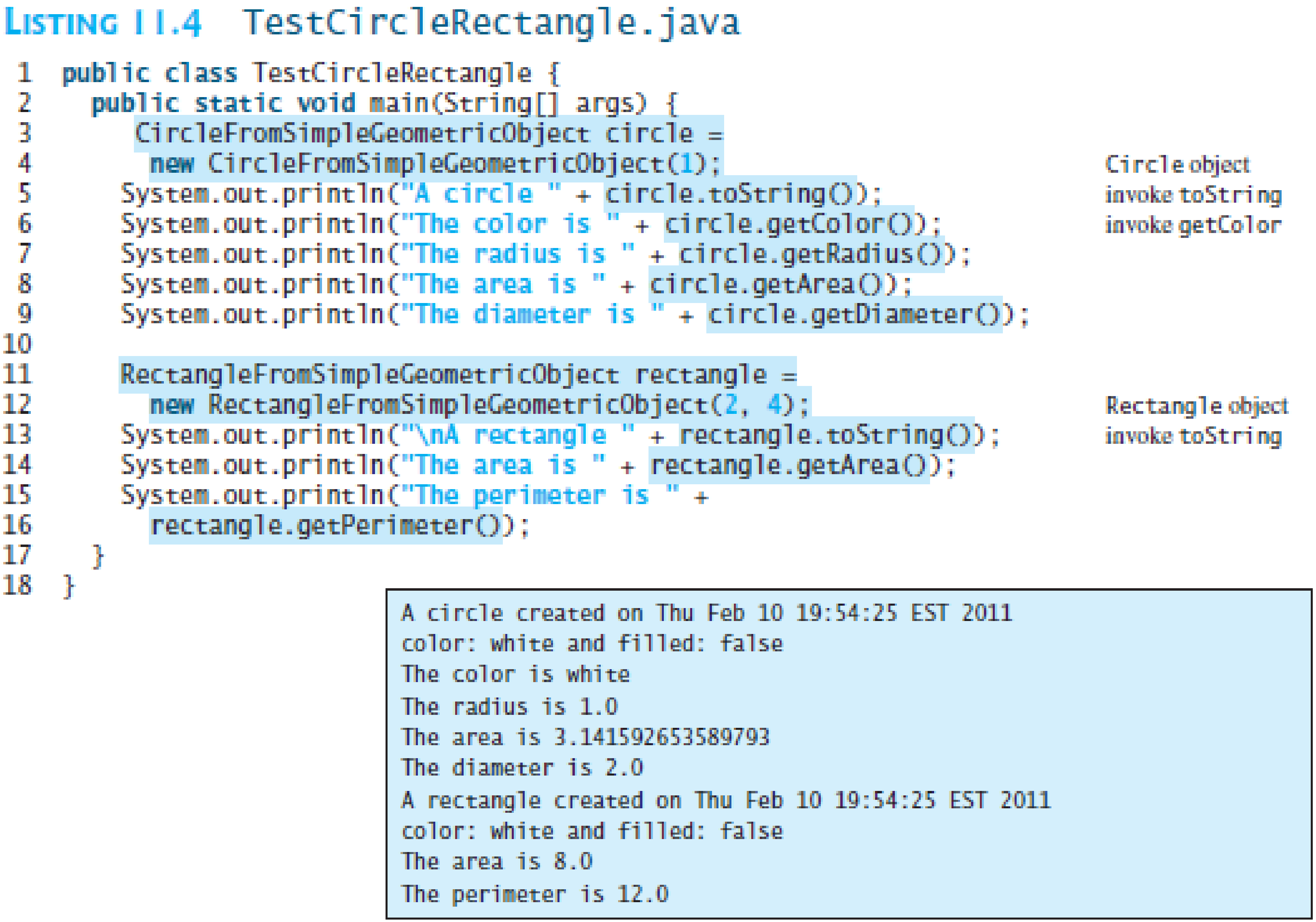
- You might attempt to use the data fields *color* and *filled* directly in the constructor as follows:



```
public CircleFromSimpleGeometricObject(
    double radius, String color, boolean filled) {
  this.radius = radius;
  this.color = color; // Illegal
  this.filled = filled; // Illegal
}
```

- This is wrong, because the private data fields *color* and *filled* in the *GeometricObject* class cannot be accessed in any class other than in the *GeometricObject* class itself.
  - The only way to read and modify *color* and *filled* is through their getter and setter methods.

TestCircleRectangle.java



LISTING 11.4 TestCircleRectangle.java

```java
public class TestCircleRectangle {
  public static void main(String[] args) {
    CircleFromSimpleGeometricObject circle =
      new CircleFromSimpleGeometricObject(1);
    System.out.println("A circle " + circle.toString());
    System.out.println("The color is " + circle.getColor());
    System.out.println("The radius is " + circle.getRadius());
    System.out.println("The area is " + circle.getArea());
    System.out.println("The diameter is " + circle.getDiameter());

    RectangleFromSimpleGeometricObject rectangle =
      new RectangleFromSimpleGeometricObject(2, 4);
    System.out.println("\nA rectangle " + rectangle.toString());
    System.out.println("The area is " + rectangle.getArea());
    System.out.println("The perimeter is " +
      rectangle.getPerimeter());
  }
}
```

Circle object
invoke toString
invoke getColor

Rectangle object
invoke toString

```
A circle created on Thu Feb 10 19:54:25 EST 2011
color: white and filled: false
The color is white
The radius is 1.0
The area is 3.141592653589793
The diameter is 2.0
A rectangle created on Thu Feb 10 19:54:25 EST 2011
color: white and filled: false
The area is 8.0
The perimeter is 12.0
```

Important Notes Regarding Inheritance (1)

- Contrary to conventional interpretation, a subclass is not a subset of its superclass.
  - In fact, a subclass usually contains more information and methods than its superclass.
- Private data fields in a superclass are not accessible outside the class.
  - They cannot be used directly in a subclass.
  - They can only be accessed/mutated through public accessors/mutators if defined in the superclass.

Important Notes Regarding Inheritance (2)

- Inheritance is used to model the *is-a* relationship.
  - Do not blindly extend a class just for the sake of reusing methods.
  - For example, it makes no sense for a *Tree* class to extend a *Person* class, even though they share common properties such as height and weight.
- Some programming languages allow you to derive a subclass from several classes.
  - This capability is called *multiple inheritance*.
  - Java does not allow multiple inheritance.
    - A Java class may inherit directly from only one class.
  - Multiple inheritance can be achieved through interfaces in Java.

The *Super* Keyword

- The keyword *super* refers to the superclass and can be used to:
  - Call a superclass constructor.
  - Call a superclass method.

Using the *Super* Keyword to Call a Superclass Constructor

- Remember that a constructor is used to construct an instance of a class.
- Unlike properties and methods, the constructors of a superclass are not inherited by a subclass.
  - They can only be invoked from the constructors of the subclasses using the keyword *super*.
- The syntax to call a superclass's constructor is:
  - *super()*, or *super(arguments)*;
  - The statement *super()* invokes the no-arg constructor of its superclass.
  - The statement *super(arguments)* invokes the superclass constructor that matches the *arguments*.

Using the *Super* Keyword to Call a Superclass Constructor: Example

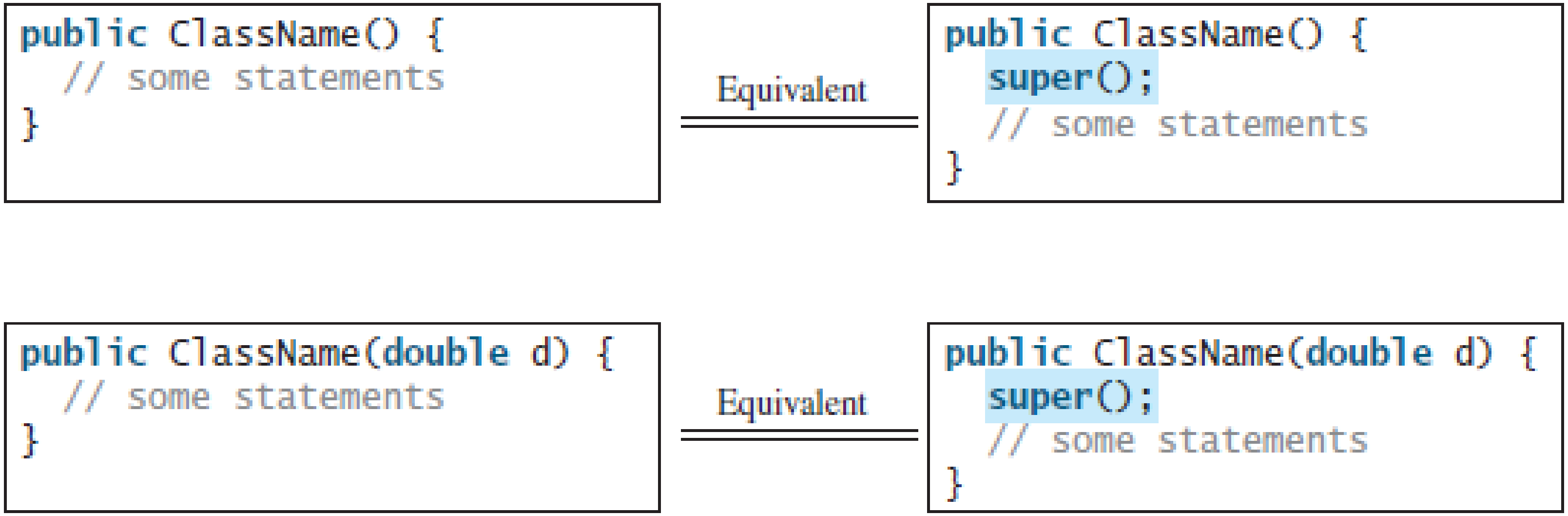
- The statement *super()* or *super(parameters)* must appear in the first line of the subclass's constructor.
- The following constructor can be added to the *Circle* class of the previous example:

```
public Circle (double radius){
    super();
    this.radius = radius;
}
```

Invokes the no-arg constructor, which is the default constructor of the GeometricObject class.

Constructor Chaining

- A constructor may invoke an overloaded constructor (using *this*) or its superclass constructor (using *super*).
- If neither is invoked explicitly, the compiler automatically puts *super()* as the first statement in the constructor.



```
public ClassName() {
  // some statements
}
```

Equivalent

```
public ClassName() {
  super();
  // some statements
}
```

```
public ClassName(double d) {
  // some statements
}
```

Equivalent

```
public ClassName(double d) {
  super();
  // some statements
}
```

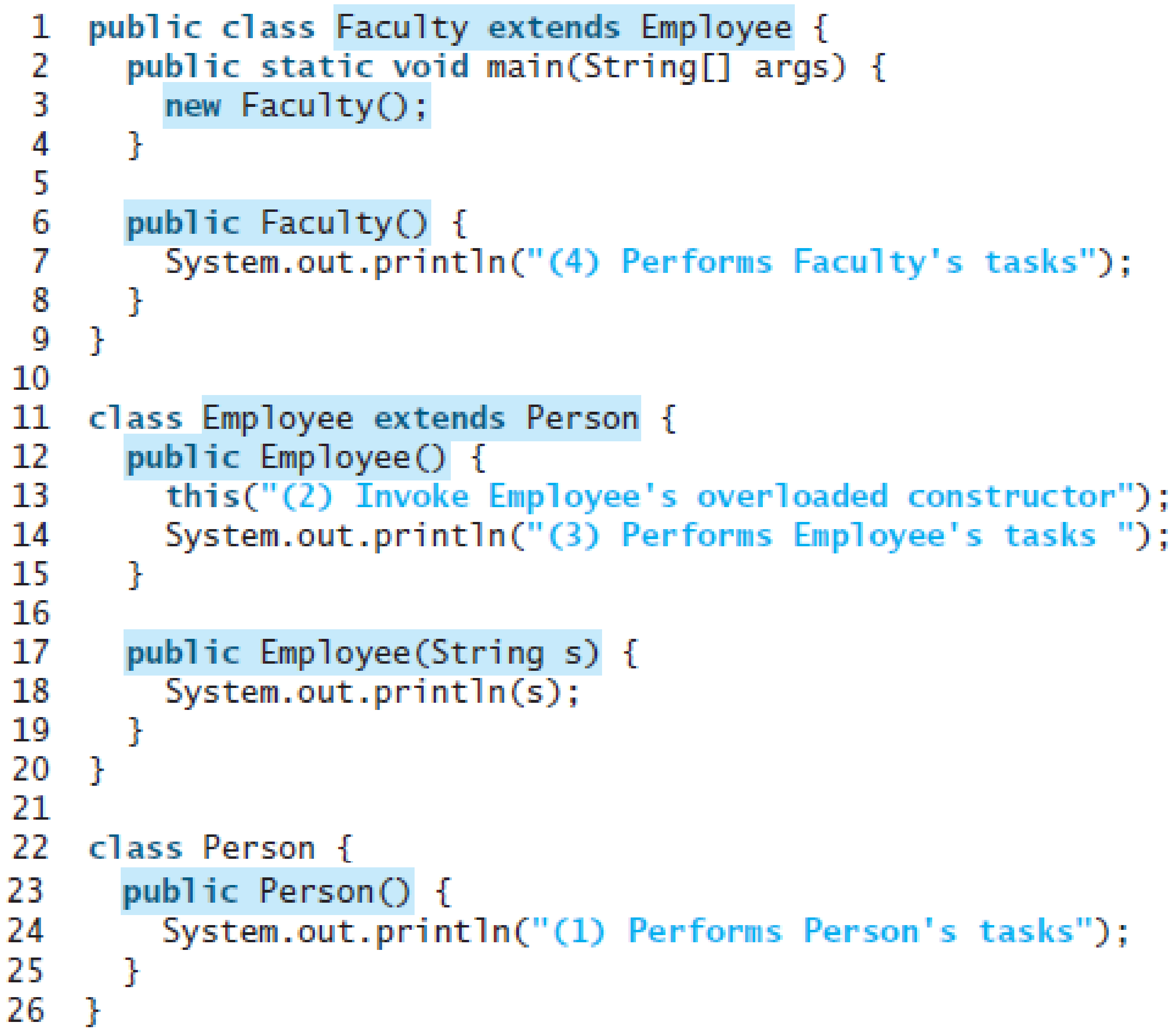
Constructor Chaining (Cont.)

- In any case, constructing an instance of a class invokes the constructors of all the superclasses along the inheritance hierarchy.

Constructor Chaining

- – When constructing an object of a subclass, the subclass constructor first invokes its superclass constructor before performing its own tasks.
- – If the superclass is derived from another class, the superclass constructor invokes its parent-class constructor before performing its tasks.
- – This process continues until the last constructor along the inheritance hierarchy is called.

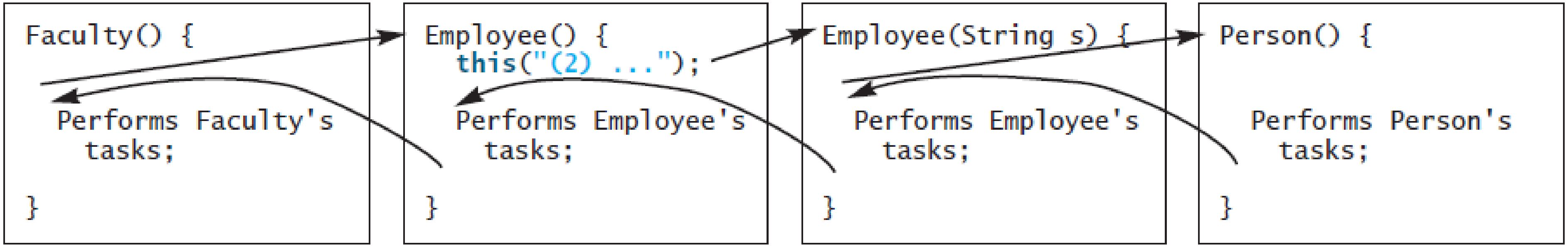
Constructor Chaining: Example



```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Performs Faculty's tasks");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Performs Employee's tasks ");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Performs Person's tasks");
  }
}
```

Constructor Chaining: Example (Cont.)

```
(1) Performs Person's tasks
(2) Invoke Employee's overloaded constructor
(3) Performs Employee's tasks
(4) Performs Faculty's tasks
```



```
Faculty() {

  Performs Faculty's
    tasks;

}
```

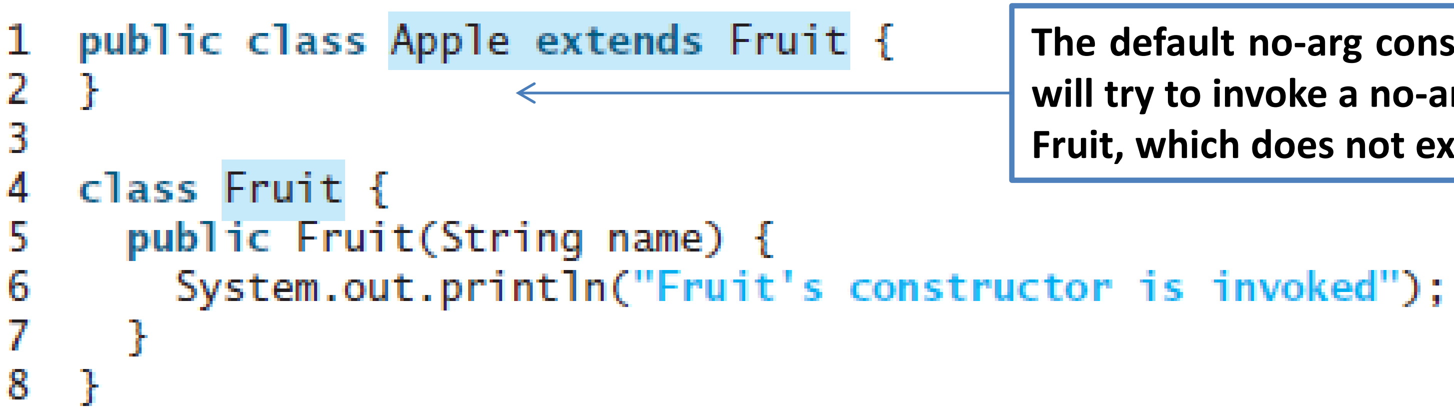
```
Employee() {
  this("(2) ...");

  Performs Employee's
    tasks;

}
```

```
Employee(String s) {

  Performs Employee's
    tasks;

}
```

```
Person() {

  Performs Person's
    tasks;

}
```

Caution!!

- If a class is designed to be extended, it is better to provide a no-arg constructor to avoid programming errors.
- Example: this code cannot be compiled:

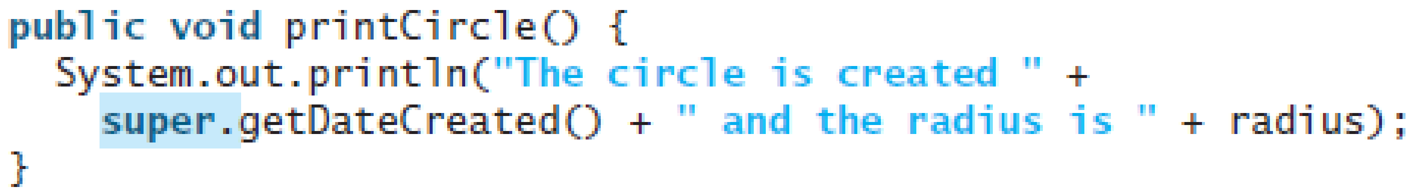


```
public class Apple extends Fruit {
}

class Fruit {
  public Fruit(String name) {
    System.out.println("Fruit's constructor is invoked");
  }
}
```

The default no-arg constructor of Apple will try to invoke a no-arg constructor of Fruit, which does not exist!

Using the *Super* Keyword to Call a Superclass Method

- The keyword *super* can be used to reference a method other than the constructor in the superclass. The syntax is:
  - ***super.method(parameters);***
- You could rewrite the *printCircle()* method in the Circle class as follows:

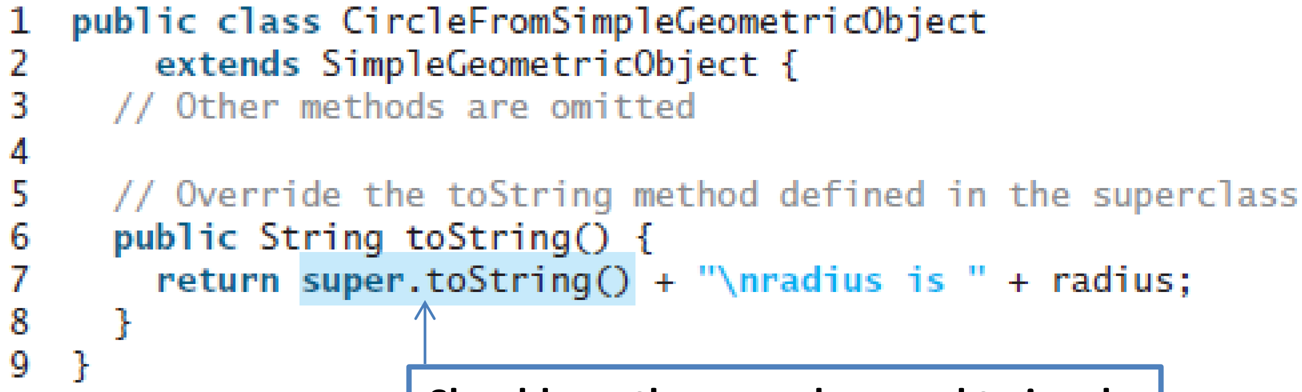


```
public void printCircle() {
  System.out.println("The circle is created " +
    super.getDateCreated() + " and the radius is " + radius);
}
```

- It is not necessary to put super before *getDateCreated()* in this case, however, because *getDateCreated* is a method in the *GeometricObject* class and is inherited by the Circle class.
  - Cases were the *super* keyword is needed to invoke the superclass methods will be showed when methods overriding is introduced.

Overriding Methods

- A subclass inherits methods from a superclass.
- Sometimes, it is necessary for the subclass to modify the implementation of a method defined in the superclass.
  - This is referred to as *method overriding*.
- The *toString* method in the *GeometricObject* class returns the string representation of a geometric object.
- This method can be overridden to return the string representation of a circle:



```java
public class CircleFromSimpleGeometricObject
    extends SimpleGeometricObject {
  // Other methods are omitted

  // Override the toString method defined in the superclass
  public String toString() {
    return super.toString() + "\nradius is " + radius;
  }
}
```

Should use the super keyword to invoke the *toSrting* **method of the superclass** *GeometricObject***.**

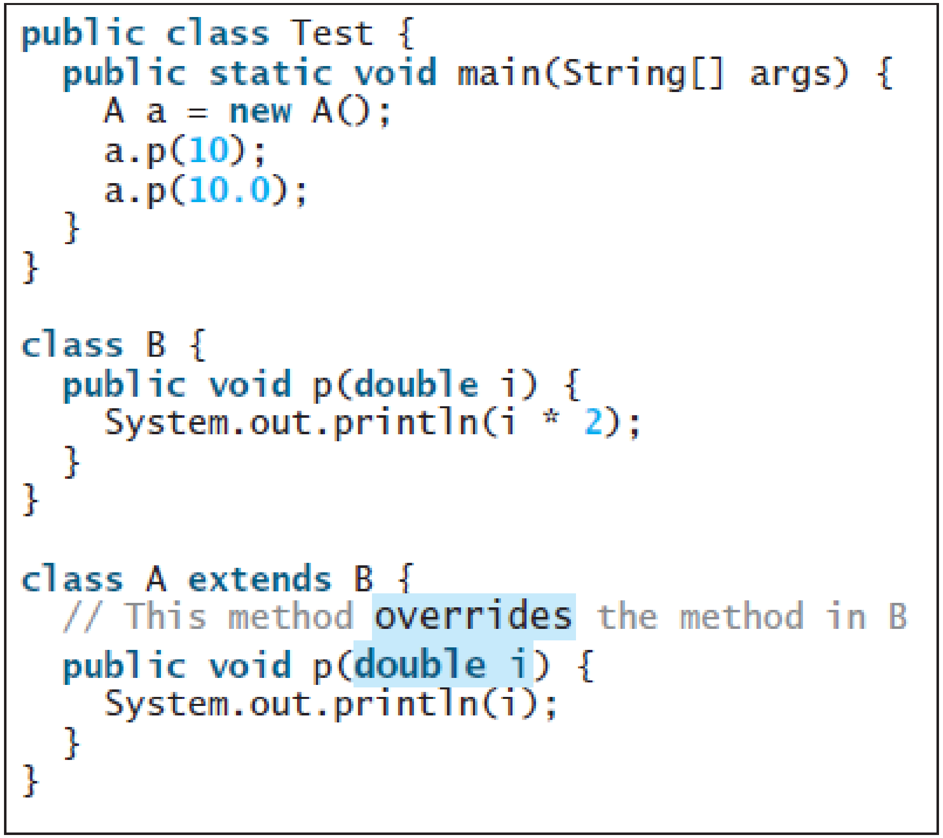
Overriding Methods (Cont.)

- An instance method can be overridden only if it is accessible.
  - Thus, a private method cannot be overridden, because it is not accessible outside its own class.
  - If a method defined in a subclass is private in its superclass, the two methods are completely unrelated.
- Like an instance method, a static method can be inherited. However a static method cannot be overridden.
  - If a static method defined in the superclass is redefined in a subclass, the method defined in the superclass is hidden.
  - The hidden static methods can be invoked using the syntax *SuperClassName.staticMethodName*.

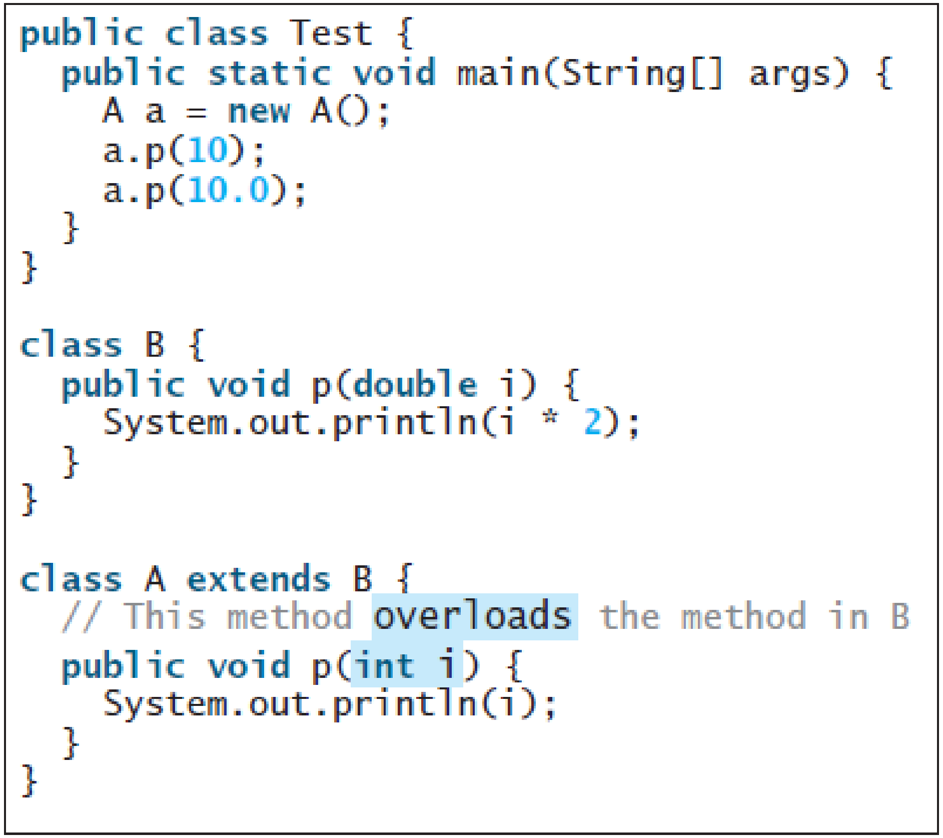
Overriding vs. Overloading

- Overloading means to define multiple methods with the same name but different signatures.
- Overriding means to provide a new implementation for a method in the subclass.
  - The method should be defined in the subclass using the same signature and the same return type.

Overriding vs. Overloading: Example



```java
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overrides the method in B
  public void p(double i) {
    System.out.println(i);
  }
}
```



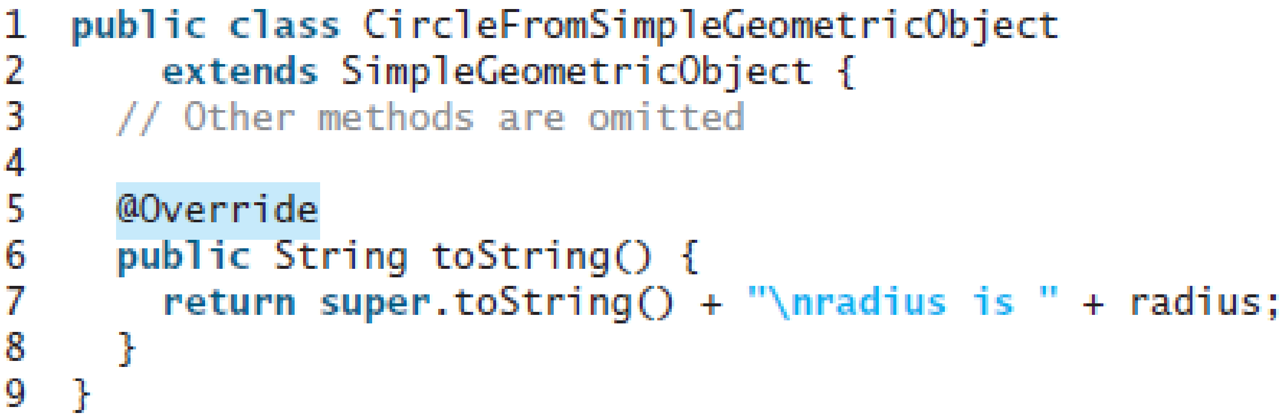
```java
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overloads the method in B
  public void p(int i) {
    System.out.println(i);
  }
}
```

Overriding vs. Overloading: Notes

- Overridden methods are in different classes related by inheritance; overloaded methods can be either in the same class or different classes related by inheritance.
- Overridden methods have the same signature and return type; overloaded methods have the same name but a different parameter list.

Override Annotation

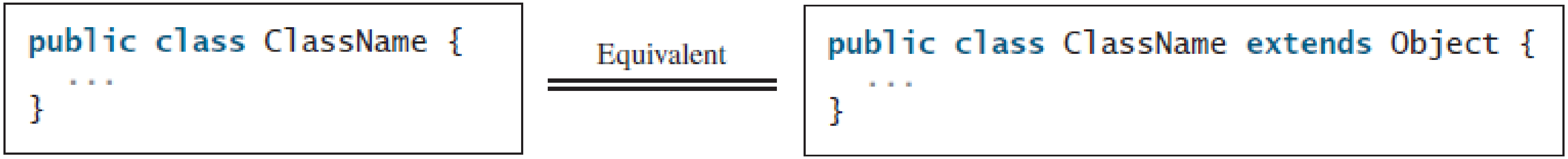
- To avoid mistakes, you can use a special Java syntax, called *override annotation*:
  - Place @Override before the method in the subclass.
- This annotation denotes that the annotated method is required to override a method in the superclass.
  - If a method with this annotation does not override its superclass's method, the compiler will report an error.
- For example, if *toString* is mistyped as *tostring*, a compile error is reported. If the override annotation isn't used, the compile won't report an error. Using annotation avoids mistakes.:



```
public class CircleFromSimpleGeometricObject
    extends SimpleGeometricObject {
  // Other methods are omitted

  @Override
  public String toString() {
    return super.toString() + "\nradius is " + radius;
  }
}
```

The Object Class and Its toString() Method

- *Every class in Java is descended from the java.lang.Object class.*
- If no inheritance is defined when a class is defined, the superclass of the class is *Object* by default.
- For example the following two class definitions are the same:



```
public class ClassName {
  ...
}
```

Equivalent

```
public class ClassName extends Object {
  ...
}
```

The Object Class and Its toString() Method (Cont.)

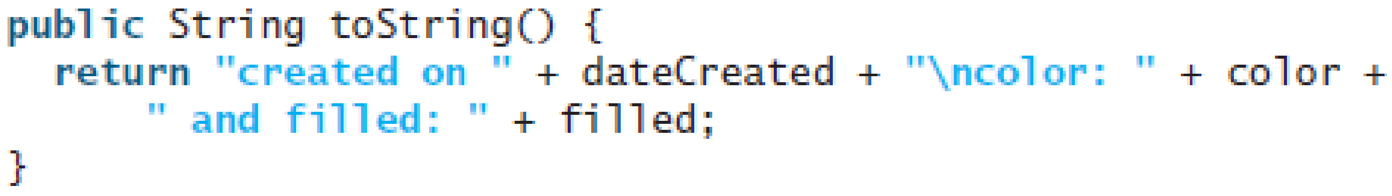
- One of the most important methods provided by the *Object* class is the method *toString*.
- The signature of the *toString* method is:
  - ***public String toString()***
- Invoking *toString()* on an object returns a string that describes the object.
  - By default, it returns a string consisting of a class name of which the object is an instance, an at sign (@), and the object's memory address in hexadecimal.

    Circle c = new Circle();

    System.out.println(c.toString());
  - For example, the output of the following code is something like: Circle@780324ff
  - This message is not very helpful or informative.
  - Usually you should override the *toString* method so that it returns a descriptive string representation of the object.

The Object Class and Its toString() Method (Cont.)

- Usually, we override the *toString* method so that it returns a descriptive string representation of the object.
- For example, the *toString* method in the *Object* class was overridden in the *GeometricObject* class as follows:



```
public String toString() {
  return "created on " + dateCreated + "\ncolor: " + color +
      " and filled: " + filled;
}
```

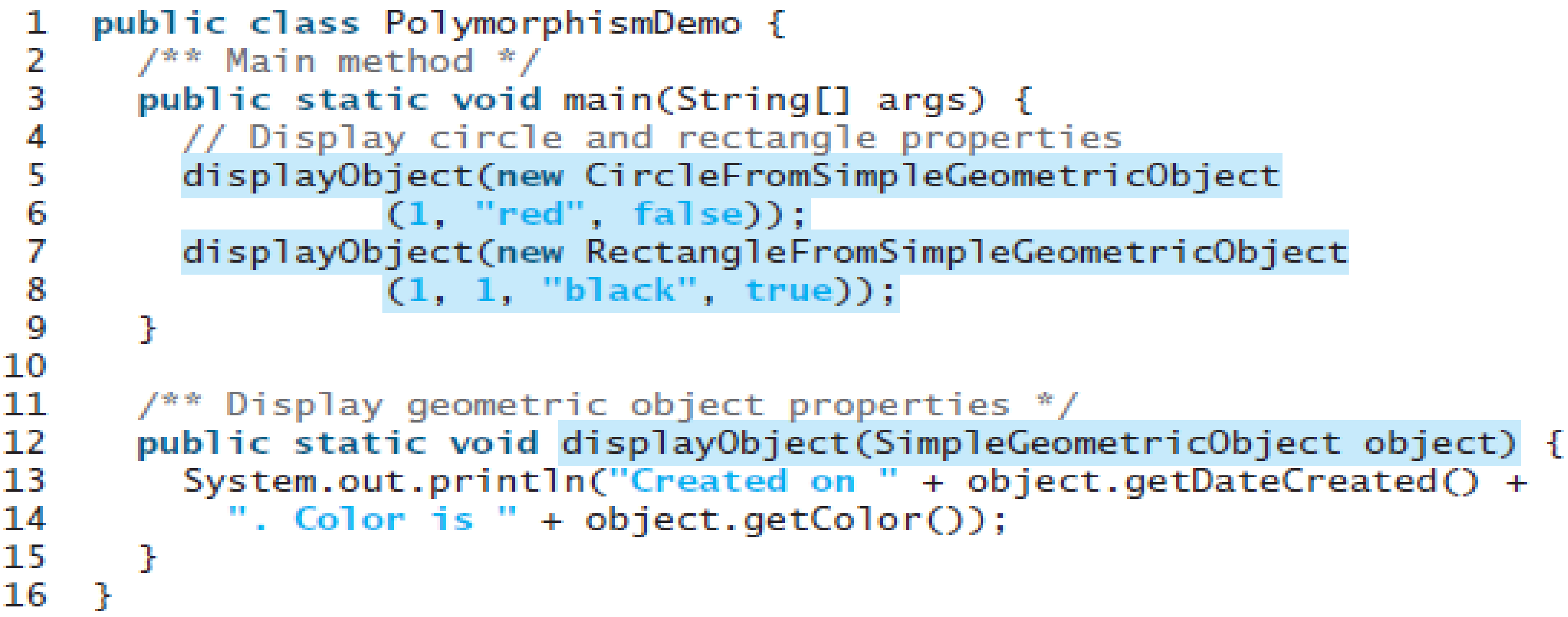
- You can also pass an object to invoke *System.out.println(object)* and *System.out.print(object)*.
  - This is equivalent to invoking *System.out.println(object.toString())* and *System.out.print(object.toString())*.

Polymorphism

- The three pillars of object-oriented programming are:
  - Encapsulation
  - Inheritance, and
  - Polymorphism.
- The inheritance relationship enables a subclass to inherit features from its superclass with additional new features.
- A class defines a type.
- A type defined by a subclass is called a *subtype*, and a type defined by its superclass is called a *supertype*.
  - Therefore, you can say that *Circle* is a subtype of *GeometricObject* and *GeometricObject* is a supertype for *Circle*.
- A subclass is a specialization of its superclass; every instance of a subclass is also an instance of its superclass, but not vice versa.
  - For example, every circle is a geometric object, but not every geometric object is a circle.

Polymorphism (Cont.)

- *Polymorphism* means that a variable of a supertype can refer to a subtype object.
  - You can always pass an instance of a subclass to a parameter of its superclass type.
  - An object of a subclass can be used wherever its superclass object is used.



```java
public class PolymorphismDemo {
  /** Main method */
  public static void main(String[] args) {
    // Display circle and rectangle properties
    displayObject(new CircleFromSimpleGeometricObject
             (1, "red", false));
    displayObject(new RectangleFromSimpleGeometricObject
             (1, 1, "black", true));
  }

  /** Display geometric object properties */
  public static void displayObject(SimpleGeometricObject object) {
    System.out.println("Created on " + object.getDateCreated() +
      ". Color is " + object.getColor());
  }
}
```

Dynamic Binding

- A method can be implemented in several classes along the inheritance chain.
  - The JVM decides which method is invoked at runtime.
- A method can be defined in a superclass and overridden in its subclass.
- For example, the *toString()* method is defined in the *Object* class and overridden in *GeometricObject*.

```
Object o = new GeometricObject();
System.out.println(o.toString());
```

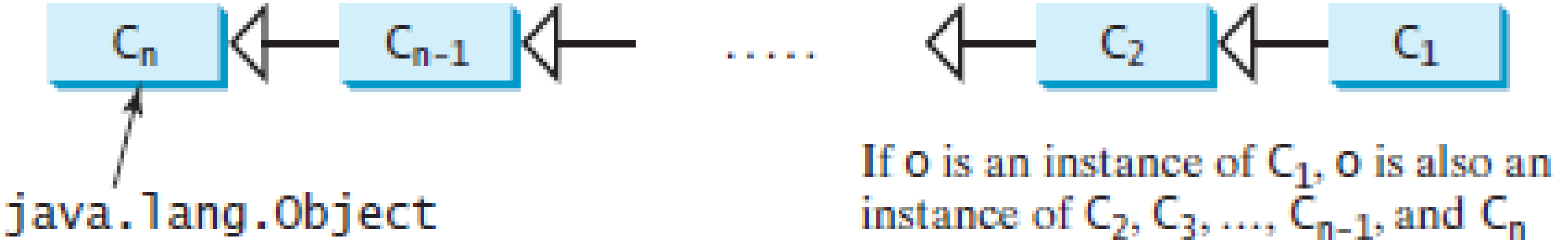
- Which *toString()* method is invoked by *o*?

Dynamic Binding (Cont.)

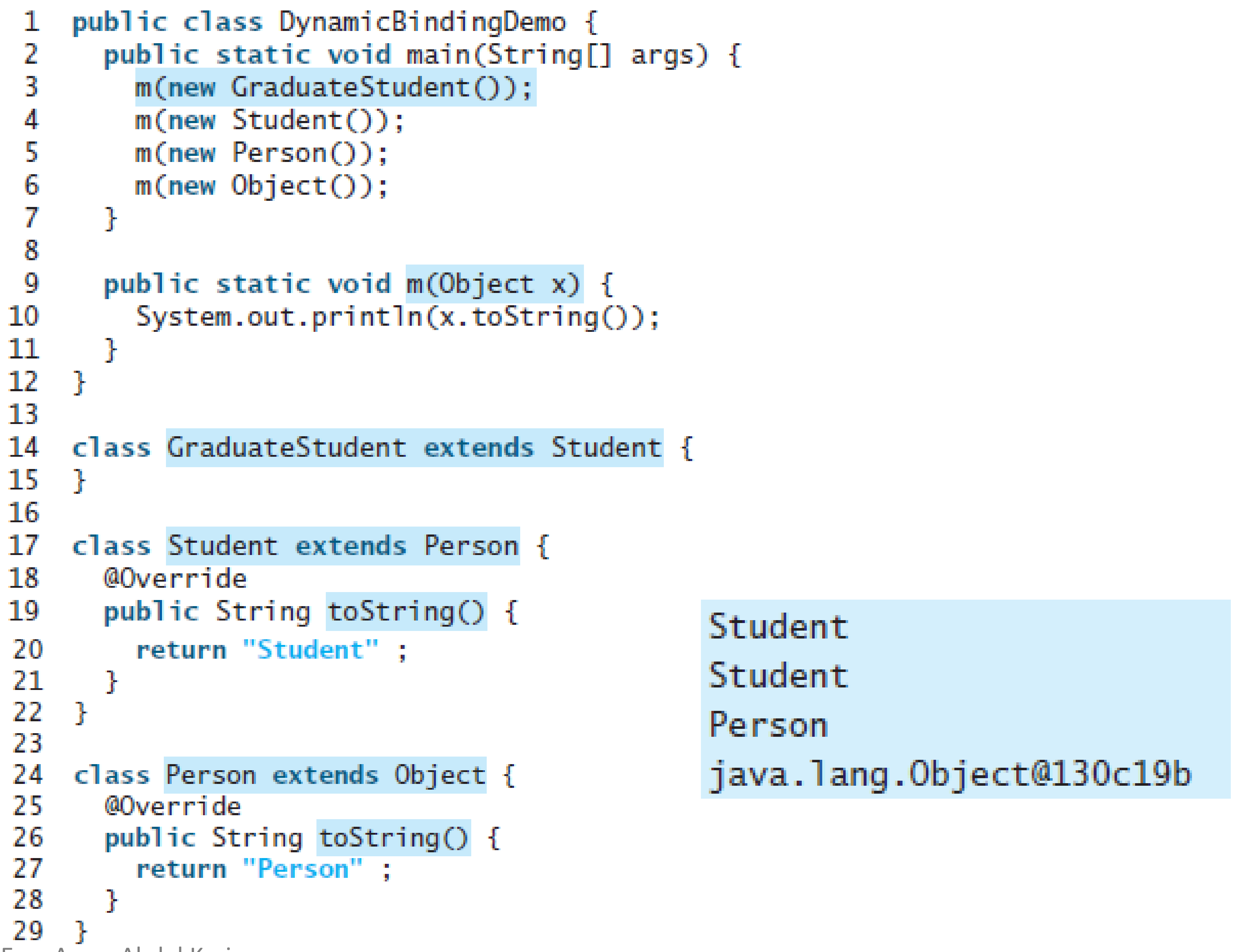
- The type that declares a variable is called the variable’s ***declared type***.
  - In the previous example, *o*’s declared type is *Object*.
  - A variable of a reference type can hold a *null* value or a reference to an instance of the declared type.
  - The instance may be created using the constructor of the declared type or its subtype.
- The ***actual type*** of the variable is the actual class for the object referenced by the variable.
  - Here *o*’s actual type is *GeometricObject*, because *o* references an object created using *new GeometricObject()*.
- Which *toString()* method is invoked by *o* is determined by *o*’s **actual type**. This is known as *dynamic binding*.

Dynamic Binding (Cont.)

- Suppose an object *o* is an instance of classes *C1*, *C2*, . . . , *Cn-1*, and *Cn*, where *C1* is a subclass of *C2*, *C2* is a subclass of *C3*, . . . , and *Cn-1* is a subclass of *Cn*, as shown in the figure.
- That is, *Cn* is the most general class, and *C1* is the most specific class.
- In Java, *Cn* is the *Object* class.
- If *o* invokes a method *p*, the JVM searches for the implementation of the method *p* in *C1*, *C2*, . . . , *Cn-1*, and ***Cn***, in this order, until it is found.
- Once an implementation is found, the search stops and the first-found implementation is invoked.



If o is an instance of C_1, o is also an instance of C_2, C_3, …, C_{n-1}, and C_n



```java
public class DynamicBindingDemo {
  public static void main(String[] args) {
    m(new GraduateStudent());
    m(new Student());
    m(new Person());
    m(new Object());
  }

  public static void m(Object x) {
    System.out.println(x.toString());
  }
}

class GraduateStudent extends Student {
}

class Student extends Person {
  @Override
  public String toString() {
    return "Student" ;
  }
}

class Person extends Object {
  @Override
  public String toString() {
    return "Person" ;
  }
}
```

```
Student
Student
Person
java.lang.Object@130c19b
```

Dynamic Binding (Cont.)

- Matching a method signature and binding a method implementation are two separate issues.
- The **declared type** of the reference variable decides which method to match at compile time.
- The compiler finds a matching method according to the parameter type, number of parameters, and order of the parameters at compile time.
- A method may be implemented in several classes along the inheritance chain. The JVM dynamically binds the implementation of the method at runtime, decided by the **actual type** of the variable.

Casting Objects

- One object reference can be typecast into another object reference.
  - This is called casting object.
- In the preceding section, the statement

  *m(**new** Student());*

  assigns the object *new Student()* to a parameter of the *Object* type.
- This statement is equivalent to

  *Object o = **new** Student(); // Implicit casting*

  m(o);
- The statement *Object o = new Student()*, known as *implicit casting*, is legal because an instance of *Student* is an instance of *Object*.
- Suppose you want to assign the object reference *o* to a variable of the *Student* type using the following statement:

  Student b = o;

  In this case a compile error would occur.

Casting Objects (Cont.)

- The reason is that a *Student* object is always an instance of *Object*, but an *Object* is not necessarily an instance of *Student*.
- Even though you can see that *o* is really a *Student* object, the compiler is not clever enough to know it.
- To tell the compiler that *o* is a *Student* object, use ***explicit casting***.
  - The syntax is similar to the one used for casting among primitive data types.
  - Enclose the target object type in parentheses and place it before the object to be cast, as follows:

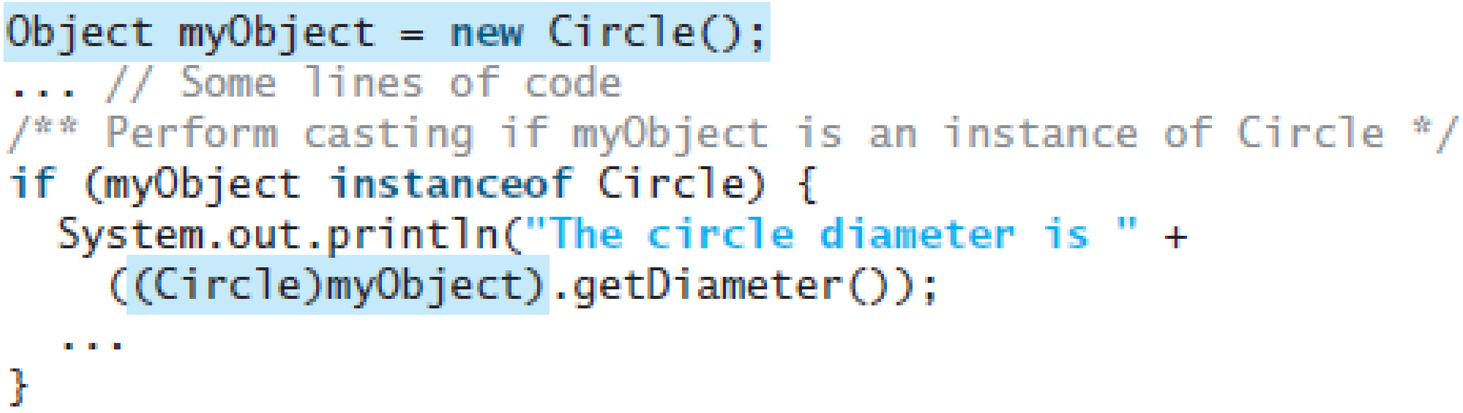
Student b = (Student)o; // Explicit casting

Casting Objects (Cont.)

- It is always possible to cast an instance of a subclass to a variable of a superclass (known as *upcasting*).
  - Because an instance of a subclass is *always* an instance of its superclass.
- When casting an instance of a superclass to a variable of its subclass (known as *downcasting*), explicit casting must be used.
  - To confirm your intention to the compiler with the *(SubclassName)* cast notation.

Casting Objects (Cont.)

- For the casting to be successful, you must make sure that the object to be cast is an instance of the subclass.
- If the superclass object is not an instance of the subclass, a runtime *ClassCastException* occurs.
  - For example, if an object is not an instance of *Student*, it cannot be cast into a variable of *Student*.
- It is a good practice, therefore, to ensure that the object is an instance of another object before attempting a casting.
  - This can be accomplished by using the *instanceof* operator.

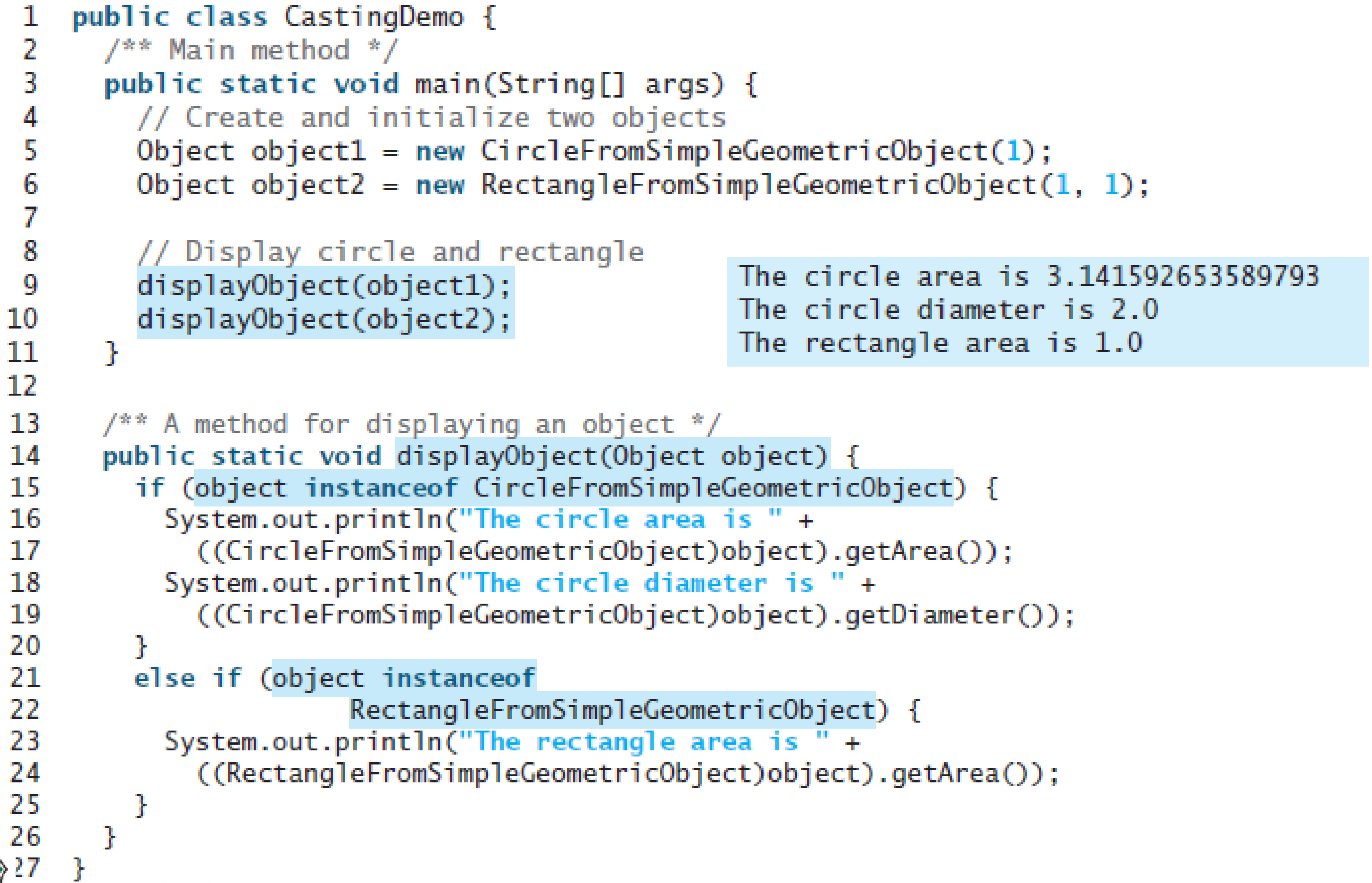


```
Object myObject = new Circle();
... // Some lines of code
/** Perform casting if myObject is an instance of Circle */
if (myObject instanceof Circle) {
  System.out.println("The circle diameter is " +
    ((Circle)myObject).getDiameter());
  ...
}
```

Why Casting is Necessary?

- You may be wondering why casting is necessary. The variable *myObject* is declared *Object*.
- The ***declared type*** decides which method to match at compile time.
  - Using *myObject.getDiameter()* would cause a compile error, because the *Object* class does not have the *getDiameter* method.
  - The compiler cannot find a match for *myObject.getDiameter()*.
- Therefore, it is necessary to cast *myObject* into the *Circle* type to tell the compiler that *myObject* is also an instance of *Circle*.
- Why not define *myObject* as a *Circle* type in the first place?
  - To enable **generic programming**, it is a good practice to define a variable with a supertype, which can accept an object of any subtype.

Casting and Polymorphism



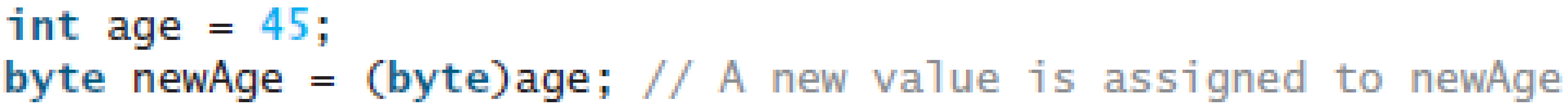
```java
public class CastingDemo {
  /** Main method */
  public static void main(String[] args) {
    // Create and initialize two objects
    Object object1 = new CircleFromSimpleGeometricObject(1);
    Object object2 = new RectangleFromSimpleGeometricObject(1, 1);

    // Display circle and rectangle
    displayObject(object1);
    displayObject(object2);
  }

  /** A method for displaying an object */
  public static void displayObject(Object object) {
    if (object instanceof CircleFromSimpleGeometricObject) {
      System.out.println("The circle area is " +
        ((CircleFromSimpleGeometricObject)object).getArea());
      System.out.println("The circle diameter is " +
        ((CircleFromSimpleGeometricObject)object).getDiameter());
    }
    else if (object instanceof
                  RectangleFromSimpleGeometricObject) {
      System.out.println("The rectangle area is " +
        ((RectangleFromSimpleGeometricObject)object).getArea());
    }
  }
}
```

```
The circle area is 3.141592653589793
The circle diameter is 2.0
The rectangle area is 1.0
```

Comments

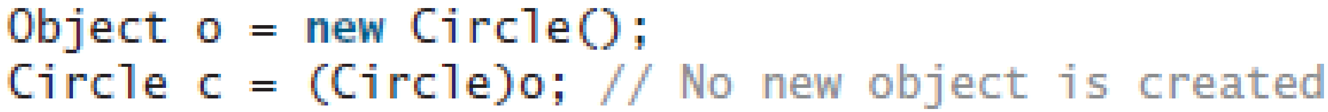
- The object member access operator (.) precedes the casting operator.
  - Use parentheses to ensure that casting is done before the . operator, as in

  ((Circle)object).getArea();
- Casting a primitive type value is different from casting an object reference.
  - Casting a primitive type value returns a new value. For example:



```
int age = 45;
byte newAge = (byte)age; // A new value is assigned to newAge
```

  - However, casting an object reference does not create a new object. For example:



```
Object o = new Circle();
Circle c = (Circle)o; // No new object is created
```

The Object's *equals* Method

- Another method defined in the *Object* class that is often used is the *equals* method. Its signature is

  public boolean equals(Object o)

- This method tests whether two objects are equal. The syntax for invoking it is:

  object1.equals(object2);

- The default implementation of the *equals* method in the *Object* class is:

```
public boolean equals(Object obj) {
        return (this == obj);
}
```

- This implementation checks whether two reference variables point to the same object using the **==** operator.
- You should override this method in your custom class to test whether two distinct objects have the same content.

The Object’s *equals* Method (Cont.)

- The *equals* method is overridden in many classes in the Java API, such as *java.lang.String* and *java.util.Date*, to compare whether the contents of two objects are equal.
- You can override the *equals* method in the *Circle* class to compare whether two circles are equal based on their radius as follows:

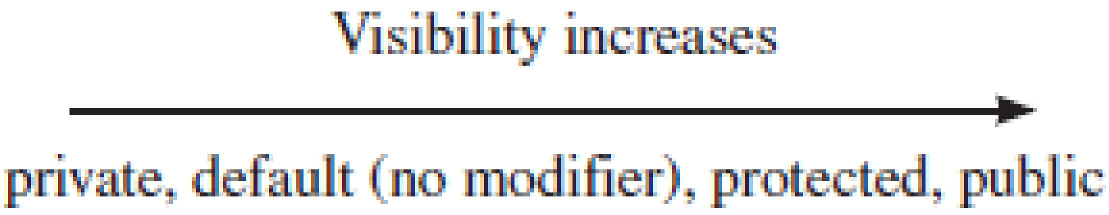
```
public boolean equals(Object o) {
    if (o instanceof Circle)
        return radius == ((Circle)o).radius;
    else
        return this == o;
}
```

- Using the signature *equals(SomeClassName obj)* (e.g., *equals(Circle c))* to override the *equals* method in a subclass is a common mistake. You should use *equals(Object ob*j).

The Protected Data and Methods

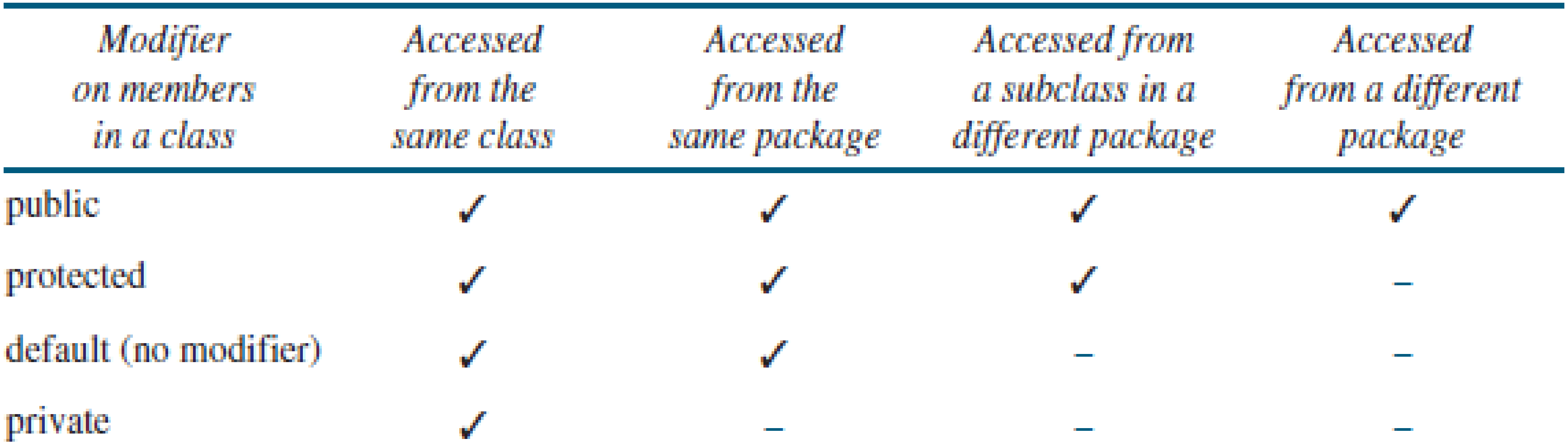
- So far you have used the *private* and *public* keywords to specify whether data fields and methods can be accessed from outside of the class.
- *Private* members can be accessed only from inside of the class, and *public* members can be accessed from any other classes.
- Often it is desirable to allow subclasses to access data fields or methods defined in the superclass, but not to allow non-subclasses to access these data fields and methods.
- To accomplish this, you can use the *protected* keyword.
  - This way you can access protected data fields or methods in a superclass from its subclasses.

The Protected Data and Methods (Cont.)



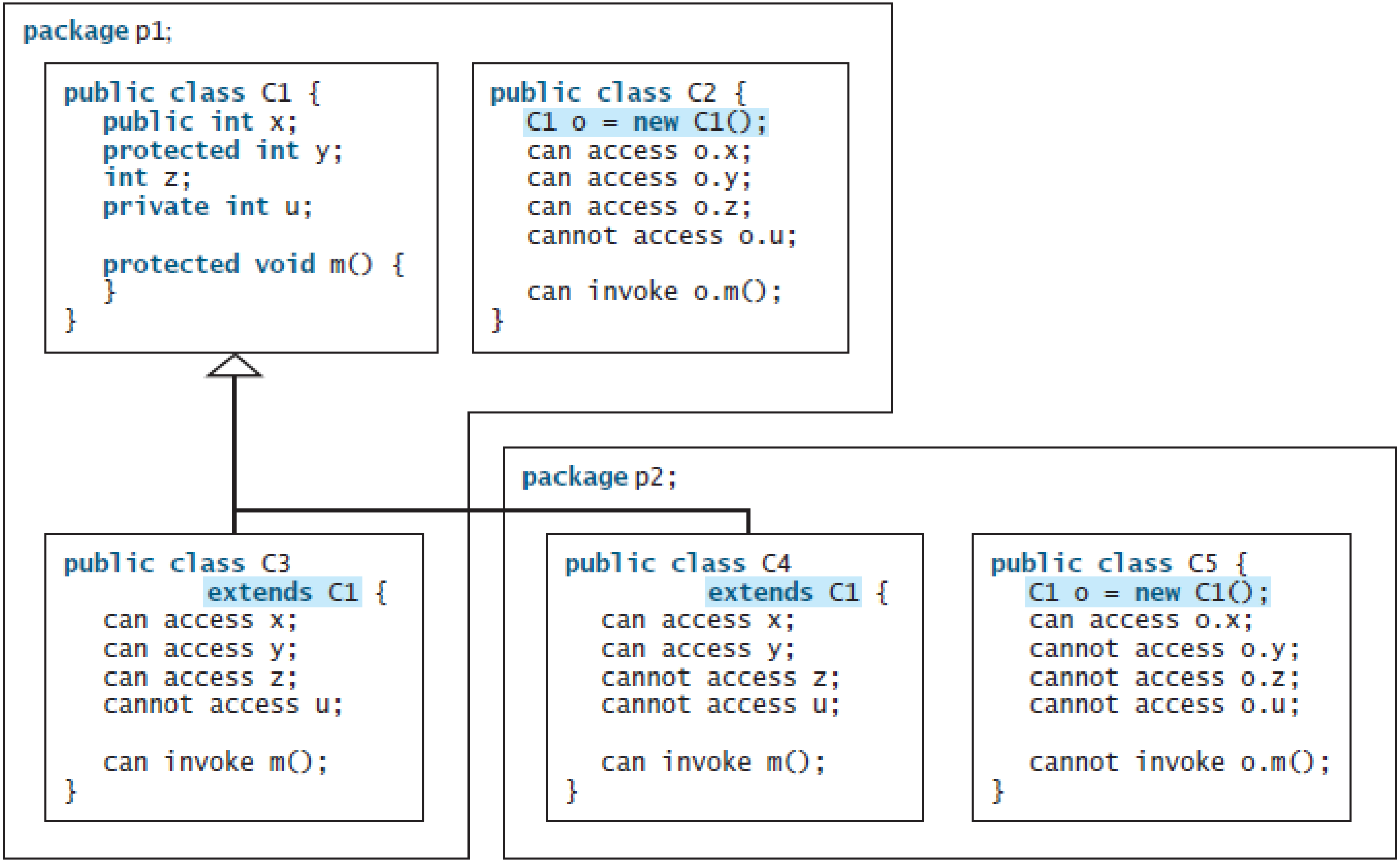
Visibility increases

private, default (no modifier), protected, public



Modifier on members in a class	*Accessed from the same class*	*Accessed from the same package*	*Accessed from a subclass in a different package*	*Accessed from a different package*
public	✓	✓	✓	✓
protected	✓	✓	✓	–
default (no modifier)	✓	✓	–	–
private	✓	–	–	–

The Protected Data and Methods (Cont.)



package p1;

```
public class C1 {
   public int x;
   protected int y;
   int z;
   private int u;

   protected void m() {
   }
}
```

```
public class C2 {
   C1 o = new C1();
   can access o.x;
   can access o.y;
   can access o.z;
   cannot access o.u;

   can invoke o.m();
}
```

```
public class C3
         extends C1 {
   can access x;
   can access y;
   can access z;
   cannot access u;

   can invoke m();
}
```

package p2;

```
public class C4
         extends C1 {
   can access x;
   can access y;
   cannot access z;
   cannot access u;

   can invoke m();
}
```

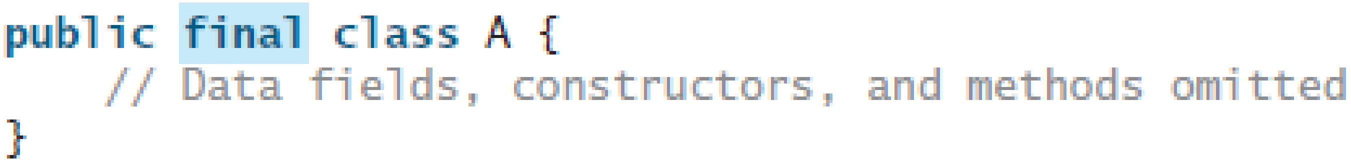
```
public class C5 {
   C1 o = new C1();
   can access o.x;
   cannot access o.y;
   cannot access o.z;
   cannot access o.u;

   cannot invoke o.m();
}
```

Visibility Modifiers (Comments)

- Your class can be used in two ways:
  (1) for creating instances of the class and
  (2) for defining subclasses by extending the class.
- Make the members *private* if they are not intended for use from outside the class.
- Make the members *public* if they are intended for the users of the class.
- Make the fields or methods *protected* if they are intended for the extenders of the class but not for the users of the class.
- A subclass cannot weaken the accessibility of a method defined in the superclass when overriding it.
  - For example, if a method is defined as public in the superclass, it must be defined as public in the subclass.

Preventing Extending and Overriding

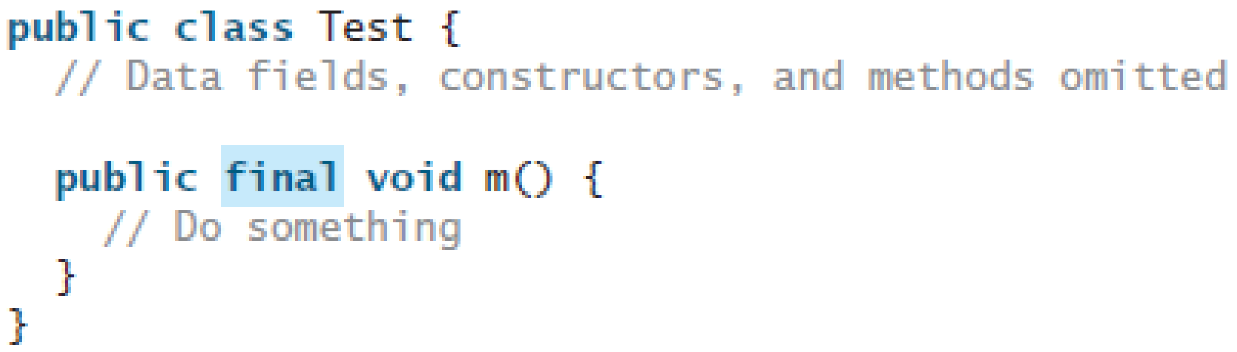
- You may occasionally want to prevent classes from being extended.
- In such cases, use the *final* modifier to indicate that a class is final and cannot be a parent class.
- The *Math* class is a final class. The *String*, StringBuilder, and *StringBuffer* classes are also final classes.



```
public final class A {
    // Data fields, constructors, and methods omitted
}
```

Preventing Extending and Overriding (Cont.)

- You also can define a method to be *final*; a final method cannot be overridden by its subclasses.
- For example, the following method *m* is final and cannot be overridden:



```
public class Test {
  // Data fields, constructors, and methods omitted

  public final void m() {
    // Do something
  }
}
```

Object-Oriented Problem Solving

Strings

Based on Chapters 4 & 10 of "Introduction to Java Programming" by Y. Daniel Liang.

Eng. Asma Abdel Karim
Computer Engineering Department

Outline

- The String Class (4.4)(10.10)
- The StringBuilder and StringBuffer Classes (10.11)

The String Class

- A *string* is a sequence of characters.
- *String* is a predefined class in the Java library, just like the classes *System* and *Scanner*.
- The *String* class has 13 constructors and more than 40 methods for manipulating strings.
- A string literal is a sequence of characters enclosed inside double quotes.
  - Example: *“Welcome to Java!”*

Constructing a String

- You can create a string object from a string literal or from an array of characters.
- The *String* type is not a primitive type. It is a reference type.
- In the example below, *message* is a reference variable that references a string object with contents *Welcome to Java*.
- Examples:
  - *String message = new String (“Welcome to Java!”);*
  - *String message = “Welcome to Java!”;*
  - *char[] charArray = {'G', 'o', 'o', 'd', ' ', 'D', 'a', 'y'};*
  - *String message = new String(charArray);*

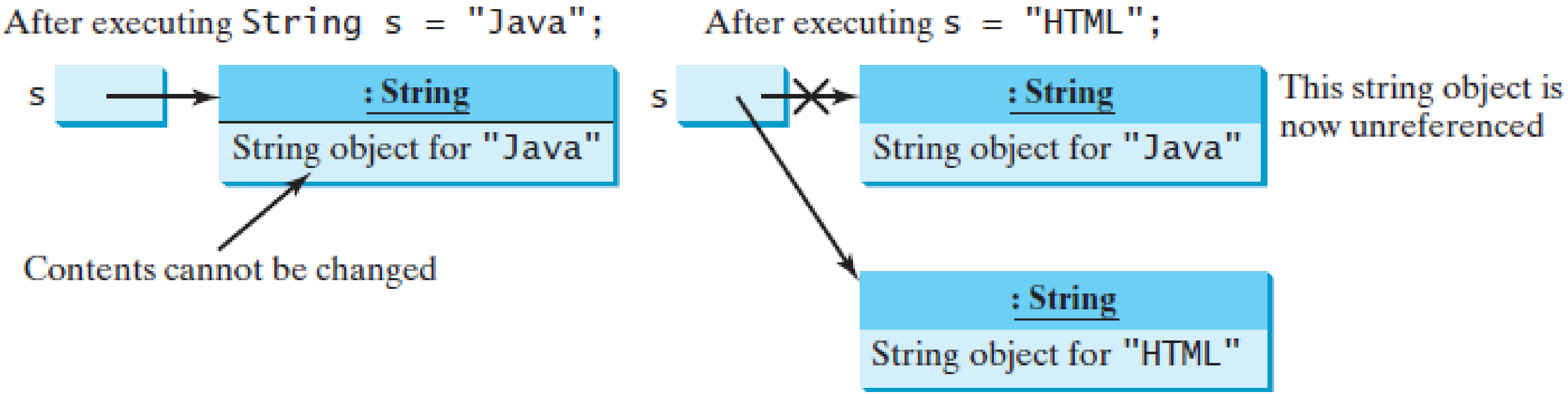
Java treats a string literal as a String object.

Immutable Strings

- A String object is immutable: its contents cannot be changed.
- Example:

  String s = “Java”;

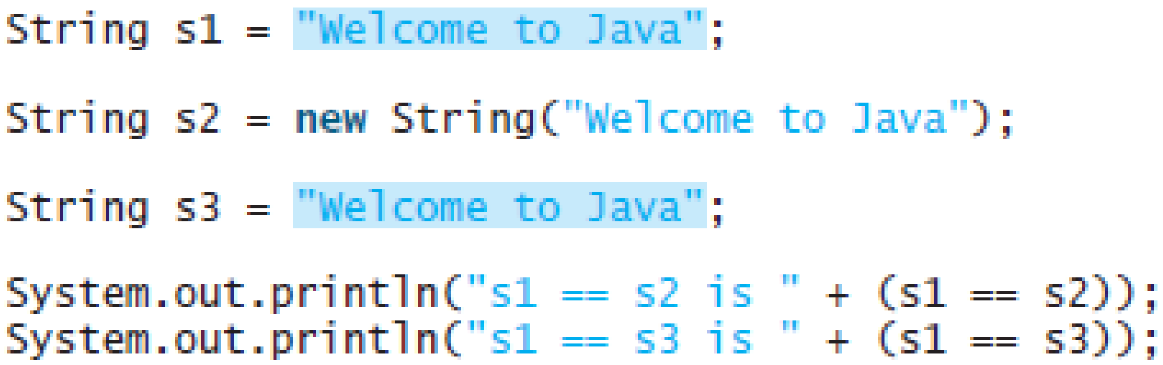
  s = “HTML”;


After executing String s = "Java";
s
: String
String object for "Java"
Contents cannot be changed
After executing s = "HTML";
s
: String
String object for "Java"
This string object is now unreferenced
: String
String object for "HTML"

Interned Strings

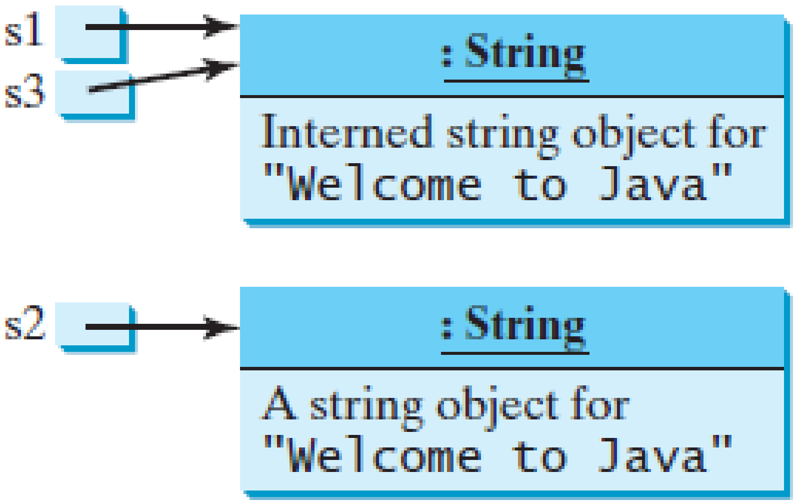
- The JVM uses a unique instance for string literals with the same character sequence.
  - In order to achieve efficiency and save memory.
  - Such an instance is called an *interned string*.

Interned Strings: Example

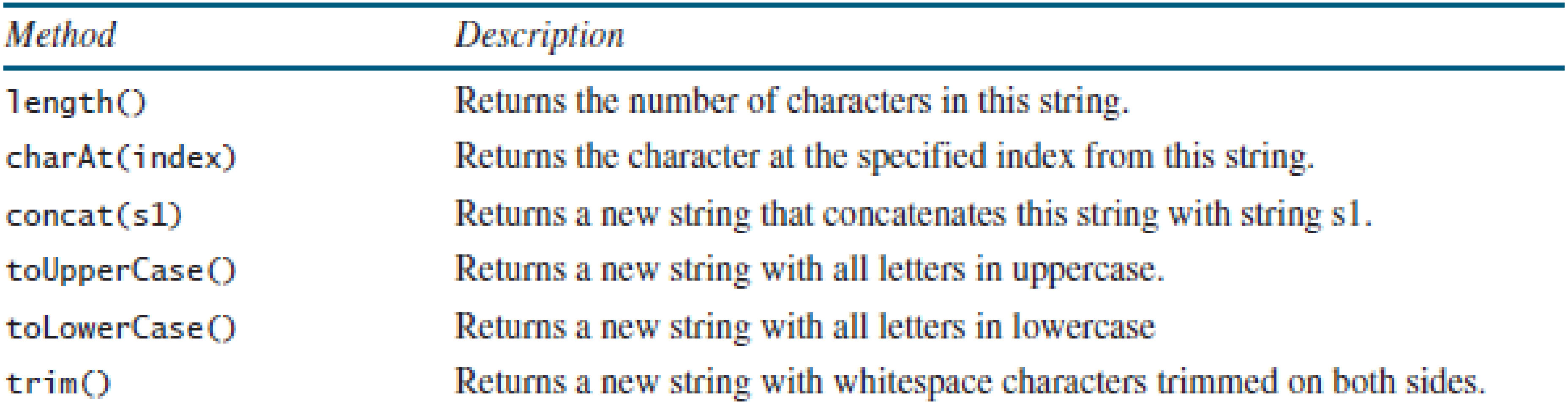


```
String s1 = "Welcome to Java";

String s2 = new String("Welcome to Java");

String s3 = "Welcome to Java";

System.out.println("s1 == s2 is " + (s1 == s2));
System.out.println("s1 == s3 is " + (s1 == s3));
```

Output:
s1==s2 is false
s1==s3 is true


s1
s3
: String
Interned string object for "Welcome to Java"
s2
: String
A string object for "Welcome to Java"

Simple Methods for String Class



Method	*Description*
`length()`	Returns the number of characters in this string.
`charAt(index)`	Returns the character at the specified index from this string.
`concat(s1)`	Returns a new string that concatenates this string with string s1.
`toUpperCase()`	Returns a new string with all letters in uppercase.
`toLowerCase()`	Returns a new string with all letters in lowercase
`trim()`	Returns a new string with whitespace characters trimmed on both sides.

Getting String Length

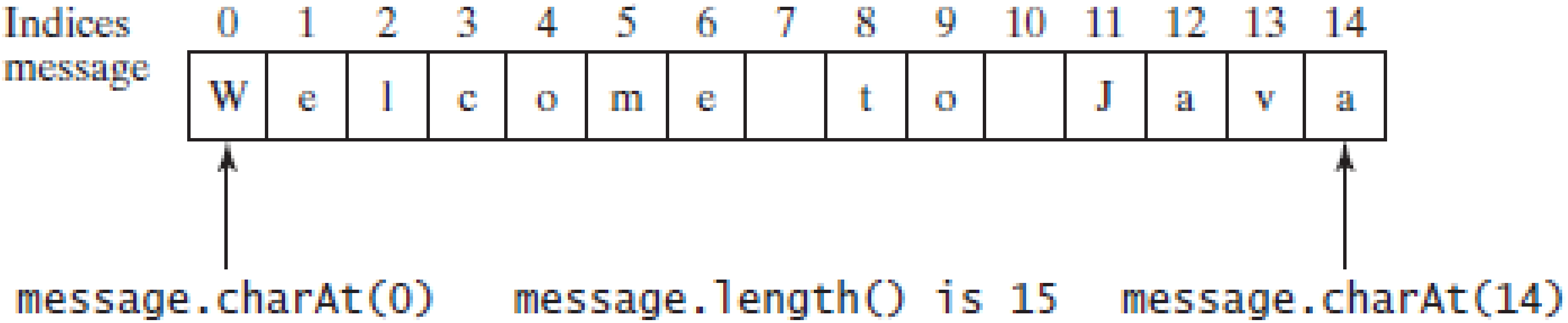
- You can use the *length()* method to return the number of characters in a string. For example, the following code

```
String message = "Welcome to Java";
System.out.println("The length of " + message + " is "+
message.length());
```

- For convenience, Java allows you to use the string literal to refer directly to strings without creating new variables.
  - Thus, *"Welcome to Java".length()* is correct and returns *15*.
  - Note that **""** denotes an *empty string* and *"".length()* is *0*.

Getting Characters from a String

- The *s.charAt(index)* method can be used to retrieve a specific character in a string *s*, where the index is between *0* and *s.length()–1*.
- Attempting to access characters in a string **s** out of bounds is a common programming error. To avoid it, make sure that you do not use an index beyond *s.length() – 1*.
- For example, *s.charAt(s.length())* would cause a *StringIndexOutOfBoundsException*.



Indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
message	W	e	l	c	o	m	e		t	o		J	a	v	a

Concatenating Strings

- You can use the *concat* method to concatenate two strings.
- The statement shown below, for example, concatenates strings *s1* and *s2* into *s3*:

  String s3 = s1.concat(s2);

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

  String s3 = s1 + s2;

- The following code combines the strings *message*, *" and "*, and *"HTML"* into one string:

  String myString = message + ***" and "*** *+* ***"HTML"****;*

Converting Strings

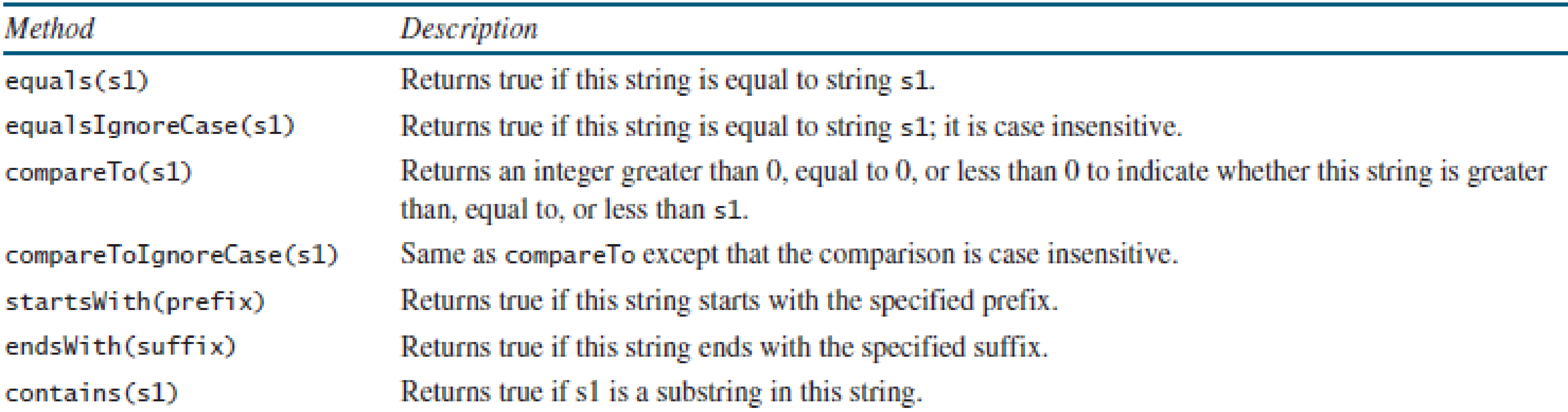
- The *toLowerCase()* method returns a new string with all lowercase letters and the *toUpperCase()* method returns a new string with all uppercase letters.
  - For example:

  "Welcome".toLowerCase() returns a new string *welcome*.
  "Welcome".toUpperCase() returns a new string *WELCOME*.

- The *trim()* method returns a new string by eliminating whitespace characters from both ends of the string.
  - The characters *' ', \t, \f, \r, or \n* are known as *whitespace characters*.
  - For example,

  "\t Good Night \n".trim() returns a new string *Good Night*.

Comparing Strings



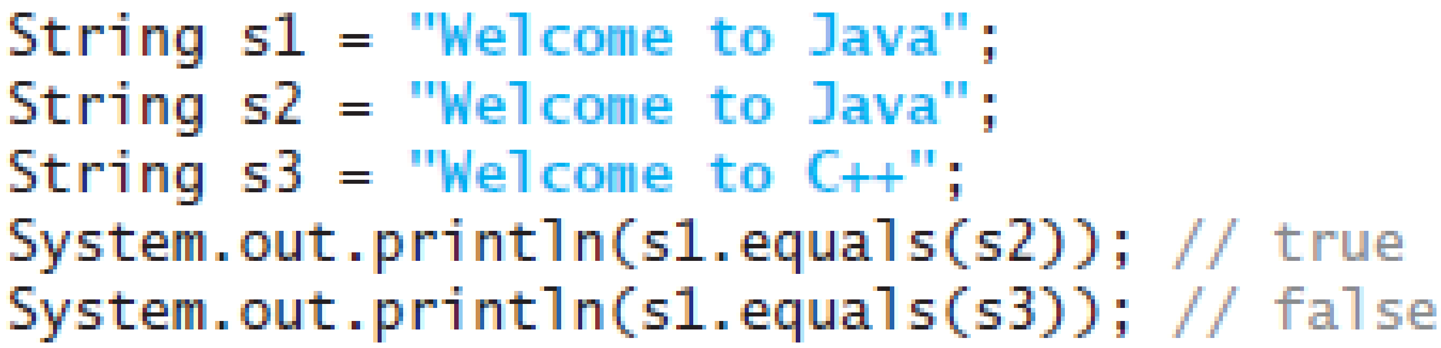
Method	*Description*
`equals(s1)`	Returns true if this string is equal to string `s1`.
`equalsIgnoreCase(s1)`	Returns true if this string is equal to string `s1`; it is case insensitive.
`compareTo(s1)`	Returns an integer greater than 0, equal to 0, or less than 0 to indicate whether this string is greater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

Comparing Strings (Cont.)

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

```
 if (string1.equals(string2))
   System.out.println("string1 and string2 have the same contents");
else
  System.out.println("string1 and string2 are not equal");
```

Comparing Strings (Cont.)



```
String s1 = "Welcome to Java";
String s2 = "Welcome to Java";
String s3 = "Welcome to C++";
System.out.println(s1.equals(s2)); // true
System.out.println(s1.equals(s3)); // false
```

Comparing Strings (Cont.)

- The *compareTo* method can also be used to compare two strings.
- For example, consider the following code:

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

Comparing Strings (Cont.)

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

"Welcome to Java".startsWith("We") *returns* ***true****.*

"Welcome to Java".startsWith("we") *returns* ***false****.*

"Welcome to Java".endsWith("va") *returns* ***true****.*

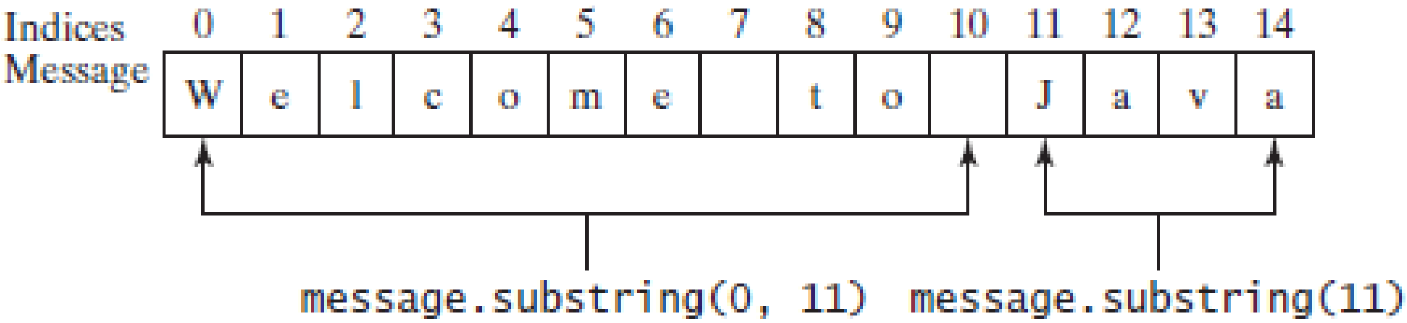
"Welcome to Java".endsWith("v") *returns* ***false****.*

"Welcome to Java".contains("to") *returns* ***true****.*

"Welcome to Java".contains("To") *returns* ***false****.*

Obtaining Substrings

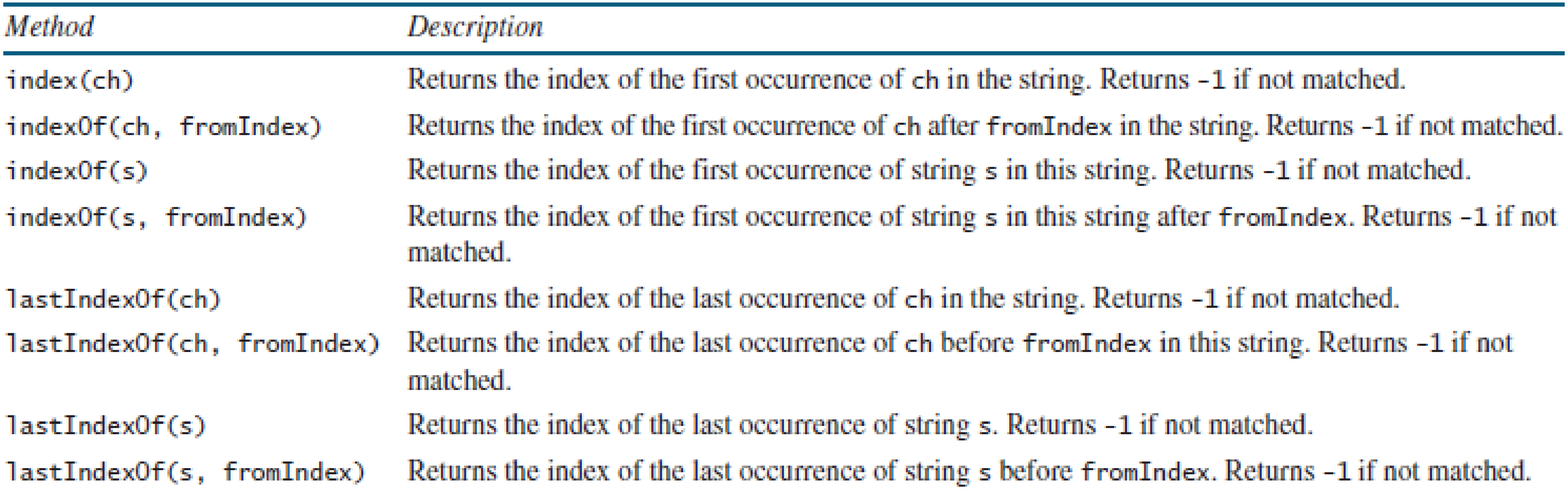
Method	*Description*
`substring(beginIndex)`	Returns this string's substring that begins with the character at the specified `beginIndex` and extends to the end of the string, as shown in Figure 4.2.
`substring(beginIndex, endIndex)`	Returns this string's substring that begins at the specified `beginIndex` and extends to the character at index `endIndex – 1`, as shown in Figure 4.2. Note that the character at `endIndex` is not part of the substring.



Indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Message	W	e	l	c	o	m	e		t	o		J	a	v	a

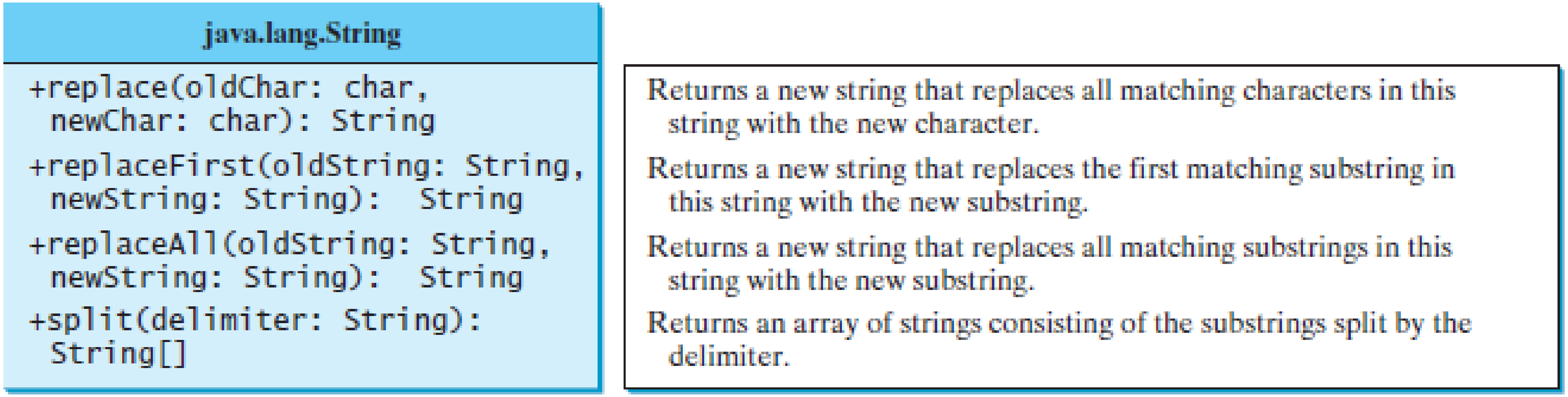
`message.substring(0, 11)` `message.substring(11)`

Finding a Character or a Substring in a String



Method	*Description*
`index(ch)`	Returns the index of the first occurrence of `ch` in the string. Returns `-1` if not matched.
`indexOf(ch, fromIndex)`	Returns the index of the first occurrence of `ch` after `fromIndex` in the string. Returns `-1` if not matched.
`indexOf(s)`	Returns the index of the first occurrence of string `s` in this string. Returns `-1` if not matched.
`indexOf(s, fromIndex)`	Returns the index of the first occurrence of string `s` in this string after `fromIndex`. Returns `-1` if not matched.
`lastIndexOf(ch)`	Returns the index of the last occurrence of `ch` in the string. Returns `-1` if not matched.
`lastIndexOf(ch, fromIndex)`	Returns the index of the last occurrence of `ch` before `fromIndex` in this string. Returns `-1` if not matched.
`lastIndexOf(s)`	Returns the index of the last occurrence of string `s`. Returns `-1` if not matched.
`lastIndexOf(s, fromIndex)`	Returns the index of the last occurrence of string `s` before `fromIndex`. Returns `-1` if not matched.

Replacing and Splitting Strings



java.lang.String	
+replace(oldChar: char, newChar: char): String	Returns a new string that replaces all matching characters in this string with the new character.
+replaceFirst(oldString: String, newString: String): String	Returns a new string that replaces the first matching substring in this string with the new substring.
+replaceAll(oldString: String, newString: String): String	Returns a new string that replaces all matching substrings in this string with the new substring.
+split(delimiter: String): String[]	Returns an array of strings consisting of the substrings split by the delimiter.

`"Welcome".replace('e', 'A')` returns a new string, `WAlcomA`.
`"Welcome".replaceFirst("e", "AB")` returns a new string, `WABlcome`.
`"Welcome".replace("e", "AB")` returns a new string, `WABlcomAB`.
`"Welcome".replace("el", "AB")` returns a new string, `WABcome`.

```
String[] tokens = "Java#HTML#Perl".split("#");
for (int i = 0; i < tokens.length; i++)
  System.out.print(tokens[i] + " ");
```

Matching, Replacing and Splitting by Patterns

- A *regular expression* (abbreviated *regex*) is a string that describes a pattern for matching a set of strings.
- You can match, replace, or split a string by specifying a pattern.
- At first glance, the *matches* method is very similar to the *equals* method. For example, the following two statements both evaluate to *true*:

```
"Java".matches("Java");
"Java".equals("Java");
```

- However, the *matches* method is more powerful. It can match not only a fixed string, but also a set of strings that follow a pattern.

Matching, Replacing and Splitting by Patterns (Cont.)

- For example, the following statements all evaluate to **true**:

```
"Java is fun".matches("Java.*")
"Java is cool".matches("Java.*")
"Java is powerful".matches("Java.*")
```

- **Java.*** in the preceding statements is a regular expression. It describes a string pattern that begins with Java followed by *any* zero or more characters. Here, the substring matches any zero or more characters.
- The following statement evaluates to **true**.

```
"440-02-4534".matches("\\d{3}-\\d{2}-\\d{4}")
```

Here **\\d** represents a single digit, and **\\d{3}** represents three digits.

Matching, Replacing and Splitting by Patterns (Cont.)

- The *replaceAll*, *replaceFirst*, and *split* methods can be used with a regular expression.
- For example, the following statement returns a new string that replaces **$**, **+**, or **#** in **a+b$#c** with the string **NNN**.

```
String s = "a+b$#c".replaceAll("[$+#]", "NNN");
System.out.println(s);
```

  - Here the regular expression **[$+#]** specifies a pattern that matches **$**, **+**, or **#**. So, the output is **aNNNbNNNNNNc**.

Matching, Replacing and Splitting by Patterns (Cont.)

- The following statement splits the string into an array of strings delimited by punctuation marks.

```
String[] tokens = "Java,C?C#,C++".split("[.,:;?]");
for (int i = 0; i < tokens.length; i++)
    System.out.println(tokens[i]);
```

  - In this example, the regular expression *[.,:;?]* specifies a pattern that matches **.**, **,**, **:**, **;**, or **?**.
  - Each of these characters is a delimiter for splitting the string. Thus, the string is split into *Java*, *C*, *C#*, and *C++,* which are stored in array *tokens*.

Formatting Strings

- The *String* class contains the static *format* method to return a formatted string.
- The syntax to invoke this method is:

  String.format(format, item1, item2, ..., itemk)

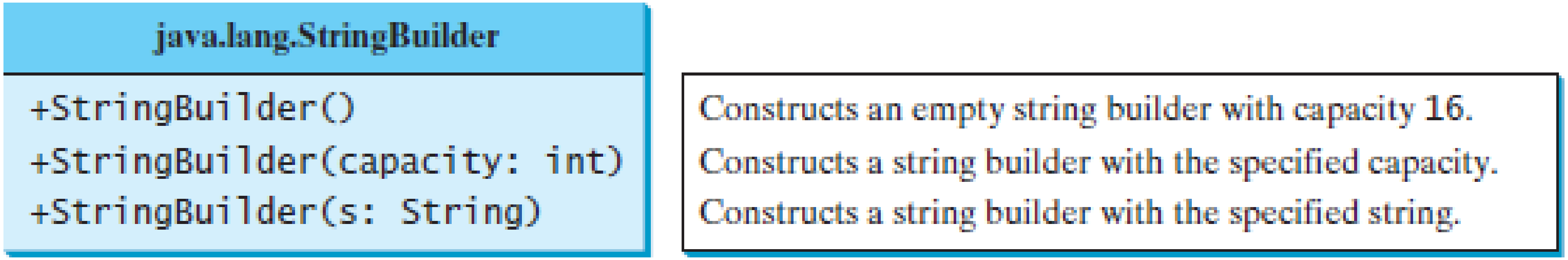
- This method is similar to the *printf* method except that the *format* method returns a formatted string, whereas the *printf* method displays a formatted string.
- Example:

```
String s = String.format("%7.2f%6d%-4s", 45.556, 14,
"AB");
System.out.println(s);
```

The *StringBuilder* and *StringBuffer* Classes

- In general, the *StringBuilder* and *StringBuffer* classes can be used wherever a string is used.
- *StringBuilder* and *StringBuffer* are more flexible than *String*.
  - You can add, insert, or append new contents into *StringBuilder* and *StringBuffer* objects, whereas the value of a *String* object is fixed once the string is created.
- The *StringBuilder* class is similar to *StringBuffer* except that the methods for modifying the buffer in *StringBuffer* are *synchronized*.
  - This means that only one task is allowed to execute the methods.

The *StringBuilder* and *StringBuffer* Classes (Constructors)



java.lang.StringBuilder

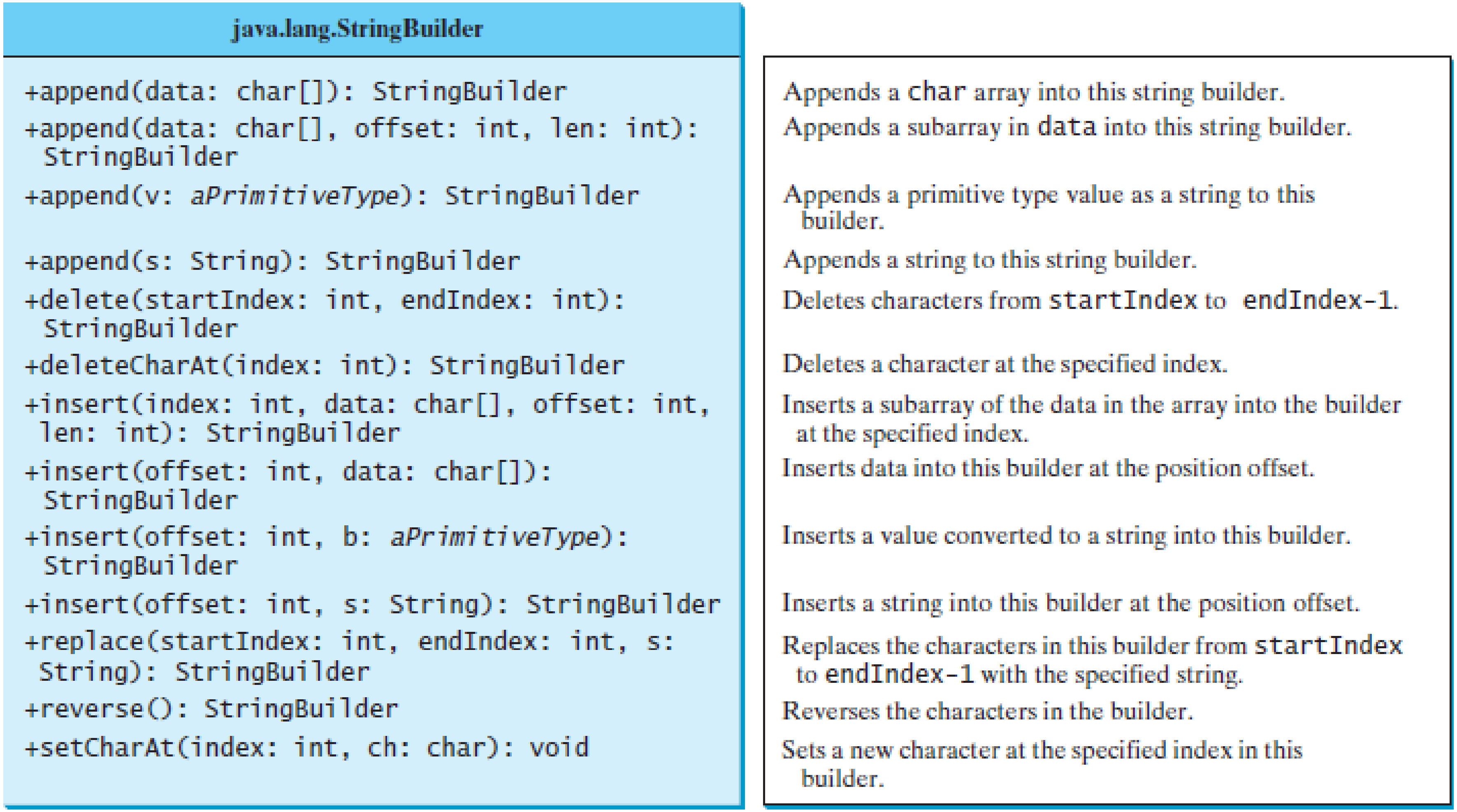
```
+StringBuilder()
+StringBuilder(capacity: int)
+StringBuilder(s: String)
```

Constructs an empty string builder with capacity 16.

Constructs a string builder with the specified capacity.

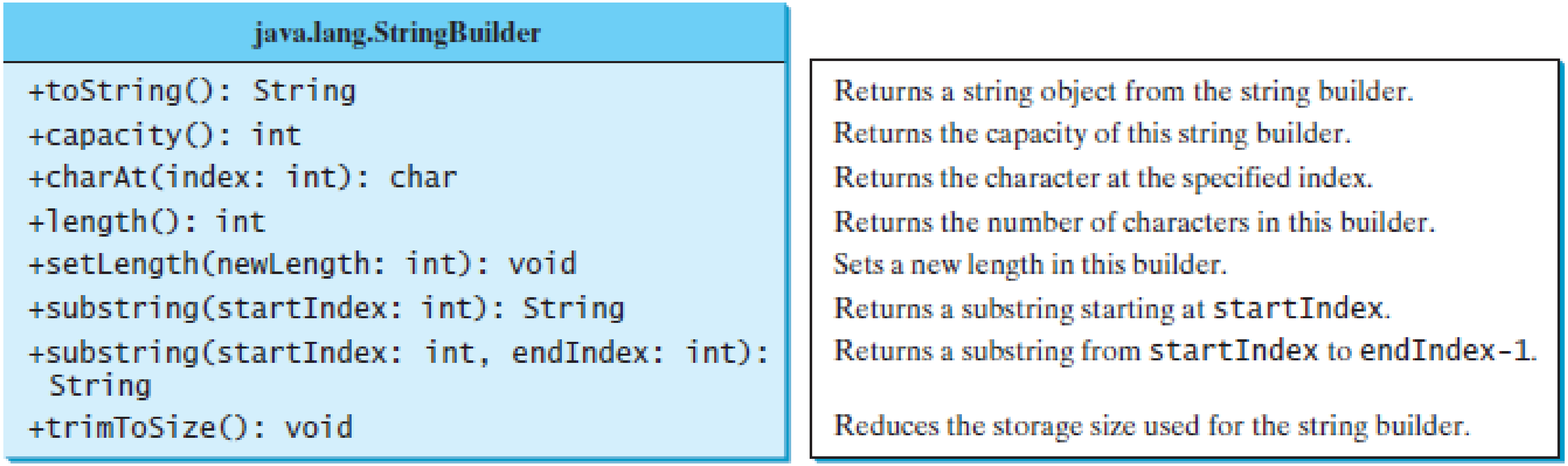
Constructs a string builder with the specified string.

The *StringBuilder* and *StringBuffer* Classes (Modifying Strings)



java.lang.StringBuilder	
+append(data: char[]): StringBuilder	Appends a char array into this string builder.
+append(data: char[], offset: int, len: int): StringBuilder	Appends a subarray in data into this string builder.
+append(v: *aPrimitiveType*): StringBuilder	Appends a primitive type value as a string to this builder.
+append(s: String): StringBuilder	Appends a string to this string builder.
+delete(startIndex: int, endIndex: int): StringBuilder	Deletes characters from startIndex to endIndex-1.
+deleteCharAt(index: int): StringBuilder	Deletes a character at the specified index.
+insert(index: int, data: char[], offset: int, len: int): StringBuilder	Inserts a subarray of the data in the array into the builder at the specified index.
+insert(offset: int, data: char[]): StringBuilder	Inserts data into this builder at the position offset.
+insert(offset: int, b: *aPrimitiveType*): StringBuilder	Inserts a value converted to a string into this builder.
+insert(offset: int, s: String): StringBuilder	Inserts a string into this builder at the position offset.
+replace(startIndex: int, endIndex: int, s: String): StringBuilder	Replaces the characters in this builder from startIndex to endIndex-1 with the specified string.
+reverse(): StringBuilder	Reverses the characters in the builder.
+setCharAt(index: int, ch: char): void	Sets a new character at the specified index in this builder.

The *StringBuilder* and *StringBuffer* Classes (Other Methods)



java.lang.StringBuilder	
`+toString(): String`	Returns a string object from the string builder.
`+capacity(): int`	Returns the capacity of this string builder.
`+charAt(index: int): char`	Returns the character at the specified index.
`+length(): int`	Returns the number of characters in this builder.
`+setLength(newLength: int): void`	Sets a new length in this builder.
`+substring(startIndex: int): String`	Returns a substring starting at `startIndex`.
`+substring(startIndex: int, endIndex: int): String`	Returns a substring from `startIndex` to `endIndex-1`.
`+trimToSize(): void`	Reduces the storage size used for the string builder.

Object-Oriented Problem Solving

Generics

Based on chapter 19 of "Introduction to Java Programming" by Y. Daniel Liang.

Eng. Asma Abdel Karim
Computer Engineering Department

Outline

- Motivation and Benefits (19.2)
- java.util.ArrayList
- Defining Generic Classes (19.3)
- Generic Methods (19.4)

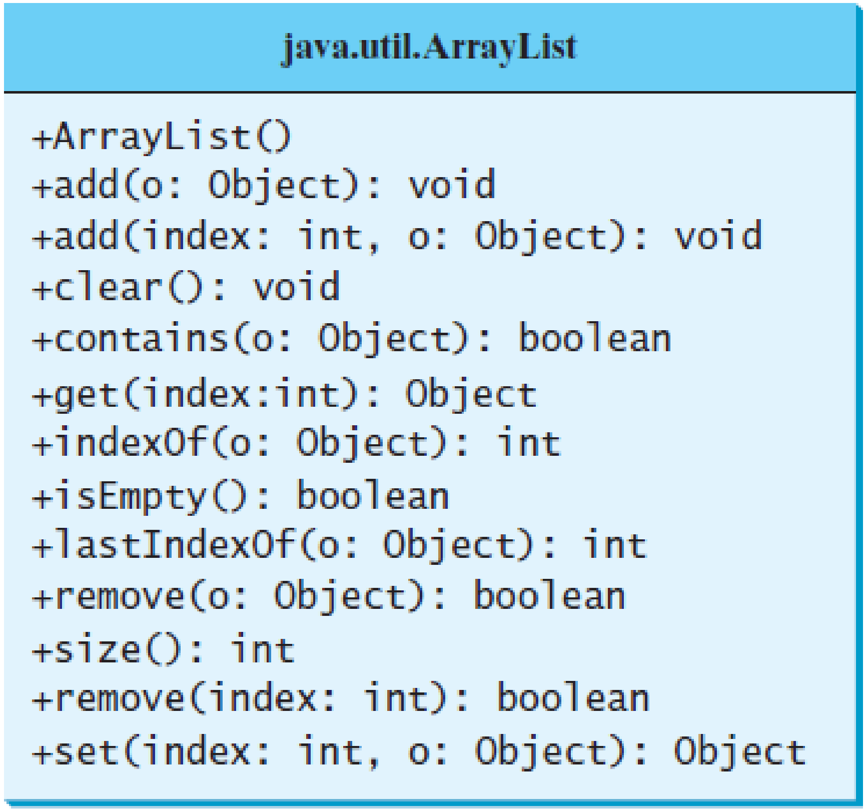
Motivation and Benefits

- *Generics* let you parameterize types.
- With this capability, you can define a class or a method with *generic types* that the compiler can replace with *concrete types*.
- The motivation for using Java generics is to detect errors at compile time.
- Java has allowed defining generic classes, interfaces, and methods since JDK 1.5.

java.util.ArrayList

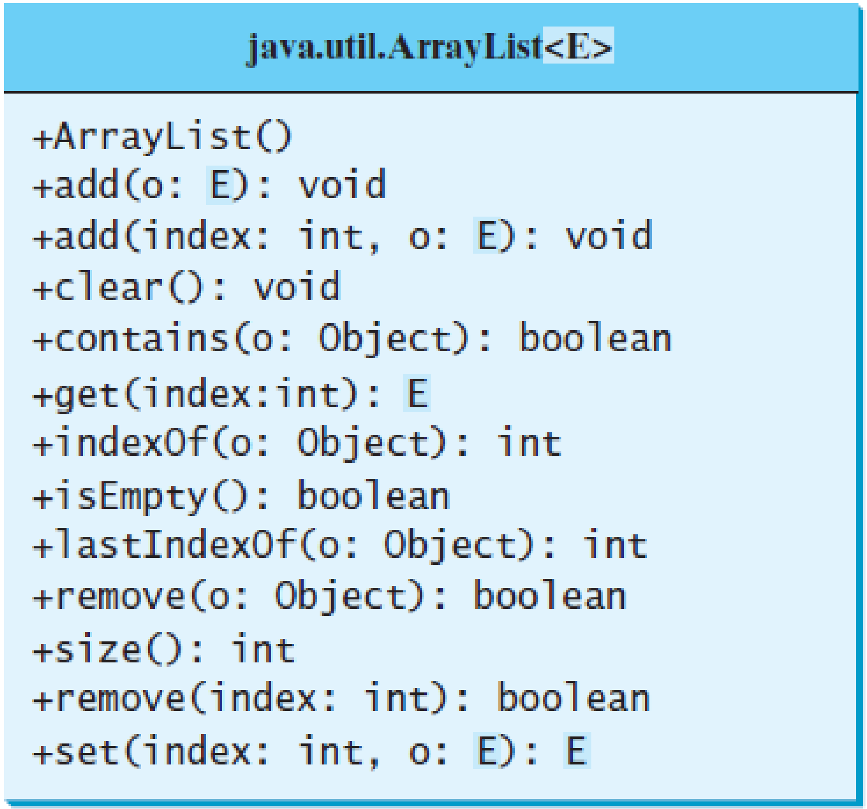
- An example of a generic class is the *ArrayList* class of the *java.util* package.
- An *ArrayList* object can be used to store a list of objects.
- So far, we used arrays to store objects.
  - Once the array is created, its size is fixed.
  - The *ArrayList* class can be used to store an unlimited number of objects.

java.util.ArrayList (Cont.)



```
java.util.ArrayList

+ArrayList()
+add(o: Object): void
+add(index: int, o: Object): void
+clear(): void
+contains(o: Object): boolean
+get(index:int): Object
+indexOf(o: Object): int
+isEmpty(): boolean
+lastIndexOf(o: Object): int
+remove(o: Object): boolean
+size(): int
+remove(index: int): boolean
+set(index: int, o: Object): Object
```

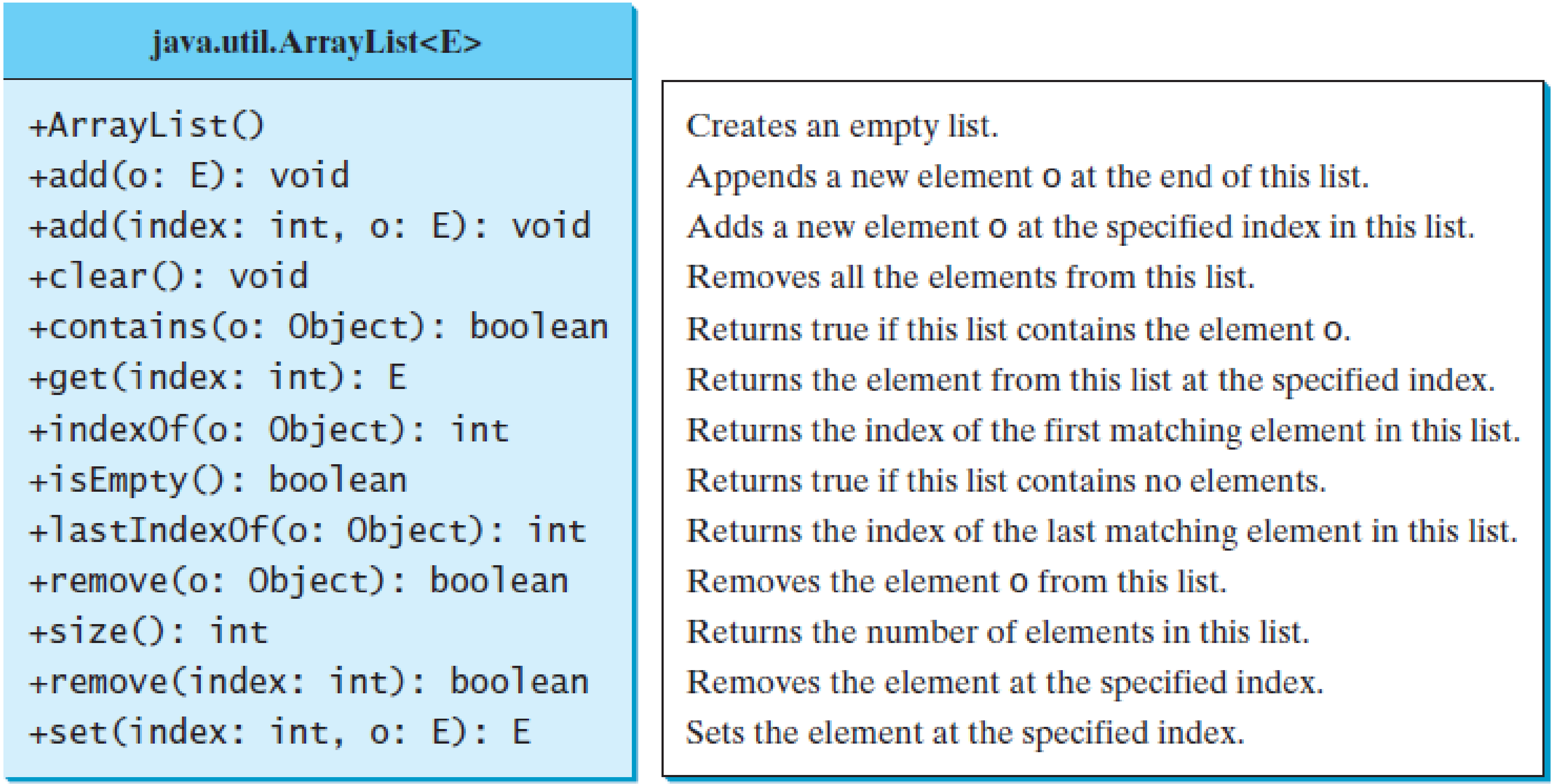
(a) ArrayList before JDK 1.5



```
java.util.ArrayList<E>

+ArrayList()
+add(o: E): void
+add(index: int, o: E): void
+clear(): void
+contains(o: Object): boolean
+get(index:int): E
+indexOf(o: Object): int
+isEmpty(): boolean
+lastIndexOf(o: Object): int
+remove(o: Object): boolean
+size(): int
+remove(index: int): boolean
+set(index: int, o: E): E
```

(b) ArrayList since JDK 1.5

java.util.ArrayList (Cont.)



java.util.ArrayList<E>	
+ArrayList()	Creates an empty list.
+add(o: E): void	Appends a new element o at the end of this list.
+add(index: int, o: E): void	Adds a new element o at the specified index in this list.
+clear(): void	Removes all the elements from this list.
+contains(o: Object): boolean	Returns true if this list contains the element o.
+get(index: int): E	Returns the element from this list at the specified index.
+indexOf(o: Object): int	Returns the index of the first matching element in this list.
+isEmpty(): boolean	Returns true if this list contains no elements.
+lastIndexOf(o: Object): int	Returns the index of the last matching element in this list.
+remove(o: Object): boolean	Removes the element o from this list.
+size(): int	Returns the number of elements in this list.
+remove(index: int): boolean	Removes the element at the specified index.
+set(index: int, o: E): E	Sets the element at the specified index.

java.util.ArrayList (Cont.)

- The *ArrayList* class is a generic class with a generic type *E*.
  - By convention, a single capital letter such as *E* or *T* is used to denote a formal generic type.
- You can specify a concrete type to replace *E* when creating an ArrayList.
  - Replacing a generic type is called a generic instantiation.
- Examples:

```
ArrayList<String> cities = new ArrayList<String>();
ArrayList<Date> dates = new ArrayList<Date>();
```

java.util.ArrayList (Cont.)

- For example, if an *ArrayList* of *Strings* is created using the following statement:

```
ArrayList <String> list = new ArrayList<String>();
```

- Only Strings can now be added to the list as follows:

```
list.add(“Red”);
```

- If you attempt to add a non-string, a compilation error will occur. For example, the following statement is not legal:

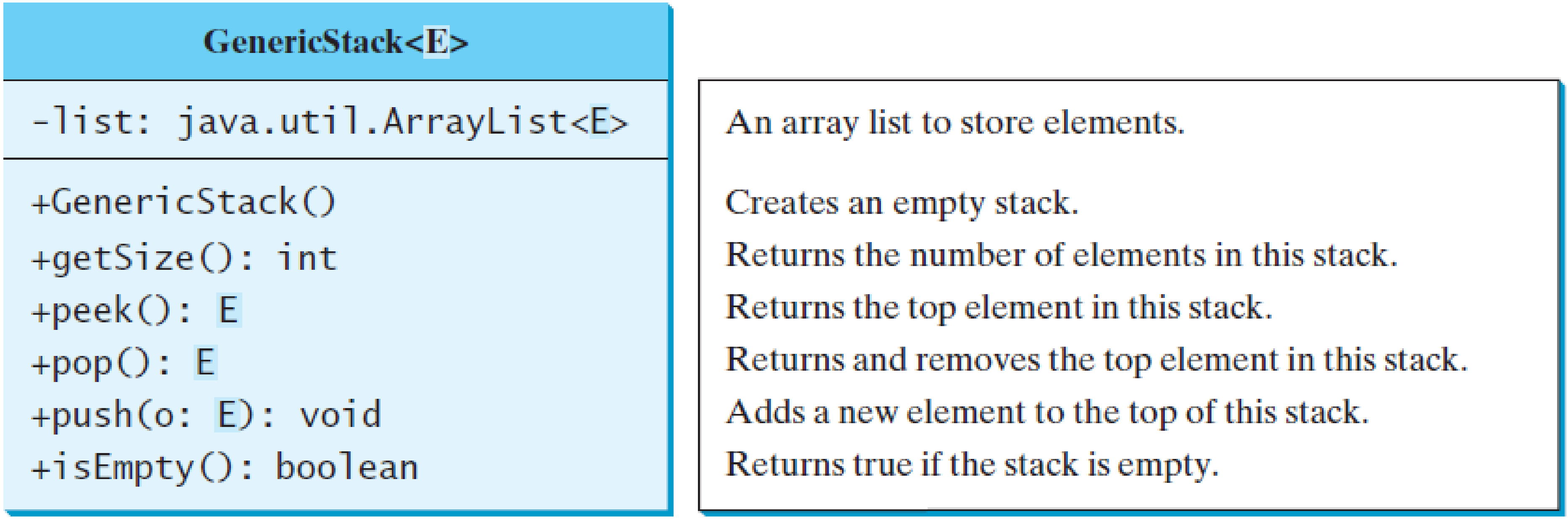
```
list.add(new Date());
```

java.util.ArrayList (Cont.)

- Generic types must be reference types.
- You cannot replace a generic type with a primitive type such as *int*, *double*, or *char*.
- For example, the following statement is wrong:

ArrayList <int> intList = new ArrayLisy<int> ();

Defining Generic Classes
(Example: The GenericStack Class)



GenericStack<E>

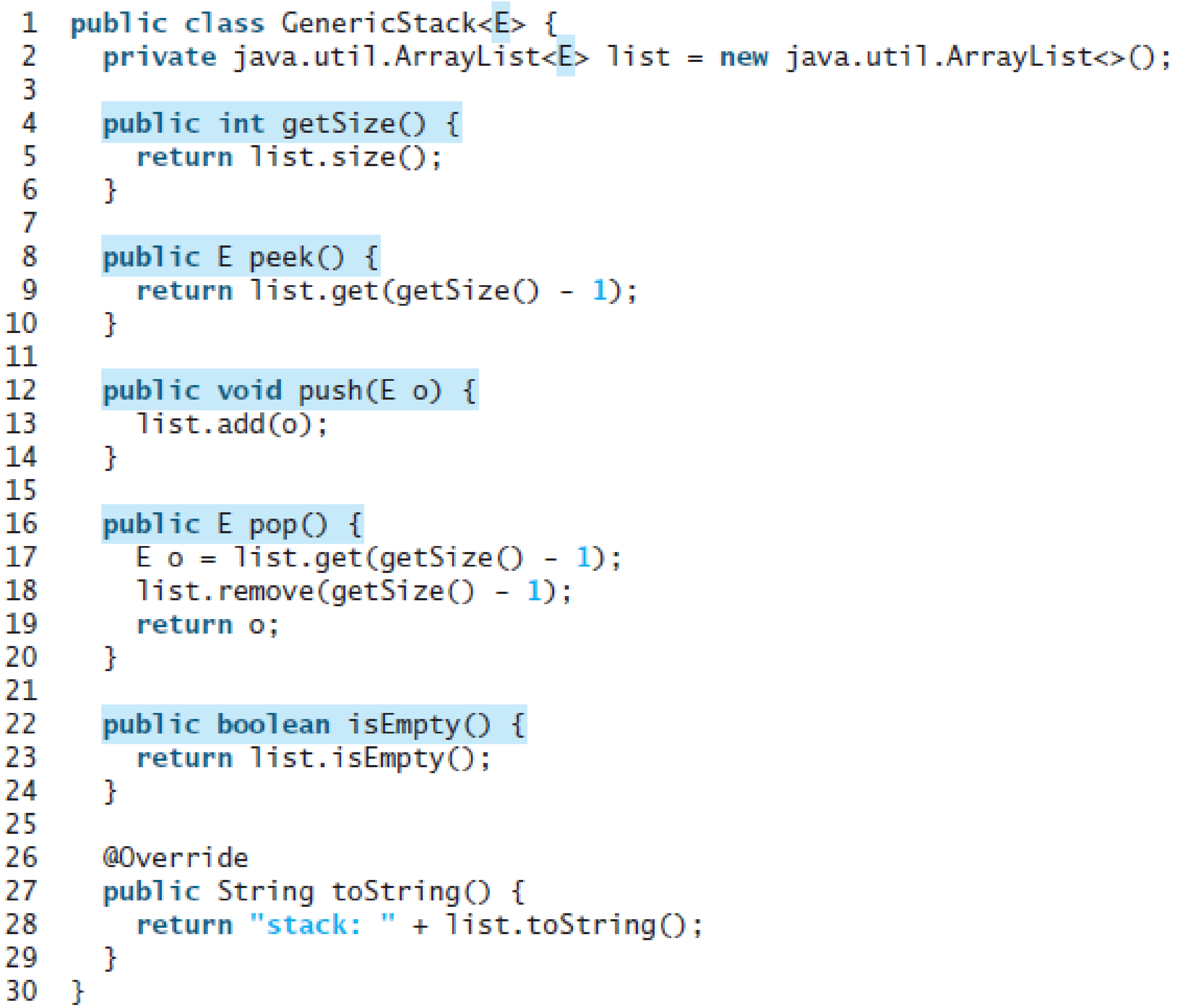
-list: java.util.ArrayList<E>

+GenericStack()
+getSize(): int
+peek(): E
+pop(): E
+push(o: E): void
+isEmpty(): boolean

An array list to store elements.

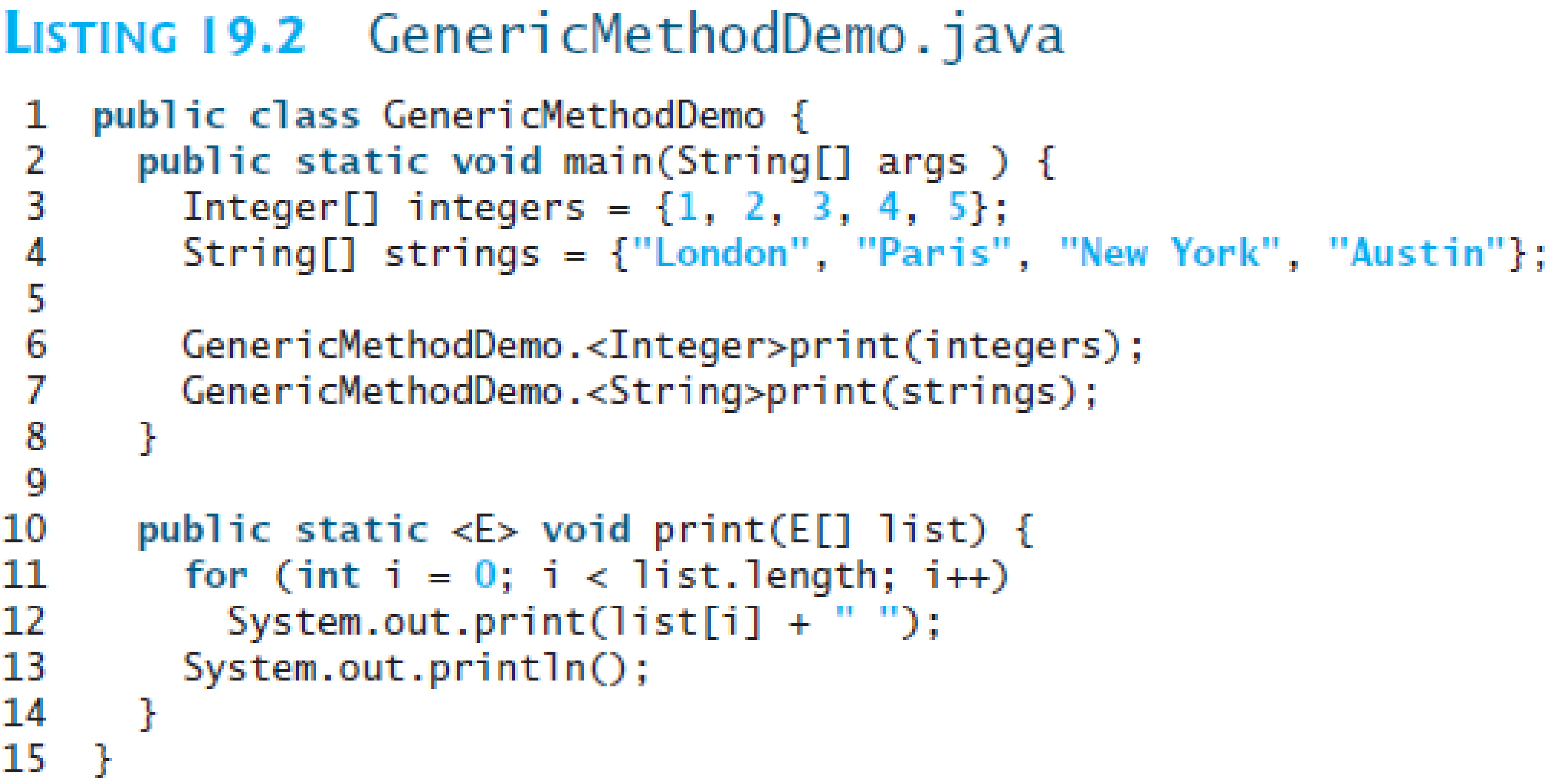
Creates an empty stack.
Returns the number of elements in this stack.
Returns the top element in this stack.
Returns and removes the top element in this stack.
Adds a new element to the top of this stack.
Returns true if the stack is empty.

Example: GenericStack.java (Cont.)



```java
public class GenericStack<E> {
  private java.util.ArrayList<E> list = new java.util.ArrayList<>();

  public int getSize() {
    return list.size();
  }

  public E peek() {
    return list.get(getSize() - 1);
  }

  public void push(E o) {
    list.add(o);
  }

  public E pop() {
    E o = list.get(getSize() - 1);
    list.remove(getSize() - 1);
    return o;
  }

  public boolean isEmpty() {
    return list.isEmpty();
  }

  @Override
  public String toString() {
    return "stack: " + list.toString();
  }
}
```

Generic Methods



Listing 19.2 GenericMethodDemo.java

```java
public class GenericMethodDemo {
  public static void main(String[] args ) {
    Integer[] integers = {1, 2, 3, 4, 5};
    String[] strings = {"London", "Paris", "New York", "Austin"};

    GenericMethodDemo.<Integer>print(integers);
    GenericMethodDemo.<String>print(strings);
  }

  public static <E> void print(E[] list) {
    for (int i = 0; i < list.length; i++)
      System.out.print(list[i] + " ");
    System.out.println();
  }
}
```

Generic Methods (Cont.)

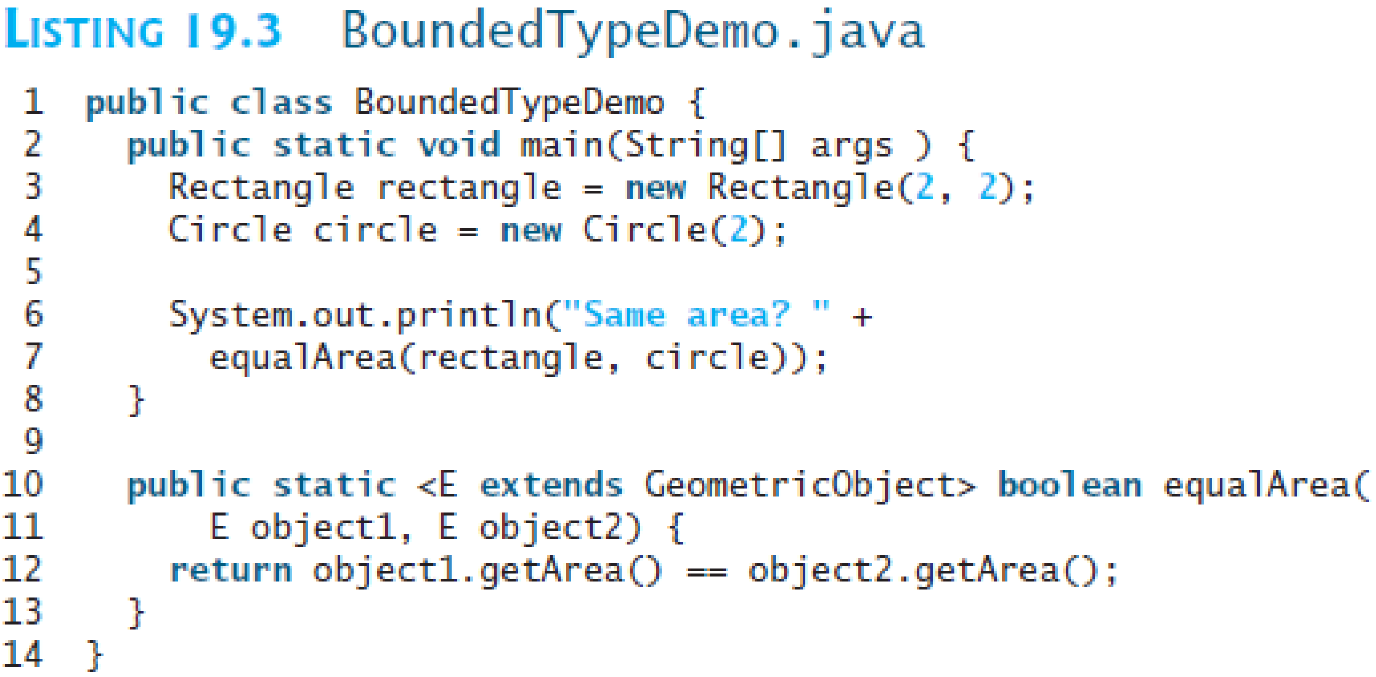
- To invoke a generic method, prefix the method name with the actual type in angle brackets.
- For example,

```
GenericMethodDemo.<Integer>print(integers);
GenericMethodDemo.<String>print(strings);
```

- or simply invoke it as follows:

```
print(integers);
print(strings);
```

Generic Methods (Cont.)



Listing 19.3 BoundedTypeDemo.java

```java
public class BoundedTypeDemo {
  public static void main(String[] args ) {
    Rectangle rectangle = new Rectangle(2, 2);
    Circle circle = new Circle(2);

    System.out.println("Same area? " +
      equalArea(rectangle, circle));
  }

  public static <E extends GeometricObject> boolean equalArea(
      E object1, E object2) {
    return object1.getArea() == object2.getArea();
  }
}
```

- A generic type can be specified as a subtype of another type. Such a generic type is called *bounded*.

Object-Oriented Problem Solving

Abstract Classes & Interfaces

Based on Chapter 13 of "Introduction to Java Programming" by Y. Daniel Liang.

Eng. Asma Abdel Karim
Computer Engineering Department

Outline

- Abstract Classes (13.2)
- Case Study: The Abstract Number Class (13.3)
- Interfaces (13.5)
- The Comparable Interface (13.6)
- Interfaces vs. Abstract Classes (13.8)
- Class Design Guidelines (13.10)

Introduction

- In the inheritance hierarchy, classes become more specific and concrete with each new subclass.
- If you move from a subclass back up to a superclass, the classes become more general and less specific.
- Class design should ensure that a superclass contains common features of its subclasses.

What are Abstract Methods?

- In the example of the previous section, *GeometricObject* was defined as the superclass for Circle and Rectangle.
- Bot Circle and Rectangle contain *getArea()* and *getPerimeter()* methods.
- It is better to define the *getArea()* and *getPerimeter()* methods in the *GeometricObject* class.
- However, these methods cannot be implemented in the *GeometricObject* class, because their implementation depends on the specific type of a geometric object.

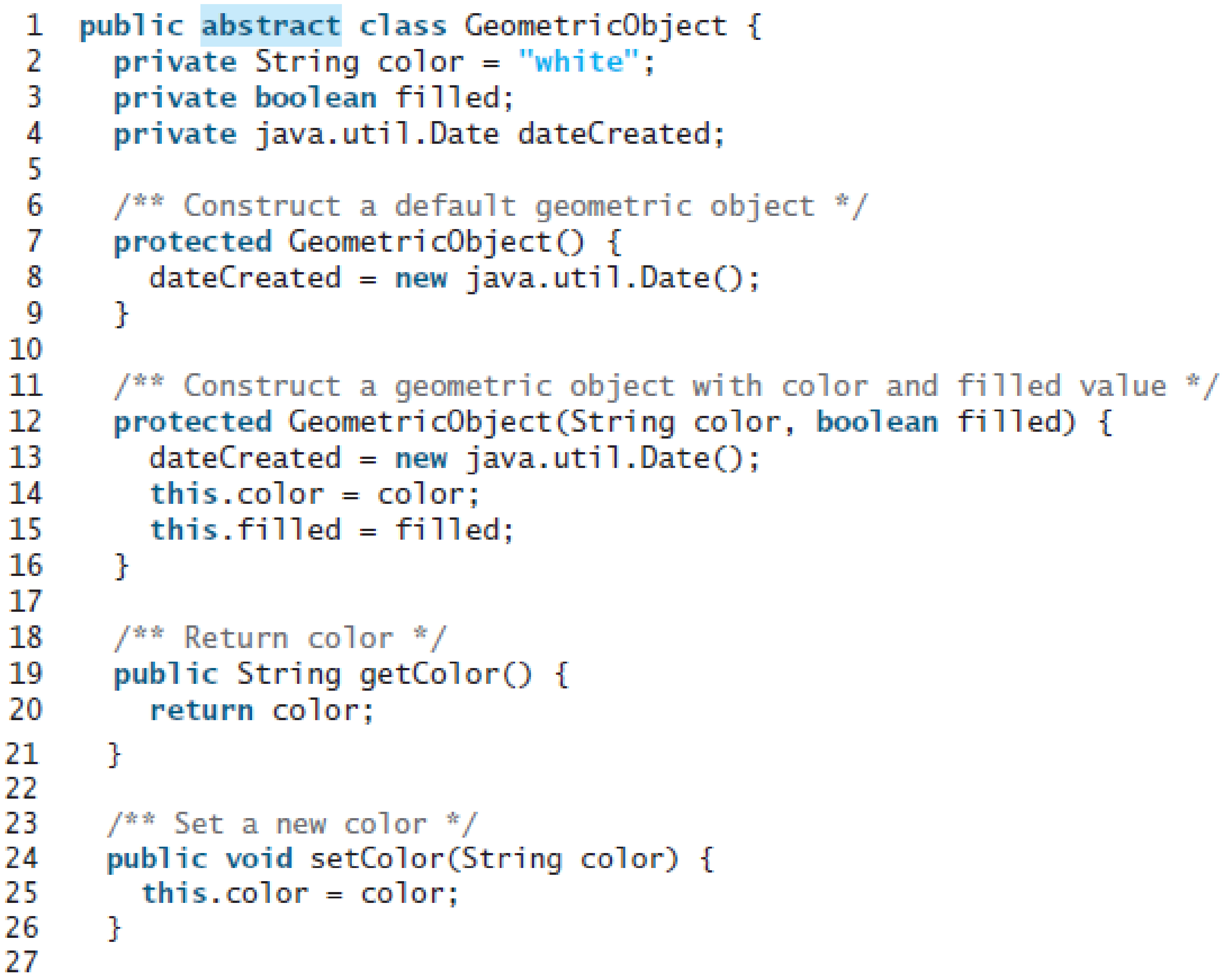
What are Abstract Methods? (Cont.)

- Such methods can be defined in the superclass as *abstract methods*.
  - An abstract method is defined without an implementation in the superclass.
    - Its implementation is provided by the subclasses.
  - Abstract methods are denoted using the *abstract* modifier in the method header.

What are Abstract Classes?

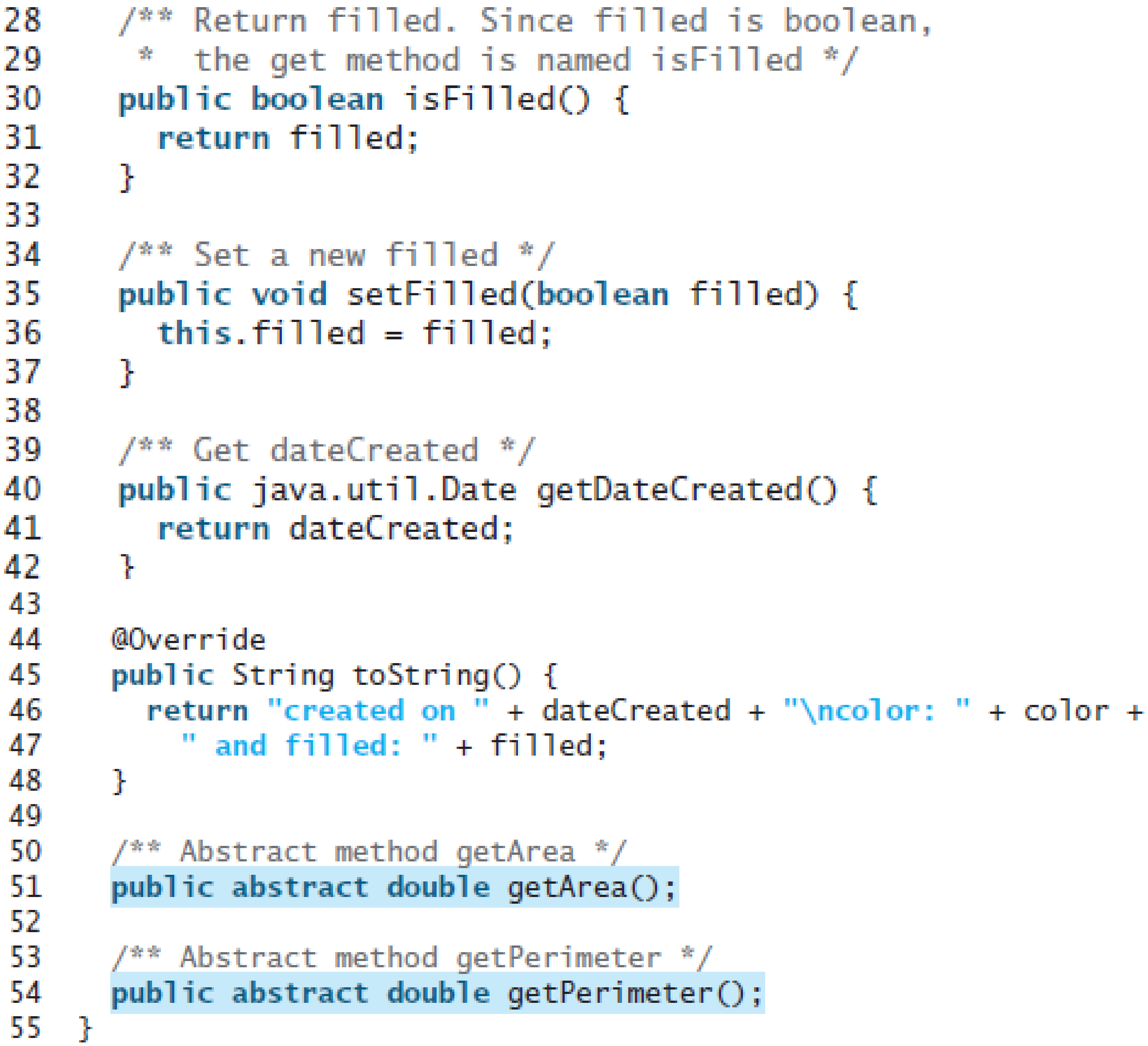
- A class that contains at least one abstract method must be defined as an *abstract class*.
  - Abstract classes are denoted using the *abstract* modifier in the class header.
- Abstract classes are like regular classes, <u>but you cannot create instances of abstract classes using the *new* operator.</u>
  - Constructors of an abstract class are defined as protected, because they are used only by subclasses.

Abstract Classes & Methods: Example



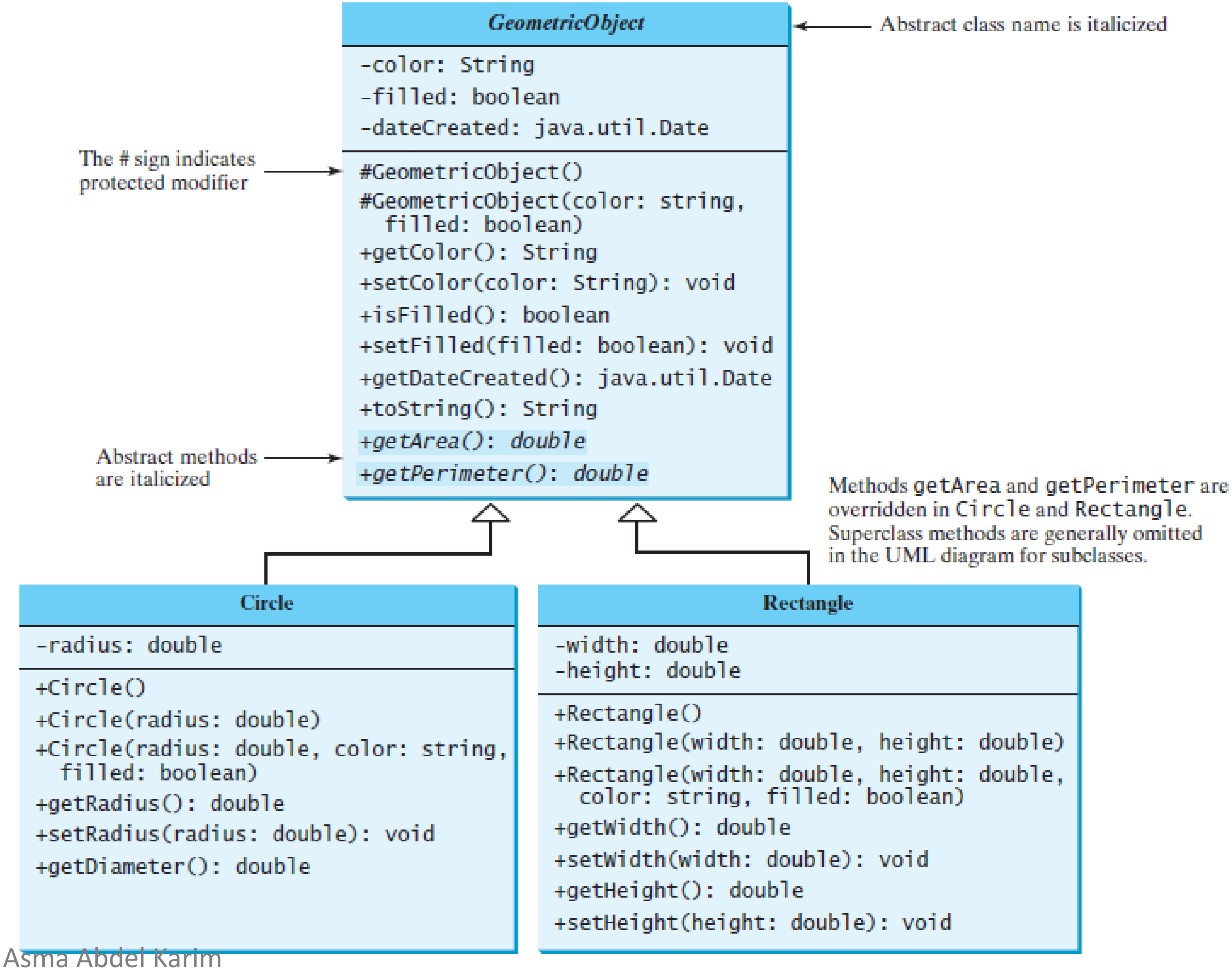
```java
public abstract class GeometricObject {
  private String color = "white";
  private boolean filled;
  private java.util.Date dateCreated;

  /** Construct a default geometric object */
  protected GeometricObject() {
    dateCreated = new java.util.Date();
  }

  /** Construct a geometric object with color and filled value */
  protected GeometricObject(String color, boolean filled) {
    dateCreated = new java.util.Date();
    this.color = color;
    this.filled = filled;
  }

  /** Return color */
  public String getColor() {
    return color;
  }

  /** Set a new color */
  public void setColor(String color) {
    this.color = color;
  }

```

Abstract Classes & Methods: Example (Cont.)



```
  /** Return filled. Since filled is boolean,
   *  the get method is named isFilled */
  public boolean isFilled() {
    return filled;
  }

  /** Set a new filled */
  public void setFilled(boolean filled) {
    this.filled = filled;
  }

  /** Get dateCreated */
  public java.util.Date getDateCreated() {
    return dateCreated;
  }

  @Override
  public String toString() {
    return "created on " + dateCreated + "\ncolor: " + color +
      " and filled: " + filled;
  }

  /** Abstract method getArea */
  public abstract double getArea();

  /** Abstract method getPerimeter */
  public abstract double getPerimeter();
}
```

Abstract Classes and Methods: UML Diagram



GeometricObject

```
-color: String
-filled: boolean
-dateCreated: java.util.Date
```

```
#GeometricObject()
#GeometricObject(color: string,
  filled: boolean)
+getColor(): String
+setColor(color: String): void
+isFilled(): boolean
+setFilled(filled: boolean): void
+getDateCreated(): java.util.Date
+toString(): String
+getArea(): double
+getPerimeter(): double
```

Abstract class name is italicized

The # sign indicates protected modifier

Abstract methods are italicized

Methods getArea and getPerimeter are overridden in Circle and Rectangle. Superclass methods are generally omitted in the UML diagram for subclasses.

Circle

```
-radius: double
```

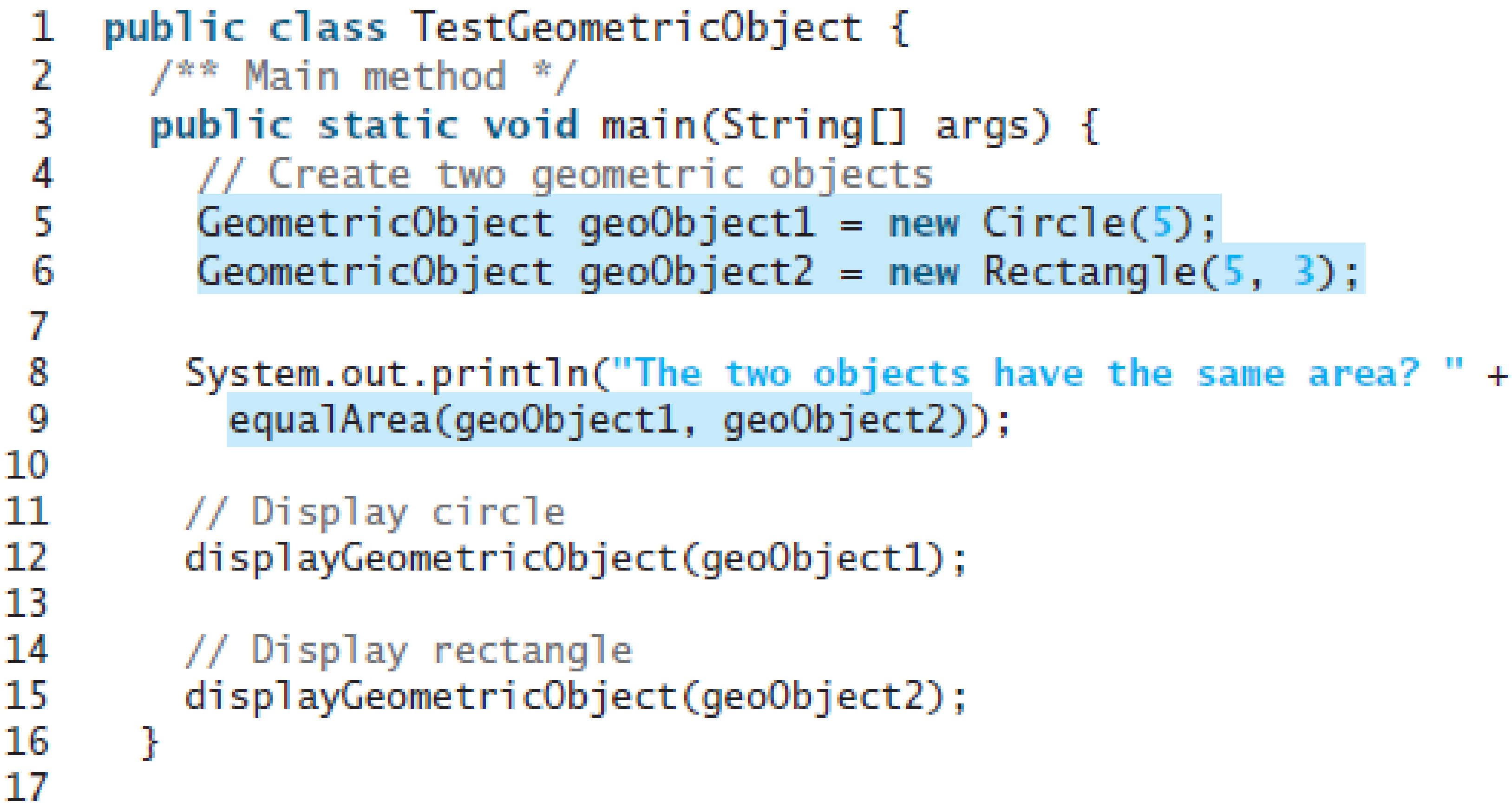
```
+Circle()
+Circle(radius: double)
+Circle(radius: double, color: string,
  filled: boolean)
+getRadius(): double
+setRadius(radius: double): void
+getDiameter(): double
```

Rectangle

```
-width: double
-height: double
```

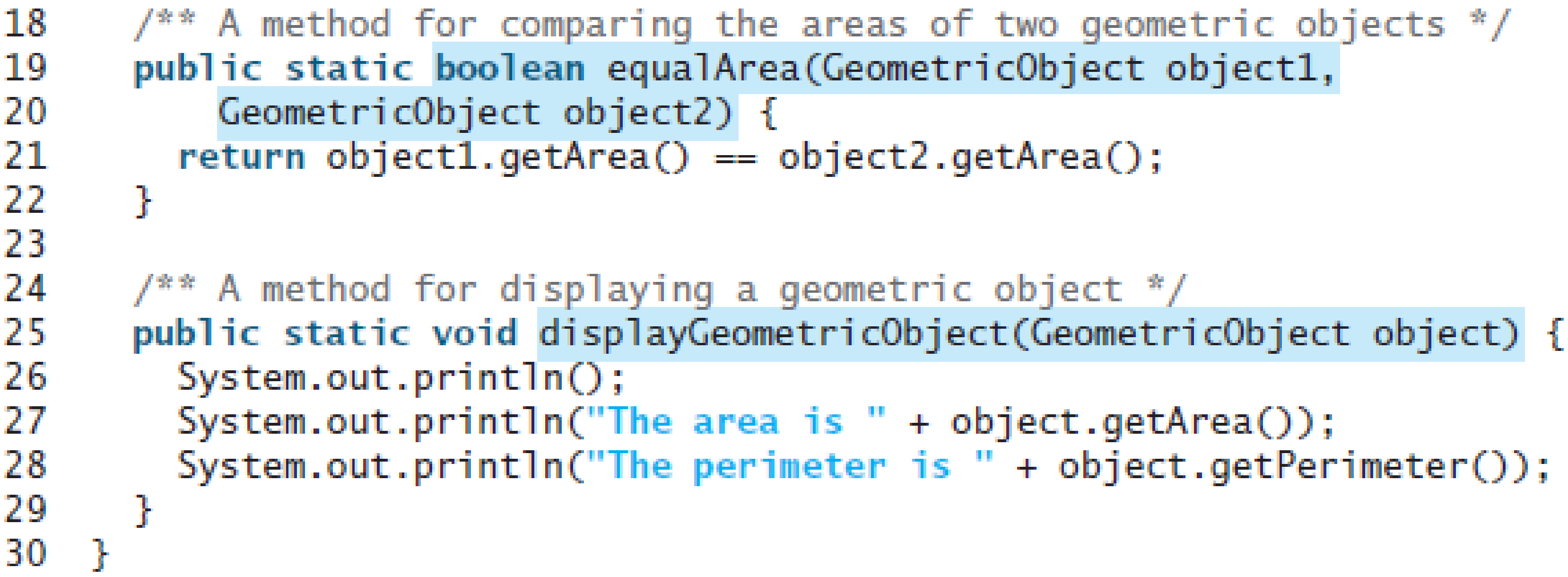
```
+Rectangle()
+Rectangle(width: double, height: double)
+Rectangle(width: double, height: double,
  color: string, filled: boolean)
+getWidth(): double
+setWidth(width: double): void
+getHeight(): double
+setHeight(height: double): void
```

Why Abstract Methods? Example



```java
public class TestGeometricObject {
  /** Main method */
  public static void main(String[] args) {
    // Create two geometric objects
    GeometricObject geoObject1 = new Circle(5);
    GeometricObject geoObject2 = new Rectangle(5, 3);

    System.out.println("The two objects have the same area? " +
      equalArea(geoObject1, geoObject2));

    // Display circle
    displayGeometricObject(geoObject1);

    // Display rectangle
    displayGeometricObject(geoObject2);
  }

```

Why Abstract Methods? Example



```
  /** A method for comparing the areas of two geometric objects */
  public static boolean equalArea(GeometricObject object1,
      GeometricObject object2) {
    return object1.getArea() == object2.getArea();
  }

  /** A method for displaying a geometric object */
  public static void displayGeometricObject(GeometricObject object) {
    System.out.println();
    System.out.println("The area is " + object.getArea());
    System.out.println("The perimeter is " + object.getPerimeter());
  }
}
```

Important Notes Regarding Abstract Classes and Methods (1)

- An abstract method cannot be contained in a non-abstract class.
  - If a subclass of an abstract superclass does not implement all abstract methods, the subclass must be defined as abstract.
  - Abstract methods are non-static.
- An abstract class cannot be instantiated using the new operator, but:
  - You still can define its constructors which are invoked in the constructors of its subclasses.
  - An abstract class can be used as a data type.
    - Therefore, the following statement, which creates an array whose elements are of the GeometricObject type is correct:

GeometricObject [] Objects = new GeometricObject[10];

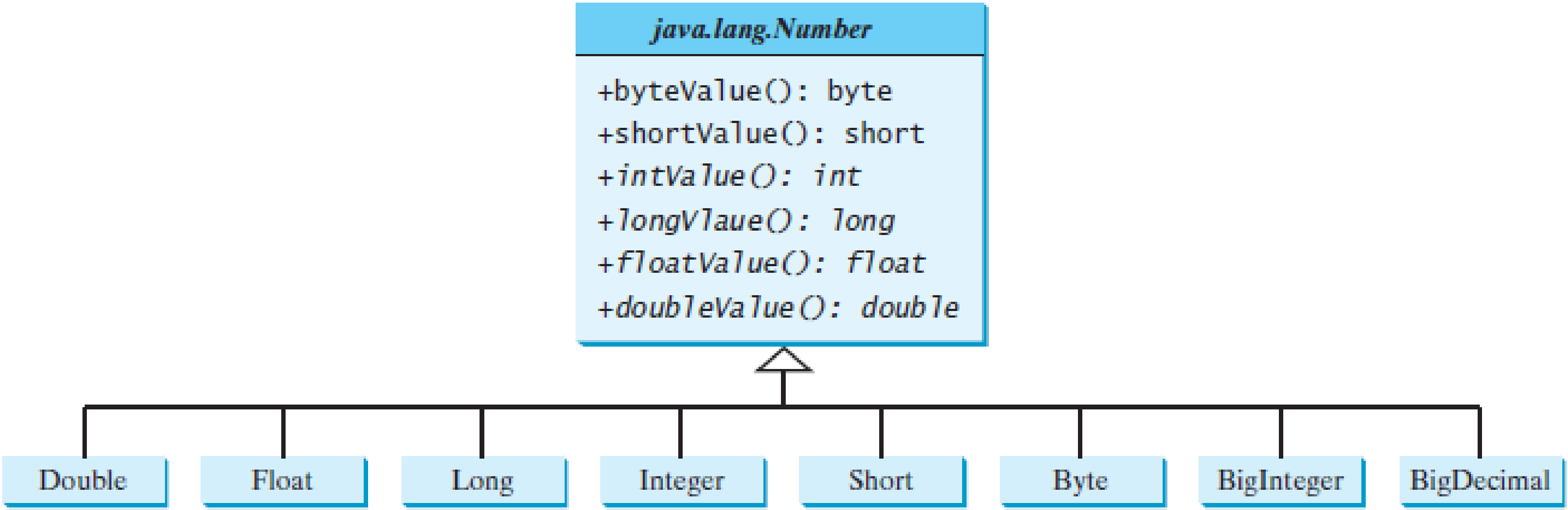
Important Notes Regarding Abstract Classes and Methods (2)

- A class that contains abstract methods must be abstract.
  - However, it is possible to define an abstract class that does not contain any abstract methods.
  - In this case, you cannot create instances of the class using the *new* operator.
    - This class is used as a base class for defining subclasses.
- A subclass can be abstract even if it's superclass is concrete.
  - For example, the *Object* class is concrete, but its subclasses may be abstract, such as *GeometricObject* in the previous example.

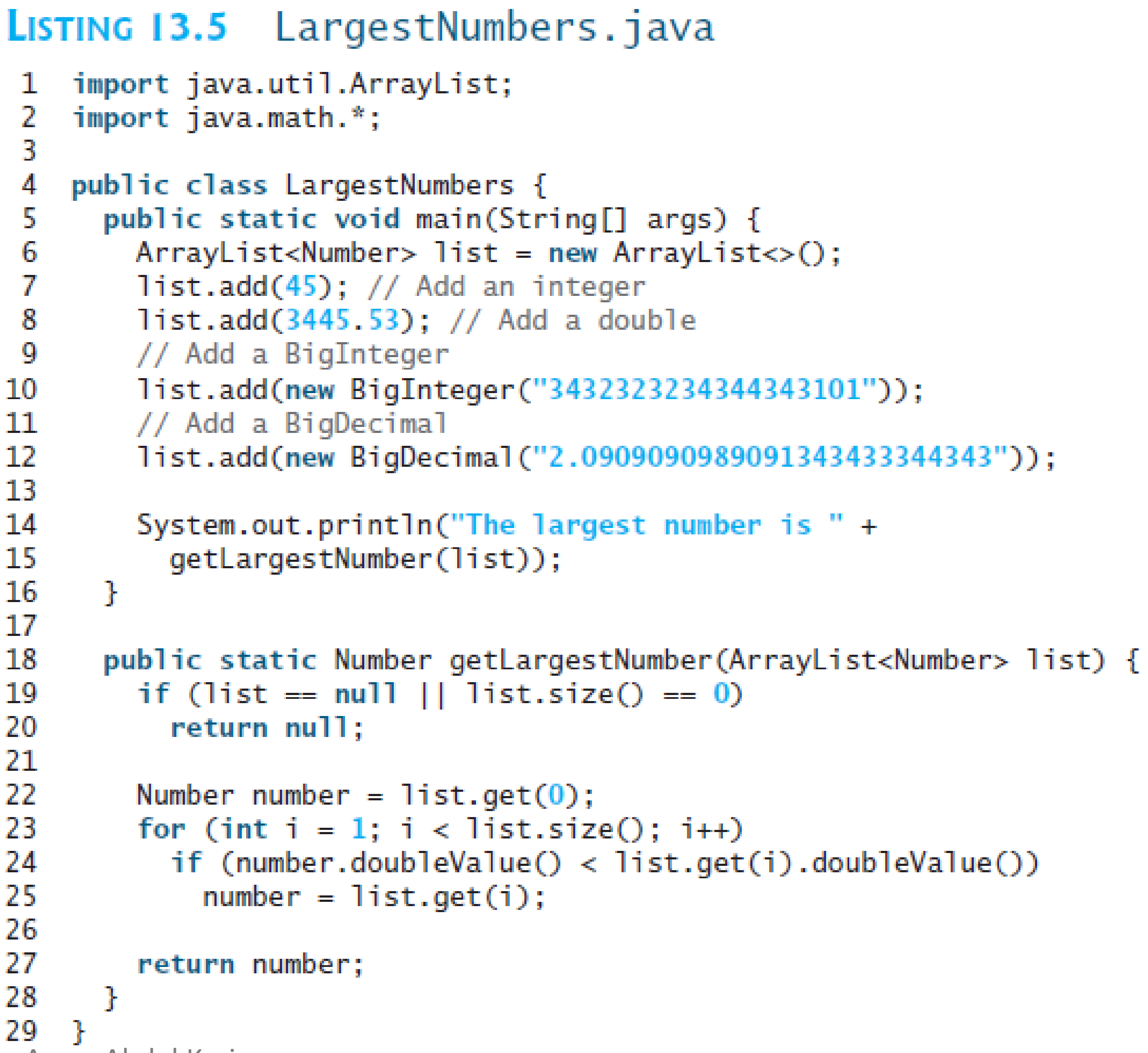
Important Notes Regarding Abstract Classes and Methods (3)

- A subclass can override a method from its superclass to define it as abstract.
  - This is *very unusual*, but it is useful when the implementation of the method in the superclass becomes invalid in the subclass.
  - In this case, the subclass must be defined as abstract.

Case Study: The Abstract Number Class



Case Study: The Abstract Number Class



LISTING 13.5 LargestNumbers.java

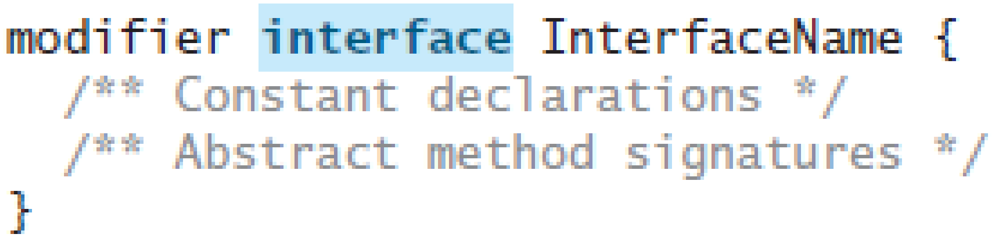
```java
import java.util.ArrayList;
import java.math.*;

public class LargestNumbers {
  public static void main(String[] args) {
    ArrayList<Number> list = new ArrayList<>();
    list.add(45); // Add an integer
    list.add(3445.53); // Add a double
    // Add a BigInteger
    list.add(new BigInteger("3432323234344343101"));
    // Add a BigDecimal
    list.add(new BigDecimal("2.0909090989091343433344343"));

    System.out.println("The largest number is " +
      getLargestNumber(list));
  }

  public static Number getLargestNumber(ArrayList<Number> list) {
    if (list == null || list.size() == 0)
      return null;

    Number number = list.get(0);
    for (int i = 1; i < list.size(); i++)
      if (number.doubleValue() < list.get(i).doubleValue())
        number = list.get(i);

    return number;
  }
}
```

Interfaces

- An *interface* is a class-like construct that contains only constants and abstract methods.
- The intent of interfaces is to define common behavior for related or unrelated classes.
- An interface is treated like a special class in Java.
  - Each interface is compiled into a separate bytecode file.
  - You can use an interface more or less the same way you use an abstract class.
    - An interface can be used as a data type for a reference variable.
    - You cannot create an instance from an interface with the new operator.

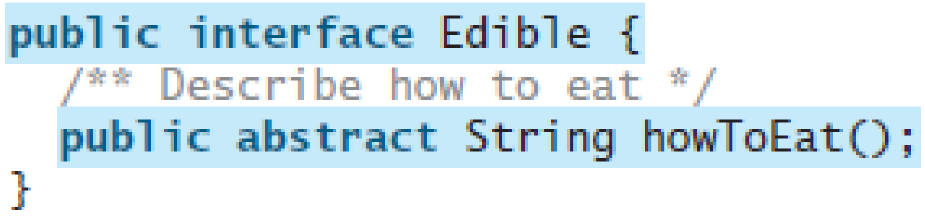
Interfaces (Cont.)

- To distinguish an interface from a class, Java uses the following syntax to define an interface:



```
modifier interface InterfaceName {
  /** Constant declarations */
  /** Abstract method signatures */
}
```

- Example:

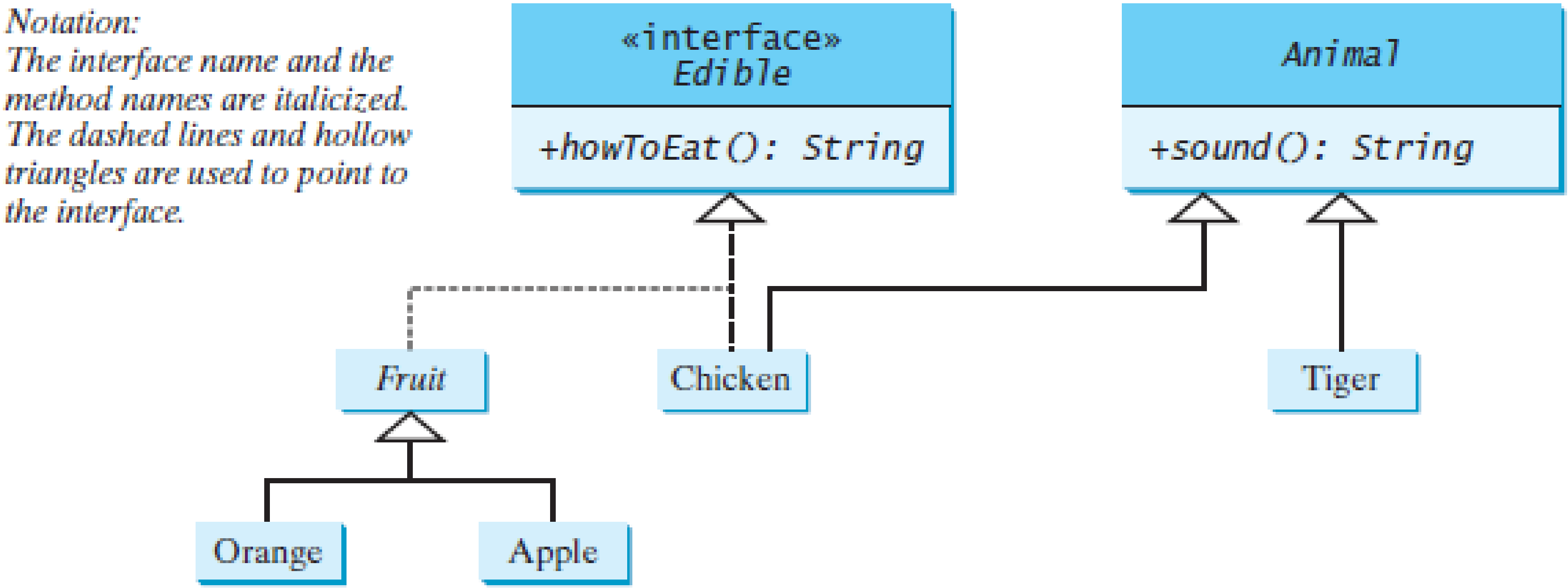


```
public interface Edible {
  /** Describe how to eat */
  public abstract String howToEat();
}
```

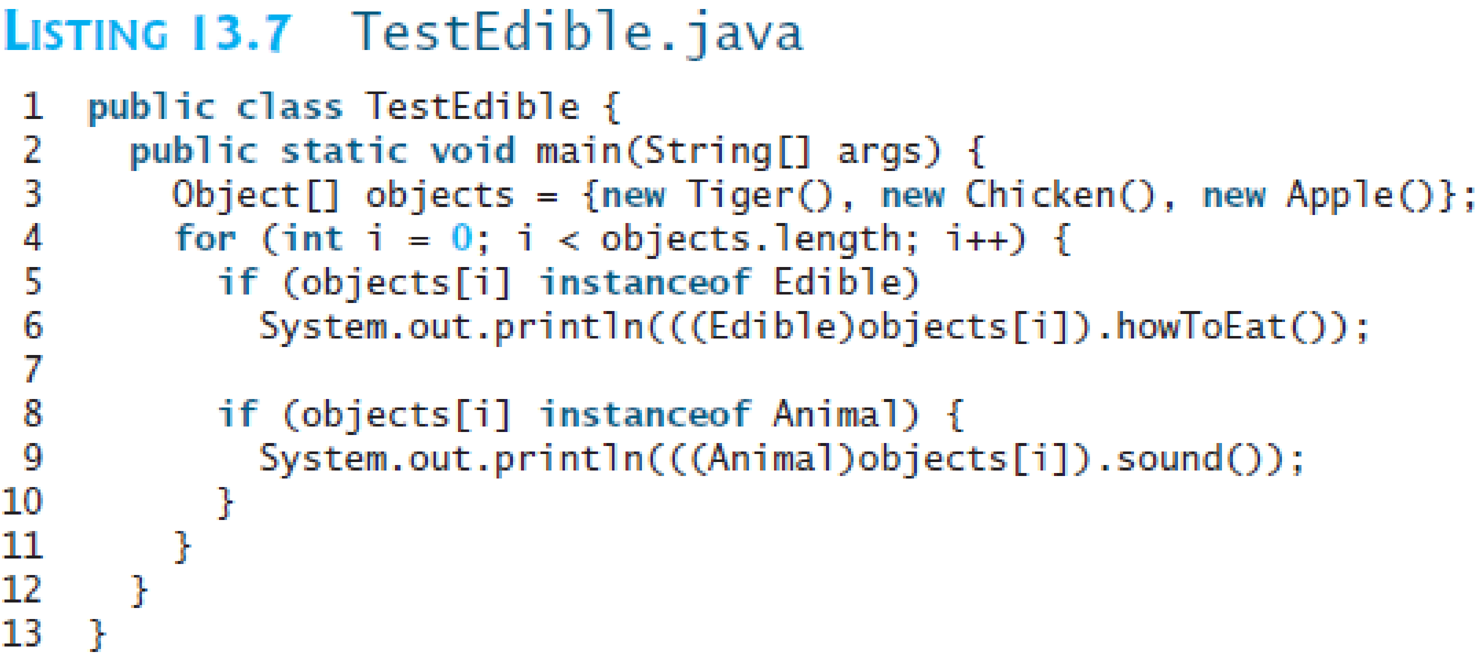
Implementing an Interface

- You can use an interface to specify the behavior of an object, by letting the class for the object *implement* this interface using the *implements* keyword.
  - When a class implements an interface, it implements all the methods defined in the interface with the exact signature and return type.
- The relationship between an interface and a class that implements it is known as *interface inheritance*.
  - Since *interface inheritance* and *class inheritance* are essentially the same, both are referred to as *inheritance*.

Implementing an Interface (Example)


Notation:
The interface name and the method names are italicized. The dashed lines and hollow triangles are used to point to the interface.
«interface»
Edible
+howToEat(): String
Animal
+sound(): String
Fruit
Chicken
Tiger
Orange
Apple

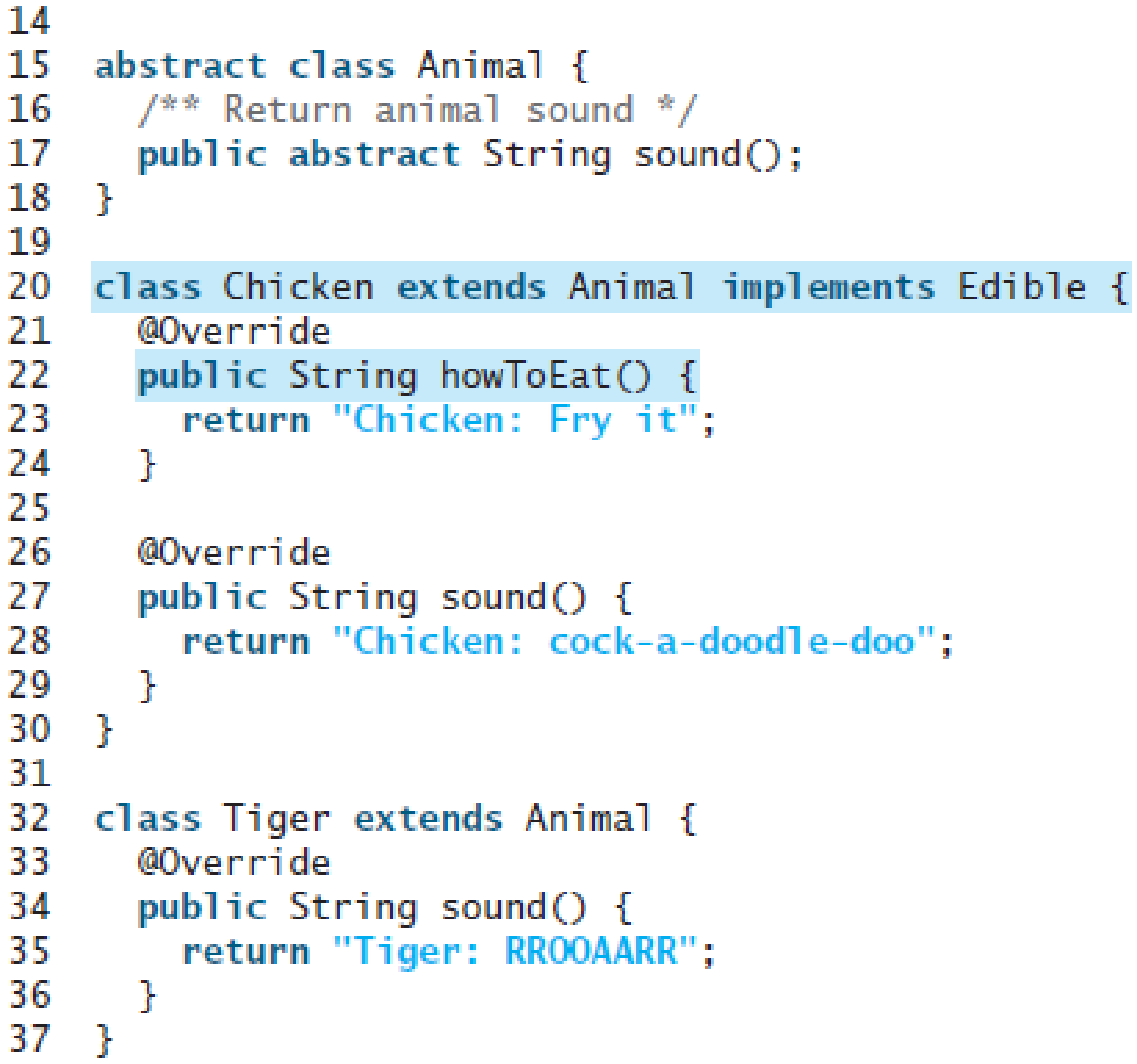
Implementing an Interface (Example)



LISTING 13.7 TestEdible.java

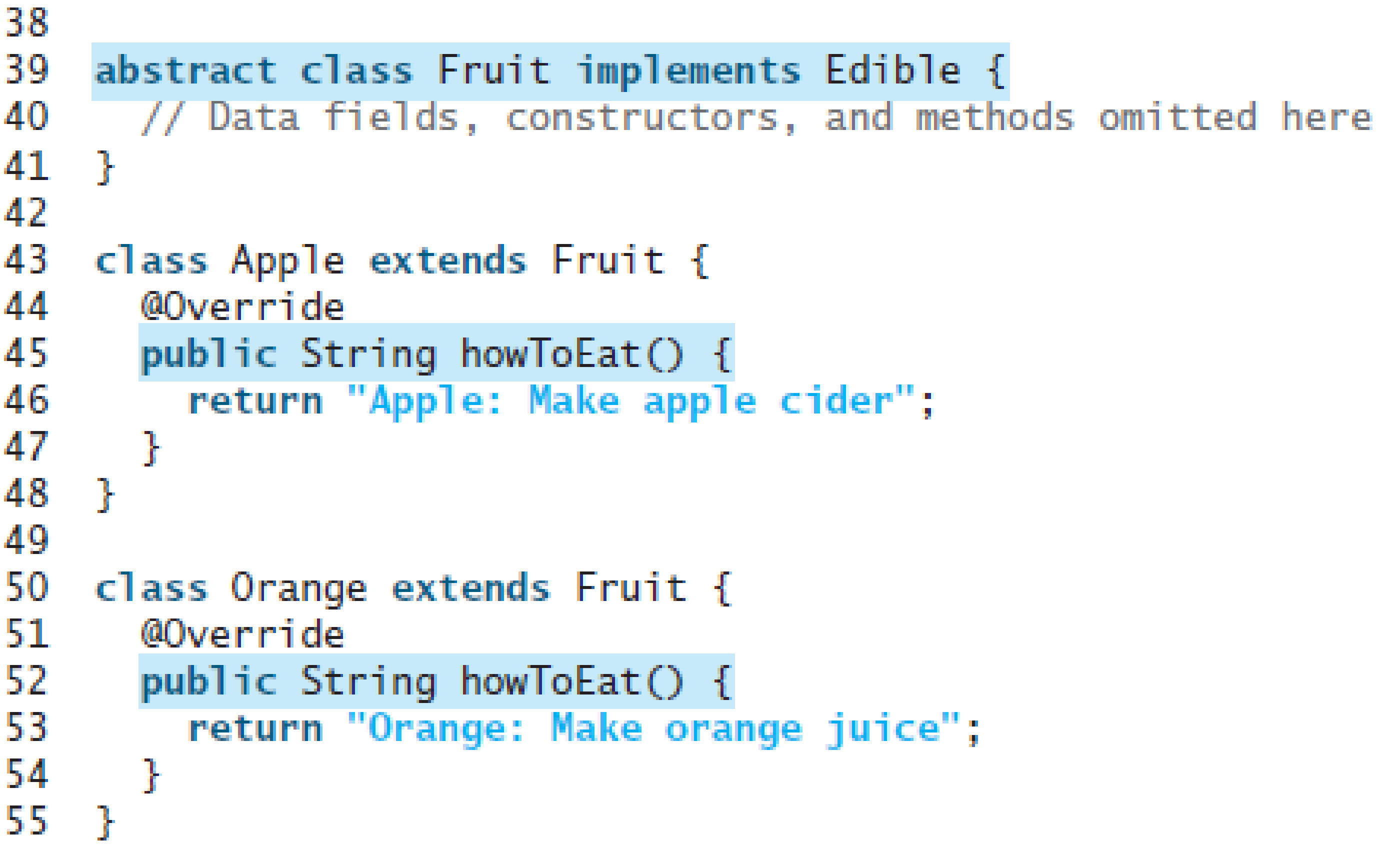
```java
public class TestEdible {
  public static void main(String[] args) {
    Object[] objects = {new Tiger(), new Chicken(), new Apple()};
    for (int i = 0; i < objects.length; i++) {
      if (objects[i] instanceof Edible)
        System.out.println(((Edible)objects[i]).howToEat());

      if (objects[i] instanceof Animal) {
        System.out.println(((Animal)objects[i]).sound());
      }
    }
  }
}
```

Implementing an Interface (Example)



```
abstract class Animal {
  /** Return animal sound */
  public abstract String sound();
}

class Chicken extends Animal implements Edible {
  @Override
  public String howToEat() {
    return "Chicken: Fry it";
  }

  @Override
  public String sound() {
    return "Chicken: cock-a-doodle-doo";
  }
}

class Tiger extends Animal {
  @Override
  public String sound() {
    return "Tiger: RROOAARR";
  }
}
```

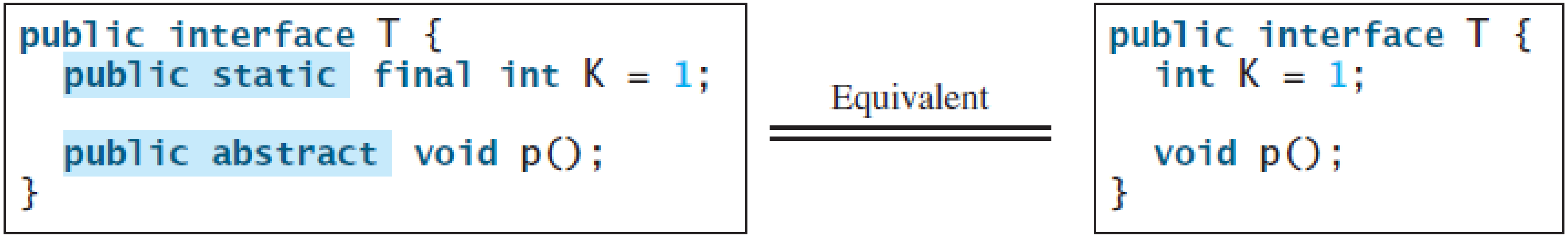
Implementing an Interface (Example)



```java

abstract class Fruit implements Edible {
  // Data fields, constructors, and methods omitted here
}

class Apple extends Fruit {
  @Override
  public String howToEat() {
    return "Apple: Make apple cider";
  }
}

class Orange extends Fruit {
  @Override
  public String howToEat() {
    return "Orange: Make orange juice";
  }
}
```

A Note Regarding Interfaces

- All data fields of an interface are *public static final*.
- All methods of an interface are *public abstract*.
- Therefore, Java allows these modifiers to be omitted as follows:



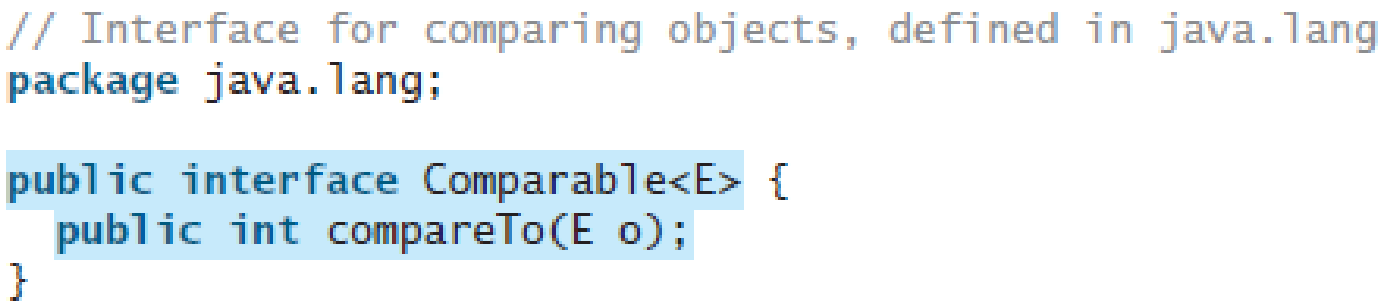
```
public interface T {
  public static final int K = 1;

  public abstract void p();
}
```

Equivalent

```
public interface T {
  int K = 1;

  void p();
}
```

The Comparable Interface

- Suppose you want to design a generic method to find the larger of two objects of the same type, such as two students, two dates, two circles, two rectangles, or two squares.
- In order to accomplish this, the two objects must be comparable, so the common behavior for the objects must be comparable.
- Java provides the *Comparable* interface for this purpose.
- The interface is defined as follows:

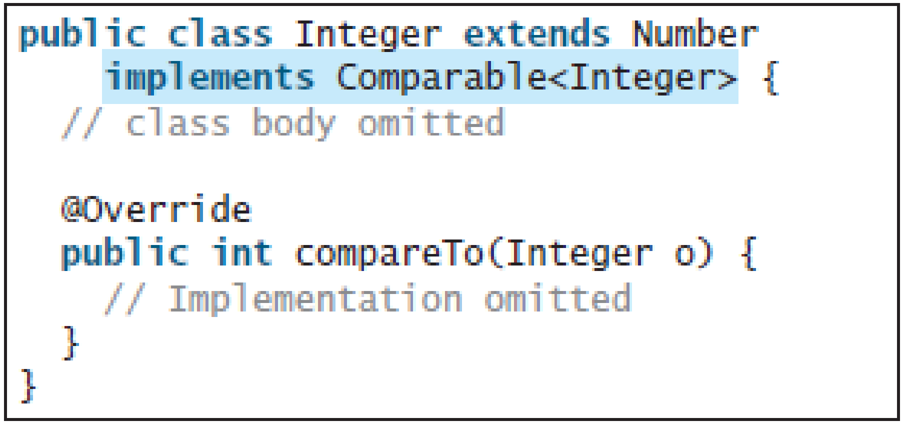


```
// Interface for comparing objects, defined in java.lang
package java.lang;

public interface Comparable<E> {
  public int compareTo(E o);
}
```

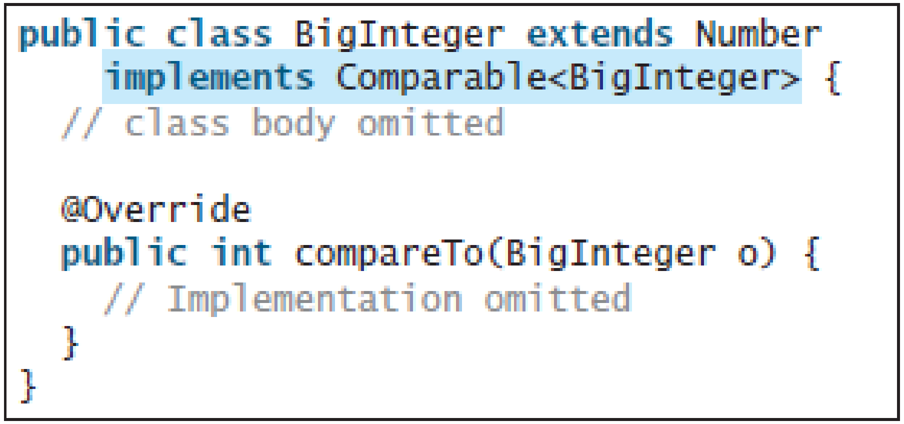
The Comparable Interface (Cont.)

- The *compareTo* method determines the order of this object with the specified object *o* and returns a negative integer, zero, or a positive integer if this object is less than, equal to, or greater than *o*.
- The *Comparable* interface is a generic interface.
  - The generic type *E* is replaced by a concrete type when implementing this interface.
- Many classes in the Java library implement *Comparable* to define a natural order for objects.
  - The classes *Byte, Short, Integer, Long, Float, Double, Character, BigInteger, BigDecimal, Calendar, String,* and *Date* all implement the *Comparable* interface.

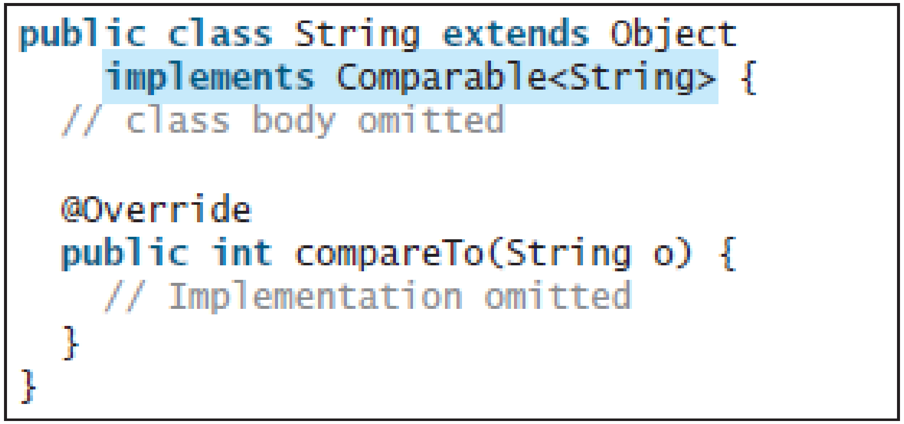
The Comparable Interface (Example)



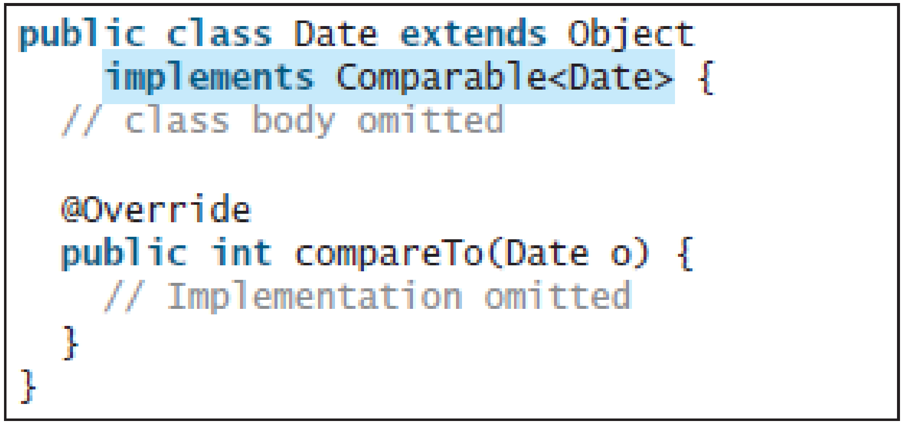
```
public class Integer extends Number
    implements Comparable<Integer> {
  // class body omitted

  @Override
  public int compareTo(Integer o) {
    // Implementation omitted
  }
}
```



```
public class BigInteger extends Number
    implements Comparable<BigInteger> {
  // class body omitted

  @Override
  public int compareTo(BigInteger o) {
    // Implementation omitted
  }
}
```



```
public class String extends Object
    implements Comparable<String> {
  // class body omitted

  @Override
  public int compareTo(String o) {
    // Implementation omitted
  }
}
```



```
public class Date extends Object
    implements Comparable<Date> {
  // class body omitted

  @Override
  public int compareTo(Date o) {
    // Implementation omitted
  }
}
```

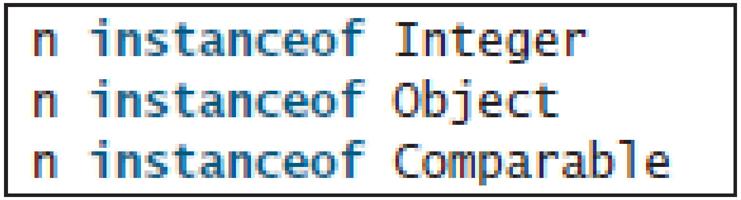
The Comparable Interface (Cont.)

- Thus, numbers are *comparable*, strings are *comparable*, and so are dates.
- You can use the *compareTo* method to compare two numbers, two strings, and two dates.
- Example:

```
System.out.println(new Integer(3).compareTo(new
Integer(5)));
System.out.println("ABC".compareTo("ABE"));
java.util.Date date1 = new java.util.Date(2013, 1, 1);
java.util.Date date2 = new java.util.Date(2012, 1, 1);
System.out.println(date1.compareTo(date2));
```

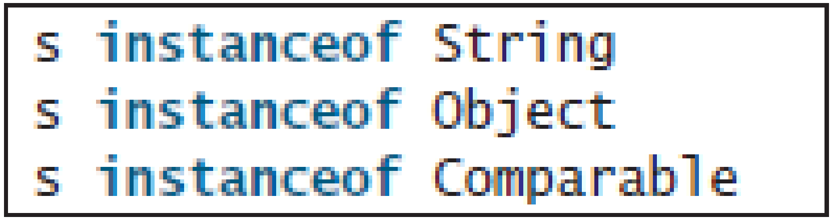
The Comparable Interface (Cont.)

- Let *n* be an *Integer* object:



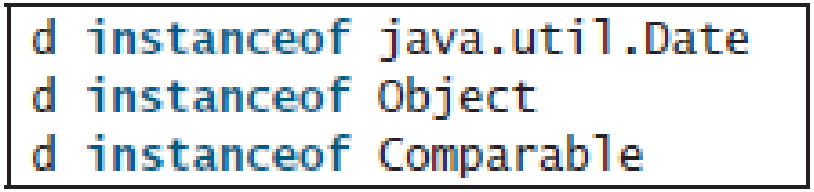
```
n instanceof Integer
n instanceof Object
n instanceof Comparable
```

- Let *s* be a *String* object:



```
s instanceof String
s instanceof Object
s instanceof Comparable
```

- Let *d* be a *Date* object:

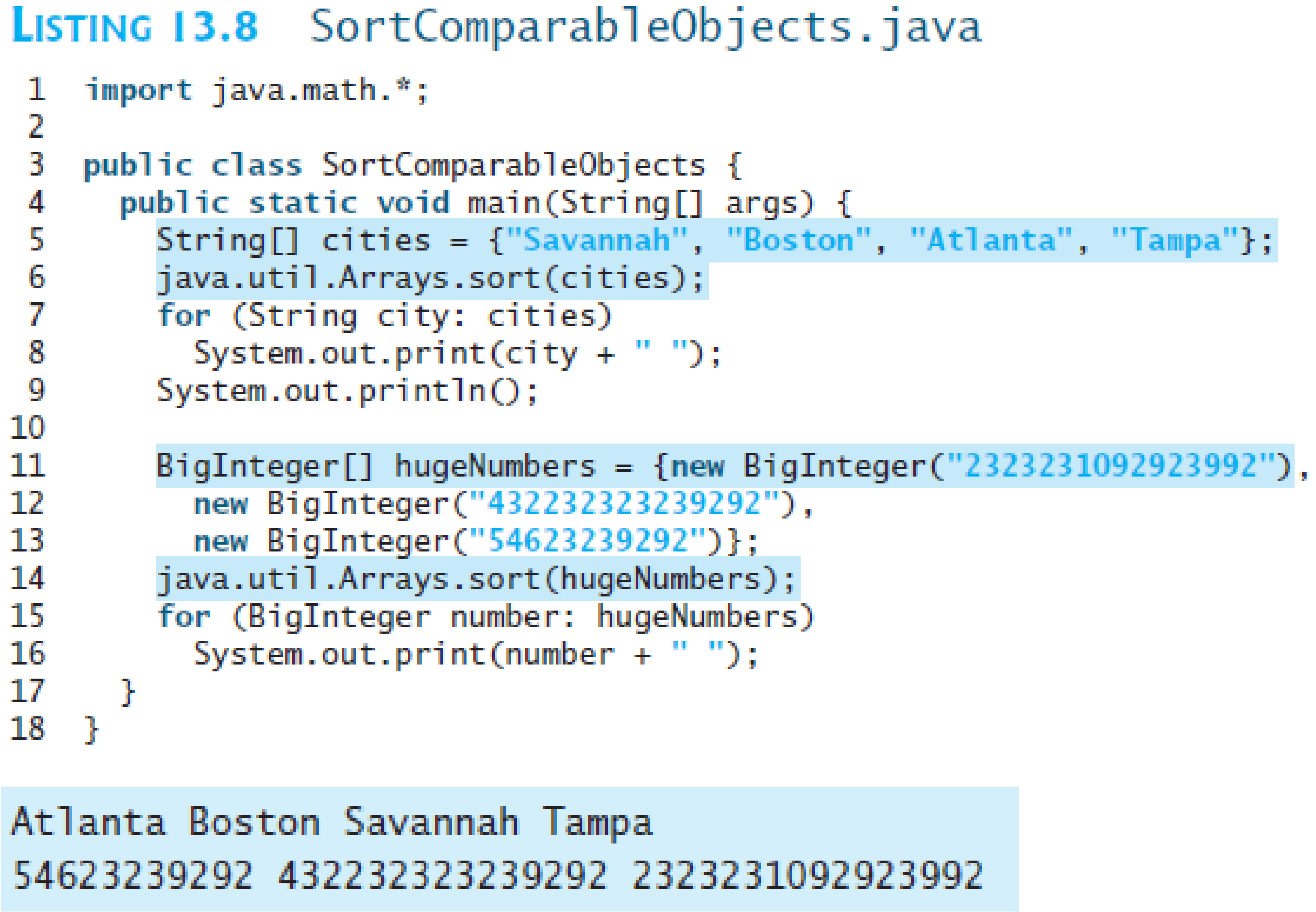


```
d instanceof java.util.Date
d instanceof Object
d instanceof Comparable
```

The Comparable Interface (Cont.)

- Since all *Comparable* objects have the *compareTo* method:
  - The *java.util.Arrays.sort(Object[])* method in the Java API uses the *compareTo* method to compare and sorts the objects in an array.
  - Provided that the objects are instances of the *Comparable* interface.

The Comparable Interface (Example)

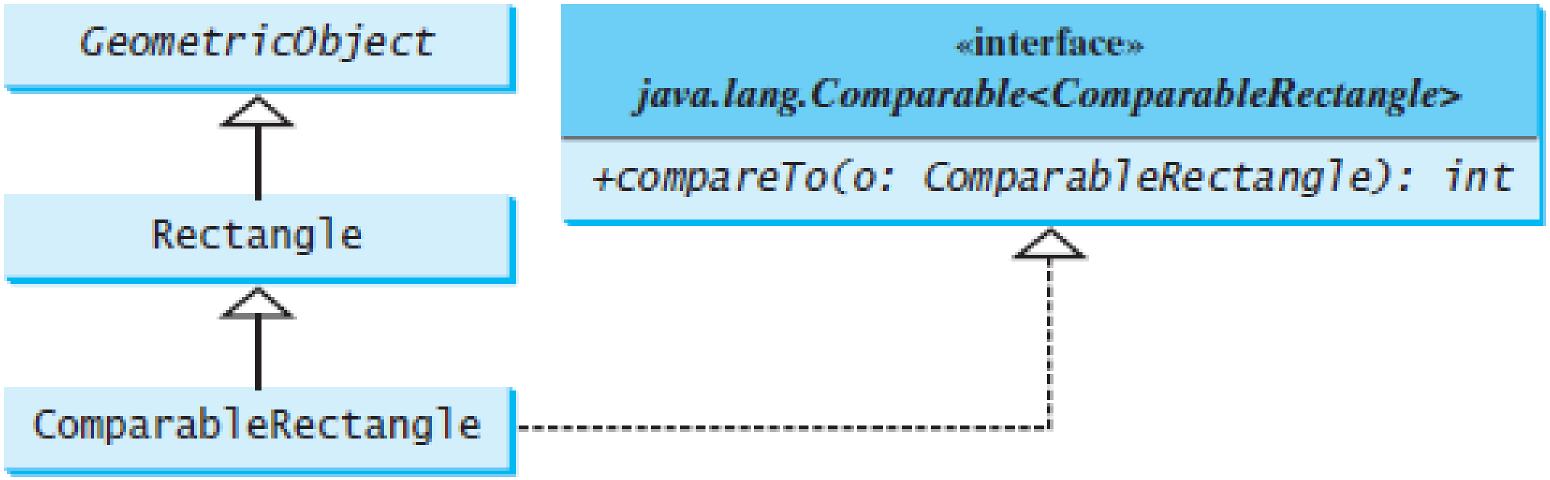


LISTING 13.8 SortComparableObjects.java

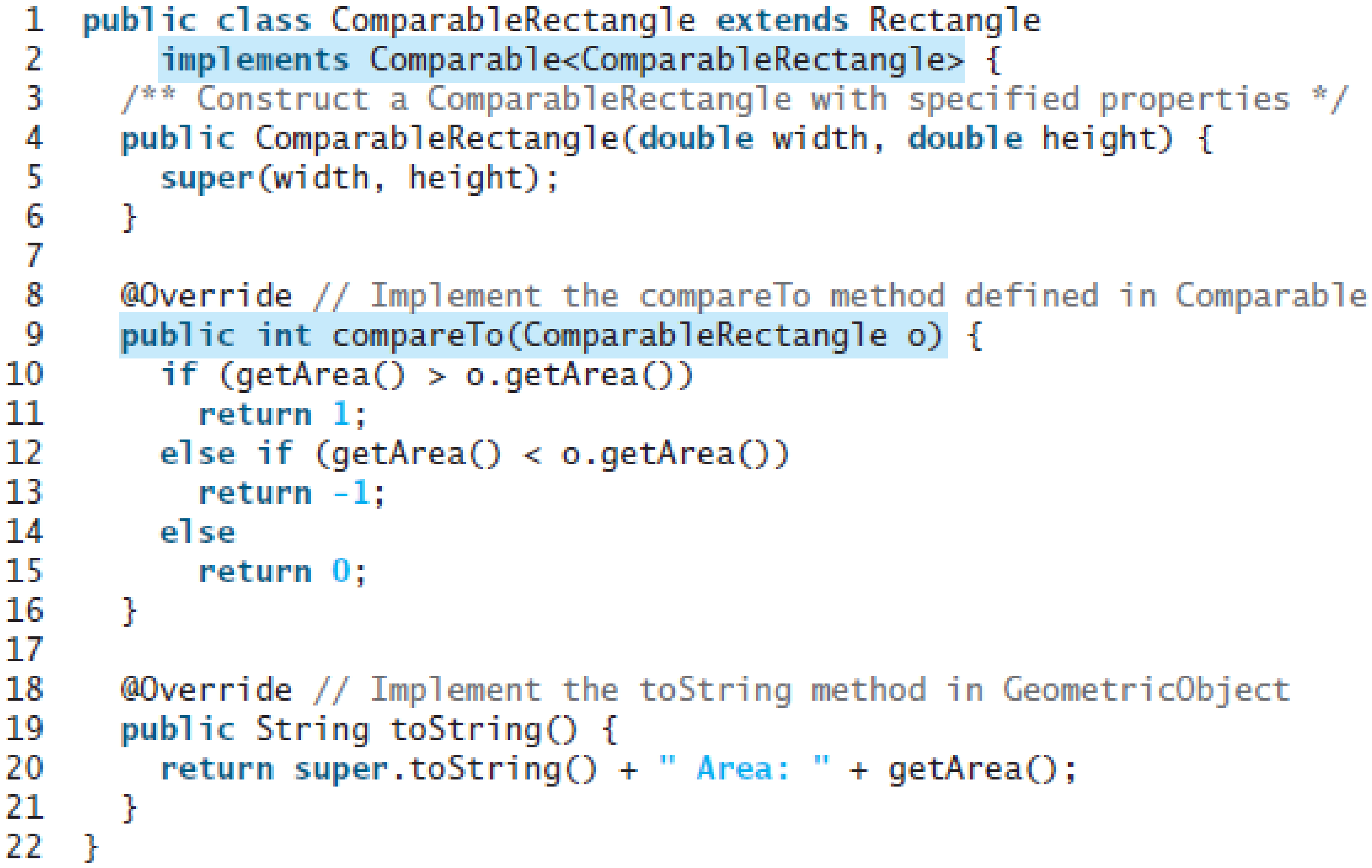
```java
import java.math.*;

public class SortComparableObjects {
  public static void main(String[] args) {
    String[] cities = {"Savannah", "Boston", "Atlanta", "Tampa"};
    java.util.Arrays.sort(cities);
    for (String city: cities)
      System.out.print(city + " ");
    System.out.println();

    BigInteger[] hugeNumbers = {new BigInteger("2323231092923992"),
      new BigInteger("432232323239292"),
      new BigInteger("54623239292")};
    java.util.Arrays.sort(hugeNumbers);
    for (BigInteger number: hugeNumbers)
      System.out.print(number + " ");
  }
}
```

```
Atlanta Boston Savannah Tampa
54623239292 432232323239292 2323231092923992
```

The Comparable Interface Comparable Rectangle (UML Diagram)

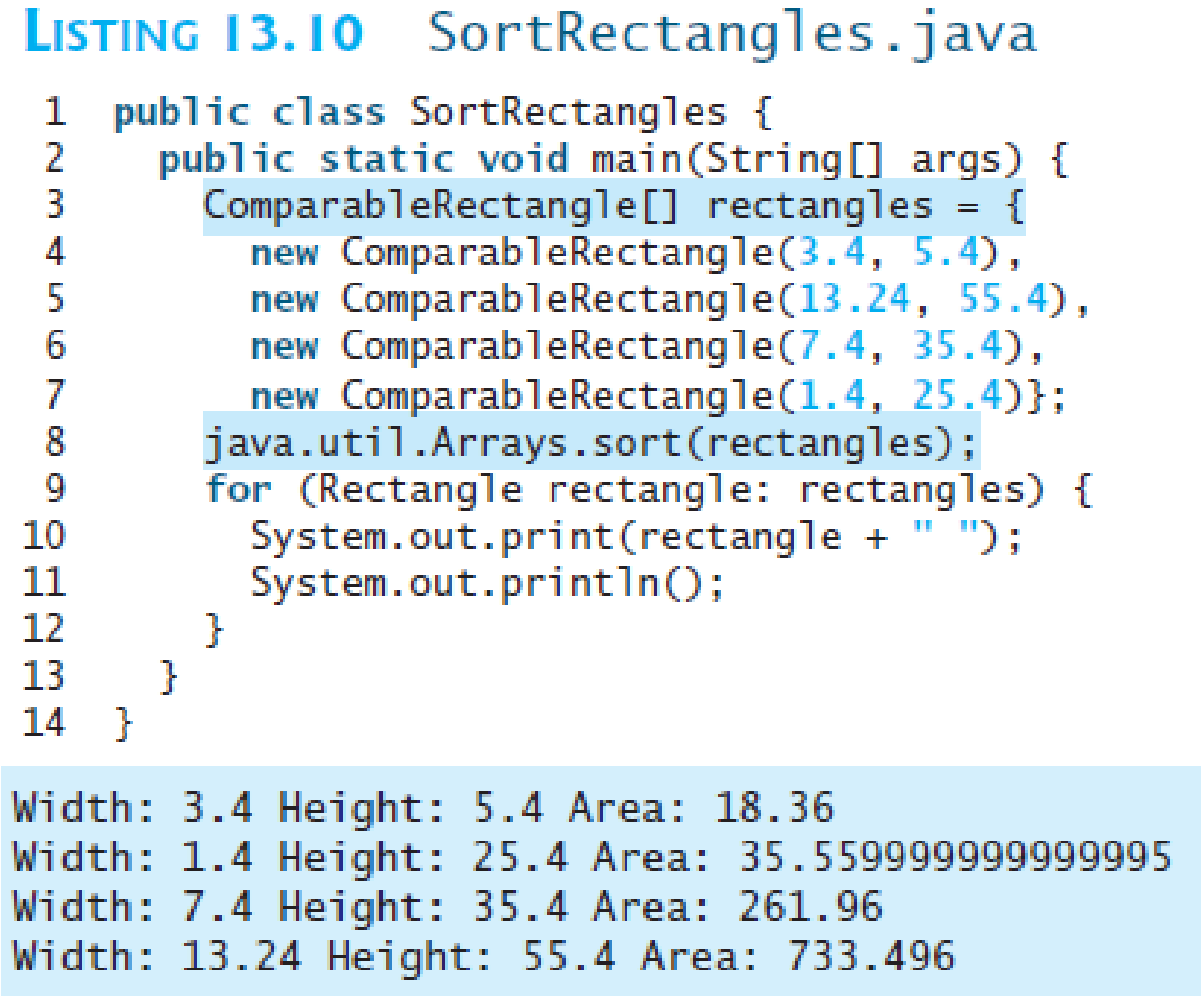

GeometricObject
Rectangle
ComparableRectangle
«interface»
java.lang.Comparable<ComparableRectangle>
+compareTo(o: ComparableRectangle): int

The Comparable Interface
Comparable Rectangle (Code)



```java
public class ComparableRectangle extends Rectangle
    implements Comparable<ComparableRectangle> {
  /** Construct a ComparableRectangle with specified properties */
  public ComparableRectangle(double width, double height) {
    super(width, height);
  }

  @Override // Implement the compareTo method defined in Comparable
  public int compareTo(ComparableRectangle o) {
    if (getArea() > o.getArea())
      return 1;
    else if (getArea() < o.getArea())
      return -1;
    else
      return 0;
  }

  @Override // Implement the toString method in GeometricObject
  public String toString() {
    return super.toString() + " Area: " + getArea();
  }
}
```

The Comparable Interface SortRectangles.java



LISTING 13.10 SortRectangles.java

```java
public class SortRectangles {
  public static void main(String[] args) {
    ComparableRectangle[] rectangles = {
      new ComparableRectangle(3.4, 5.4),
      new ComparableRectangle(13.24, 55.4),
      new ComparableRectangle(7.4, 35.4),
      new ComparableRectangle(1.4, 25.4)};
    java.util.Arrays.sort(rectangles);
    for (Rectangle rectangle: rectangles) {
      System.out.print(rectangle + " ");
      System.out.println();
    }
  }
}
```

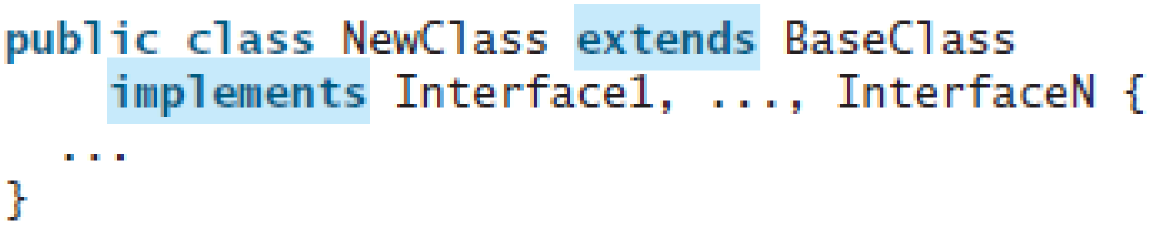
```
Width: 3.4 Height: 5.4 Area: 18.36
Width: 1.4 Height: 25.4 Area: 35.559999999999995
Width: 7.4 Height: 35.4 Area: 261.96
Width: 13.24 Height: 55.4 Area: 733.496
```

Interfaces vs. Abstract Classes

	Variables	**Constructors**	**Methods**
Abstract Class	No restrictions.	Constructors are invoked by subclasses through constructor chaining. An abstract class cannot be instantiated using the new operator	No restrictions.
Interface	All variables must be public static final.	No constructors. An interface cannot be instantiated using the new operator.	All methods must be public abstract instance methods.

More on Interfaces

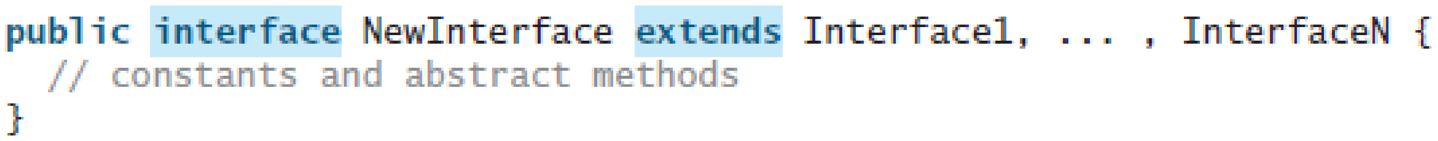
- Java allows only *single inheritance* for class extension, but allow *multiple inheritance* for interface extension.
- For example:



```
public class NewClass extends BaseClass
    implements Interface1, ..., InterfaceN {
  ...
}
```

More on Interfaces (Cont.)

- An interface can inherit other interfaces using the *extends* keyword.
  - Such an interface is called a sub-interface.
  - An interface can extend other interfaces but not classes.
- For example, *NewInterface* in the following code is a sub-interface of *interface1*, …., and *interfaceN*:



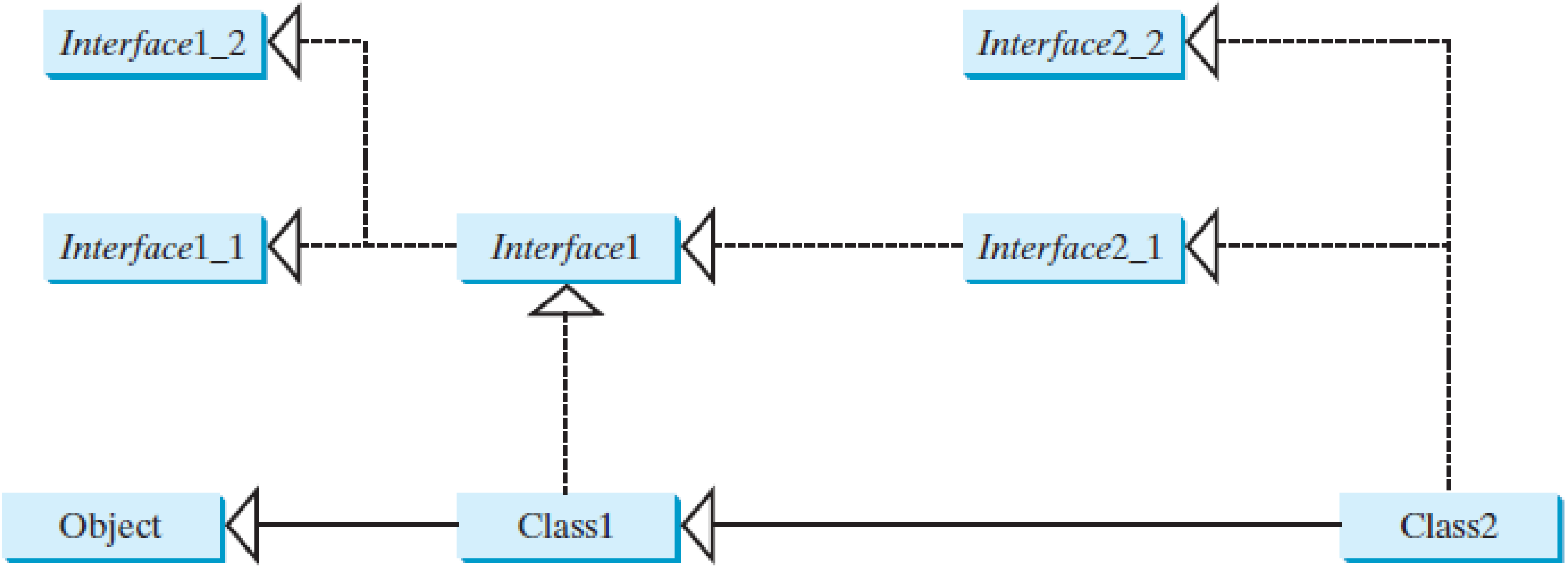
```
public interface NewInterface extends Interface1, ... , InterfaceN {
  // constants and abstract methods
}
```

  - A class implementing NewInterface must implement the abstract methods defined in NewInterface, Interface1, …., and InterfaceN.

More on Interfaces (Cont.)

- All classes share a single root, the *object* class, but there is no single root for interfaces.
- A variable of an interface type can reference any instance of the class that implements the interface.
  - If a class implements an interface, the interface is like a superclass for the class.

More on Interfaces (Cont.)



- Suppose c is an instance of class2. c is also an instance of Object, Class1, Interface1, Interface1_1, Interface1_2, Interface2_1, and Interface2_2.

Design Guide: Interface or Abstract Class

- Abstract classes and interfaces can both be used to specify common behavior of objects.
- How to decide whether to use an interface or an abstract class?
  - In general, a strong is-a relationship that clearly describes a parent-child relationship should be modeled using classes.
    - For example, since an orange is a fruit, their relationship should be modeled using class inheritance.
- A weak is-a relationship, also known as an is-kind-of relationship, indicates that an object possesses a certain property.
  - A weak is-a relationship can be modeled using interfaces.
  - When it is desired to define a common supertype for unrelated classes, an interface should be used.

Class Design Guidelines
Cohesion

- A class should describe a single entity, and all the class operations should logically fit together to support a coherent purpose.
- You can use a class for students, for example, but you should not combine students and staff in the same class, because students and staff are different entities.
- A single entity with many responsibilities can be broken into several classes to separate the responsibilities.
- The classes *String*, *StringBuilder*, and *StringBuffer* all deal with strings, for example, but have different responsibilities.
  - The *String* class deals with immutable strings.
  - The *StringBuilder* class is for creating mutable strings.
  - The *StringBuffer* class is similar to *StringBuilder* except that *StringBuffer* contains synchronized methods for updating strings.

Class Design Guidelines Consistency

- Follow standard Java programming style and naming conventions.
- Choose informative names for classes, data fields, and methods.
- A popular style is to place the data declaration before the constructor and place constructors before methods.
- Make the names consistent.
  - It is not a good practice to choose different names for similar operations. For example, the *length()* method returns the size of a *String*, a StringBuilder, and a *StringBuffer*. It would be inconsistent if different names were used for this method in these classes.
- In general, you should consistently provide a public no-arg constructor for constructing a default instance.
  - If a class does not support a no-arg constructor, document the reason.
  - If no constructors are defined explicitly, a public default no-arg constructor with an empty body is assumed.
- If you want to prevent users from creating an object for a class, you can declare a private constructor in the class, as is the case for the *Math* class.

Class Design Guidelines Encapsulation

- A class should use the ***private*** modifier to hide its data from direct access by clients.
- This makes the class easy to maintain.
- Provide a getter method only if you want the data field to be readable.
- Provide a setter method only if you want the data field to be updateable.

Object-Oriented Problem Solving

Exception Handling

Based on Chapter 12 of "Introduction to Java Programming" by Y. Daniel Liang.

Eng. Asma Abdel Karim
Computer Engineering Department

Outline

- Introduction (12.1)
- Exception Handling Overview (12.2)
- Exception Types (12.3)
- More on Exception Handling (12.4)
  - Declaring Exceptions (12.4.1)
  - Throwing Exceptions (12.4.2)
  - Catching Exceptions (12.4.3)
  - Getting Information from Exceptions (12.4.4)
  - Example: Declaring, Throwing, and Catching Exceptions (12.4.5)
- The *finally* Clause (12.5)
- When to use Exceptions? (12.6)
- Rethrowing Exceptions (12.7)
- Chained Exceptions (12.8)
- Defining Custom Exception Classes (12.9)

Introduction

- Runtime errors occur while a program is running if the JVM detects an operation that is impossible to carry out.
  - If you access an array using an index that is out of bounds, you will get a runtime error with an *ArrayIndexOutOfBoundsException*.
  - If you enter a *double* value when your program expects an *integer*, you will get a runtime error with an *InputMismatchException*.
- In Java, runtime errors are thrown as *exceptions*.
- An *exception* is an object that represents an error or a condition that prevents execution from proceeding normally.
- If the *exception* is not handled, the program will terminate abnormally.
  - *Exception handling* enables a program to deal with exceptional situations and continue its normal execution.

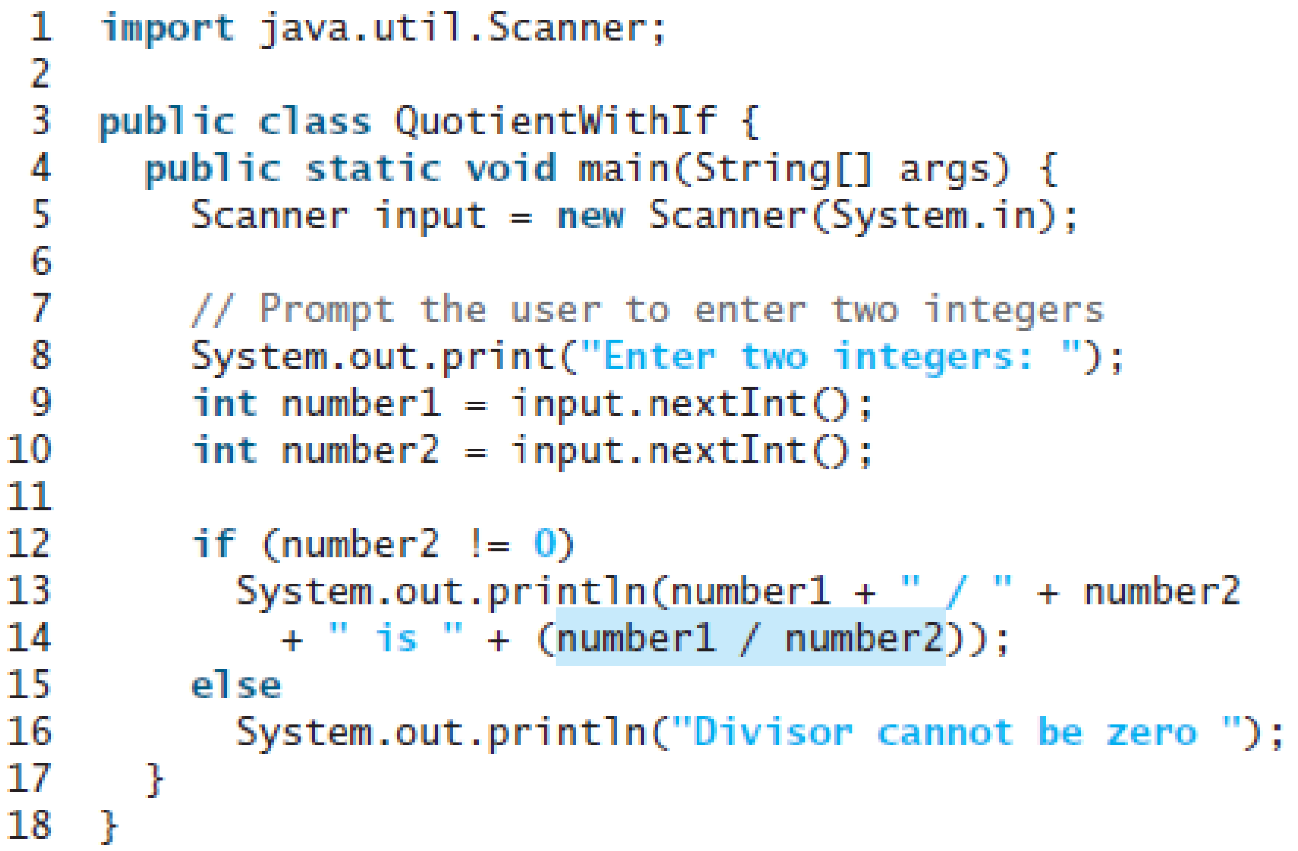
Exception Handling Overview
Quotient.java

```java
import java.util.Scanner;

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

    // Prompt the user to enter two integers
    System.out.print("Enter two integers: ");
    int number1 = input.nextInt();
    int number2 = input.nextInt();

    System.out.println(number1 + " / " + number2 + " is " +
      (number1 / number2));
  }
}
```

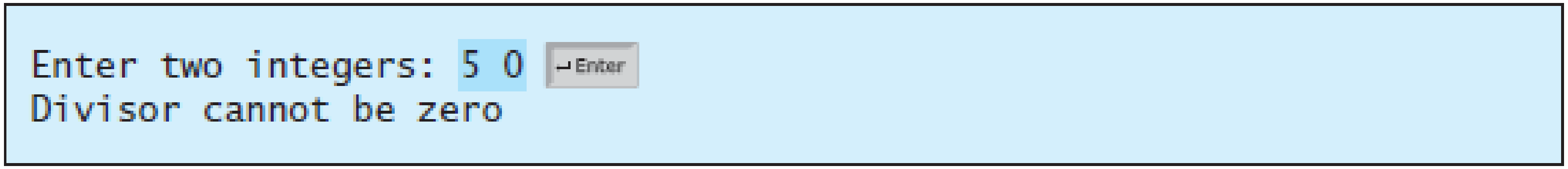
```
Enter two integers: 5 2 ↵Enter
5 / 2 is 2
```

```
Enter two integers: 3 0 ↵Enter
Exception in thread "main" java.lang.ArithmeticException: / by zero
at Quotient.main(Quotient.java:11)
```

Exception Handling Overview (Cont.) QuotientWithIf.java

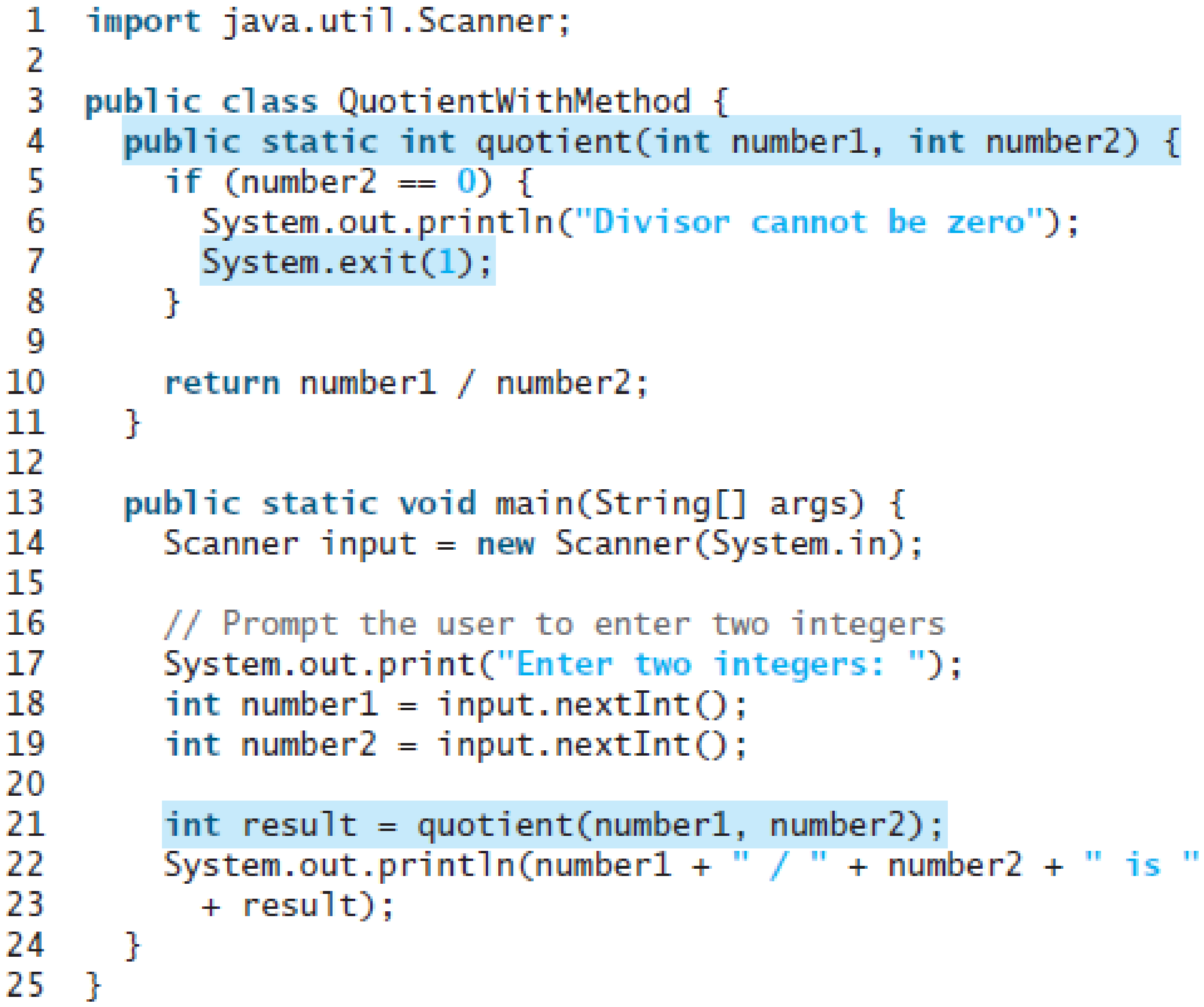


```java
import java.util.Scanner;

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

    // Prompt the user to enter two integers
    System.out.print("Enter two integers: ");
    int number1 = input.nextInt();
    int number2 = input.nextInt();

    if (number2 != 0)
      System.out.println(number1 + " / " + number2
        + " is " + (number1 / number2));
    else
      System.out.println("Divisor cannot be zero ");
  }
}
```



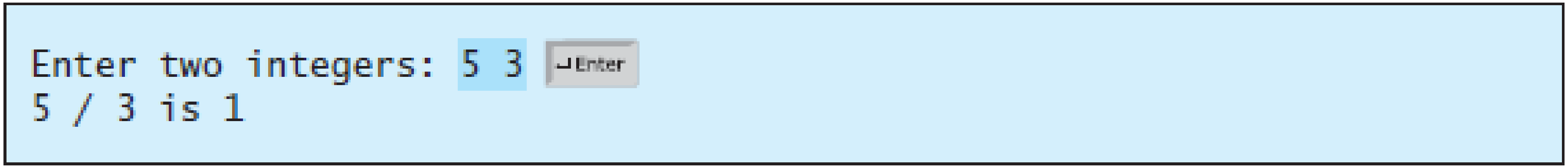
```
Enter two integers: 5 0 ↵Enter
Divisor cannot be zero
```

Exception Handling Overview (Cont.)
QuotientWithMethod.java

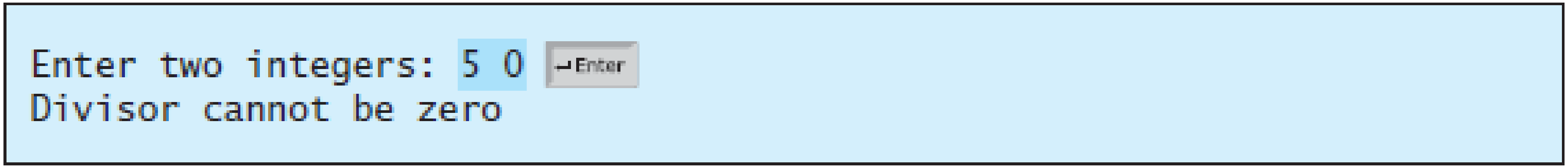


```java
import java.util.Scanner;

public class QuotientWithMethod {
  public static int quotient(int number1, int number2) {
    if (number2 == 0) {
      System.out.println("Divisor cannot be zero");
      System.exit(1);
    }

    return number1 / number2;
  }

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

    // Prompt the user to enter two integers
    System.out.print("Enter two integers: ");
    int number1 = input.nextInt();
    int number2 = input.nextInt();

    int result = quotient(number1, number2);
    System.out.println(number1 + " / " + number2 + " is "
      + result);
  }
}
```

Exception Handling Overview (Cont.)
QuotientWithMethod.java (Output)



```
Enter two integers: 5 3 ↵Enter
5 / 3 is 1
```

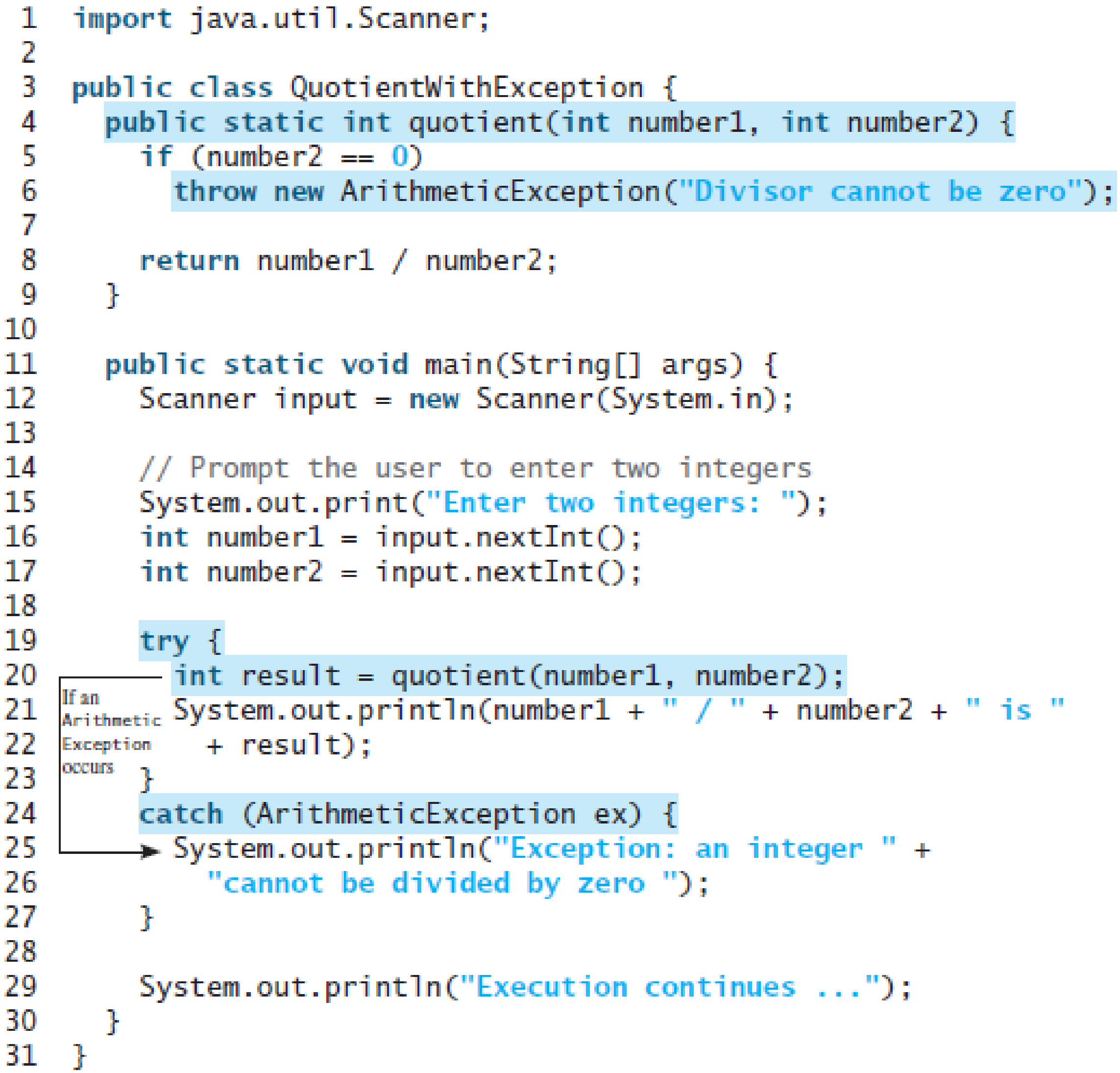


```
Enter two integers: 5 0 ↵Enter
Divisor cannot be zero
```

Program is terminated if number2 equals 0.

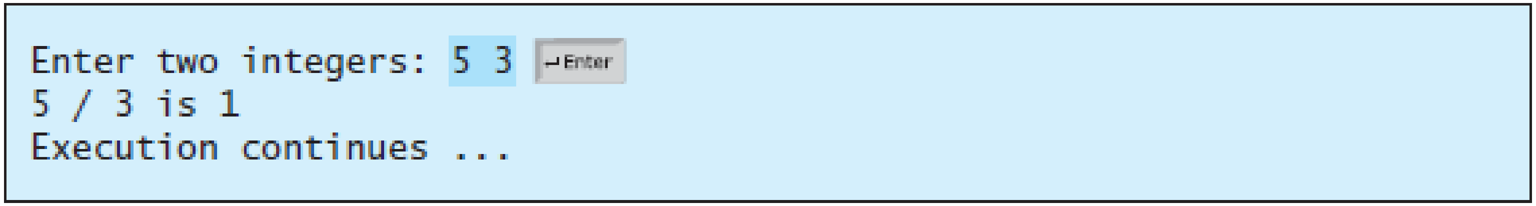
Problem: what if the caller should decide whether to terminate the program!

Exception Handling Overview
QuotientWithException.java

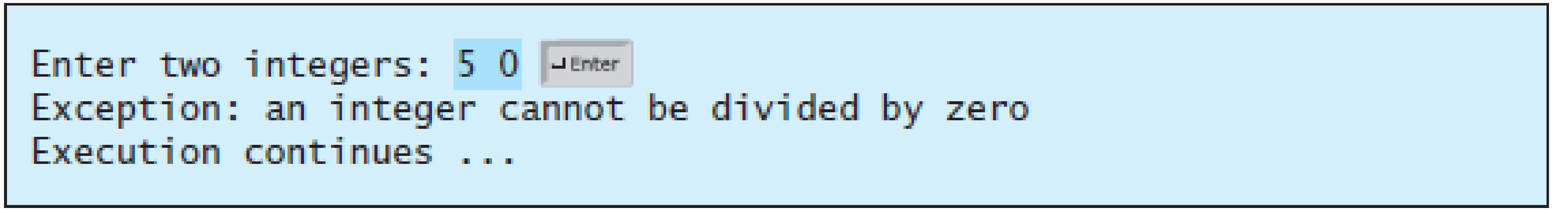


```java
import java.util.Scanner;

public class QuotientWithException {
  public static int quotient(int number1, int number2) {
    if (number2 == 0)
      throw new ArithmeticException("Divisor cannot be zero");

    return number1 / number2;
  }

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

    // Prompt the user to enter two integers
    System.out.print("Enter two integers: ");
    int number1 = input.nextInt();
    int number2 = input.nextInt();

    try {
      int result = quotient(number1, number2);
      System.out.println(number1 + " / " + number2 + " is "
        + result);
    }
    catch (ArithmeticException ex) {
      System.out.println("Exception: an integer " +
        "cannot be divided by zero ");
    }

    System.out.println("Execution continues ...");
  }
}
```

Exception Handling Overview QuotientWithException.java (Output)

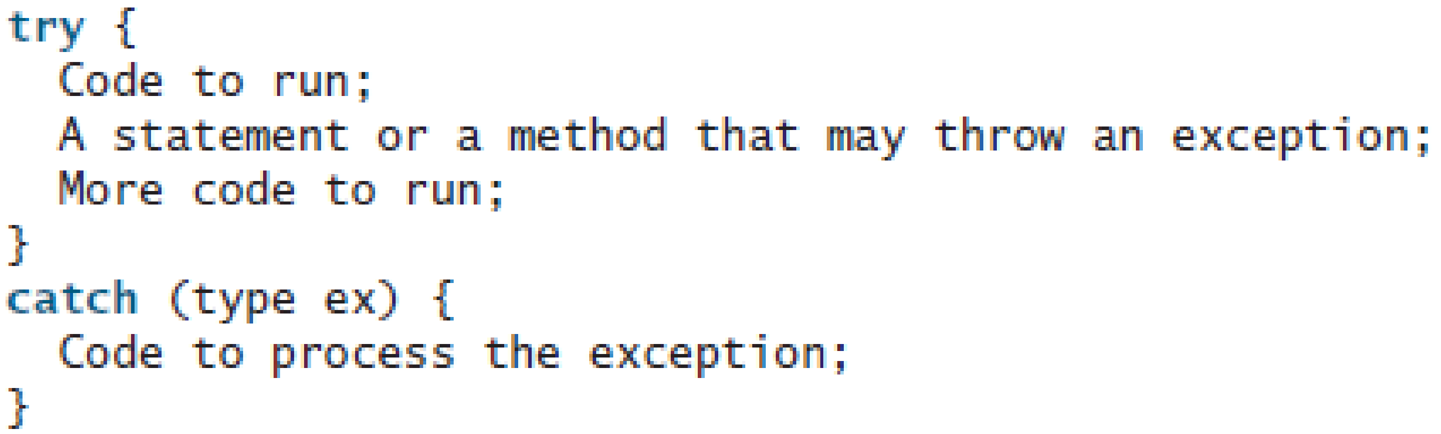


```
Enter two integers: 5 3 ↵Enter
5 / 3 is 1
Execution continues ...
```



```
Enter two integers: 5 0 ↵Enter
Exception: an integer cannot be divided by zero
Execution continues ...
```

Exception Handling Overview



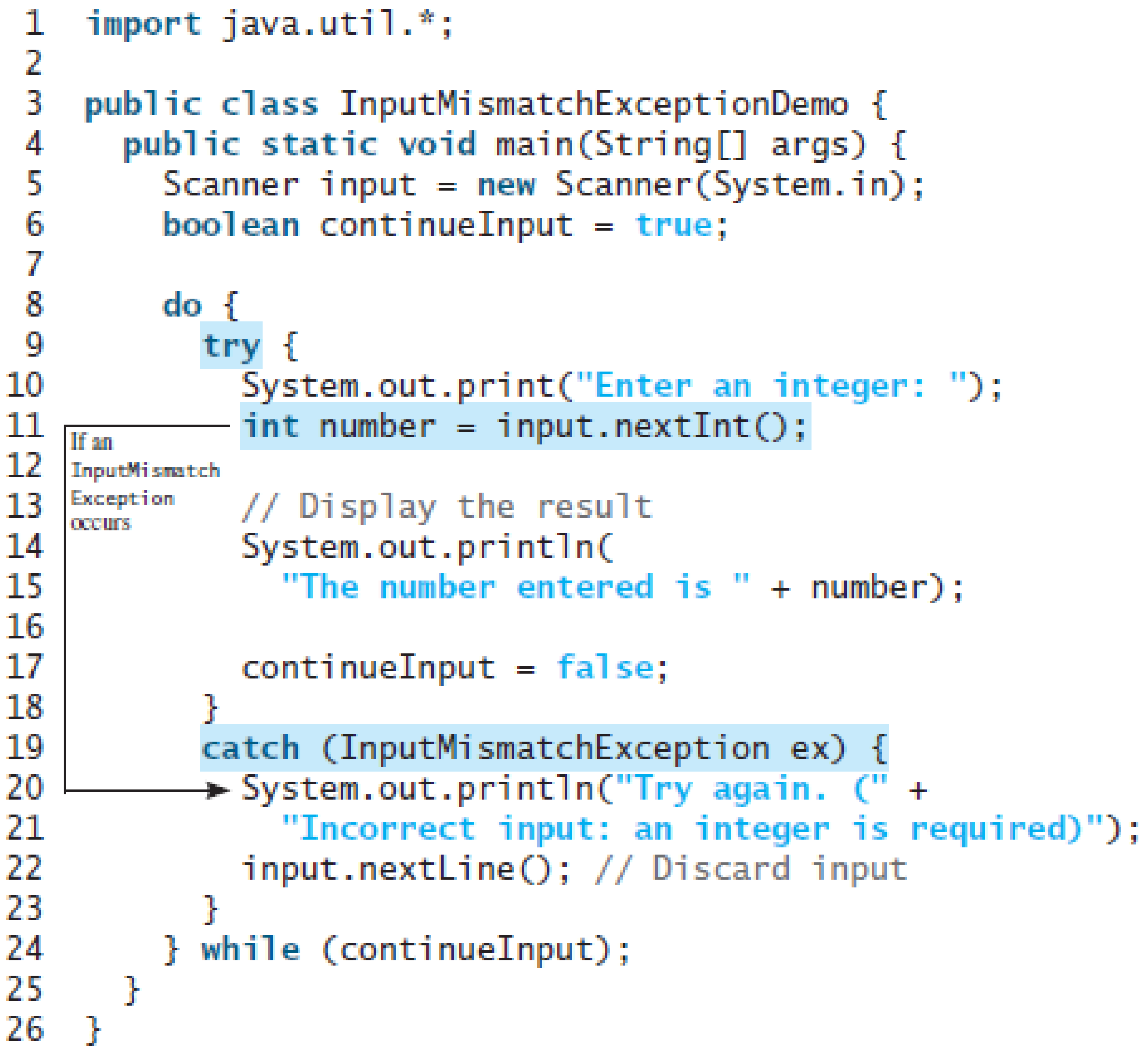
```
try {
  Code to run;
  A statement or a method that may throw an exception;
  More code to run;
}
catch (type ex) {
  Code to process the exception;
}
```

Exception Handling Overview

Benefit of Exception Handling

- The key benefit of exception handling is separating the detection of an error (done in a called method) from the handling of an error (done in the calling method).
  - Often the called method does not know what to do in case of error.
  - This is typically the case for the library methods.
    - The library method can detect the error, but only the caller knows what needs to be done when an error occurs.

Exception Handling Overview
InputMismatchExceptionDemo.java

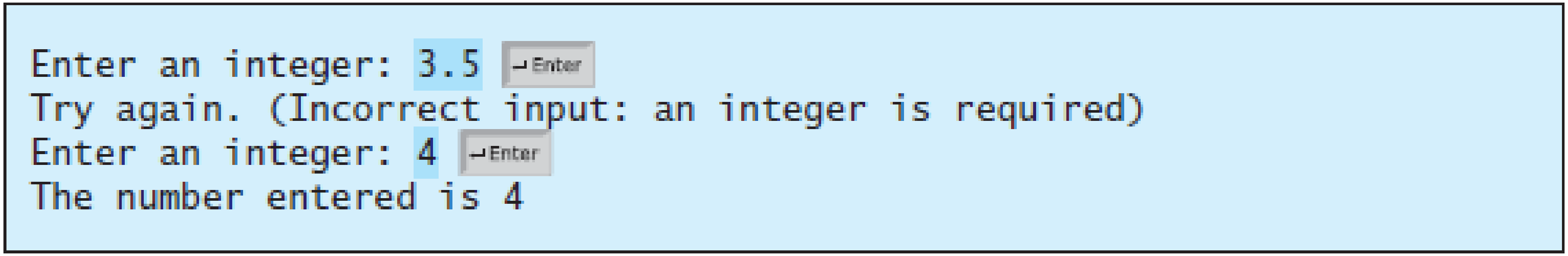


```
import java.util.*;

public class InputMismatchExceptionDemo {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    boolean continueInput = true;

    do {
      try {
        System.out.print("Enter an integer: ");
        int number = input.nextInt();

        // Display the result
        System.out.println(
          "The number entered is " + number);

        continueInput = false;
      }
      catch (InputMismatchException ex) {
        System.out.println("Try again. (" +
          "Incorrect input: an integer is required)");
        input.nextLine(); // Discard input
      }
    } while (continueInput);
  }
}
```

If an InputMismatch Exception occurs

Exception Handling Overview

InputMismatchExceptionDemo.java (Output)

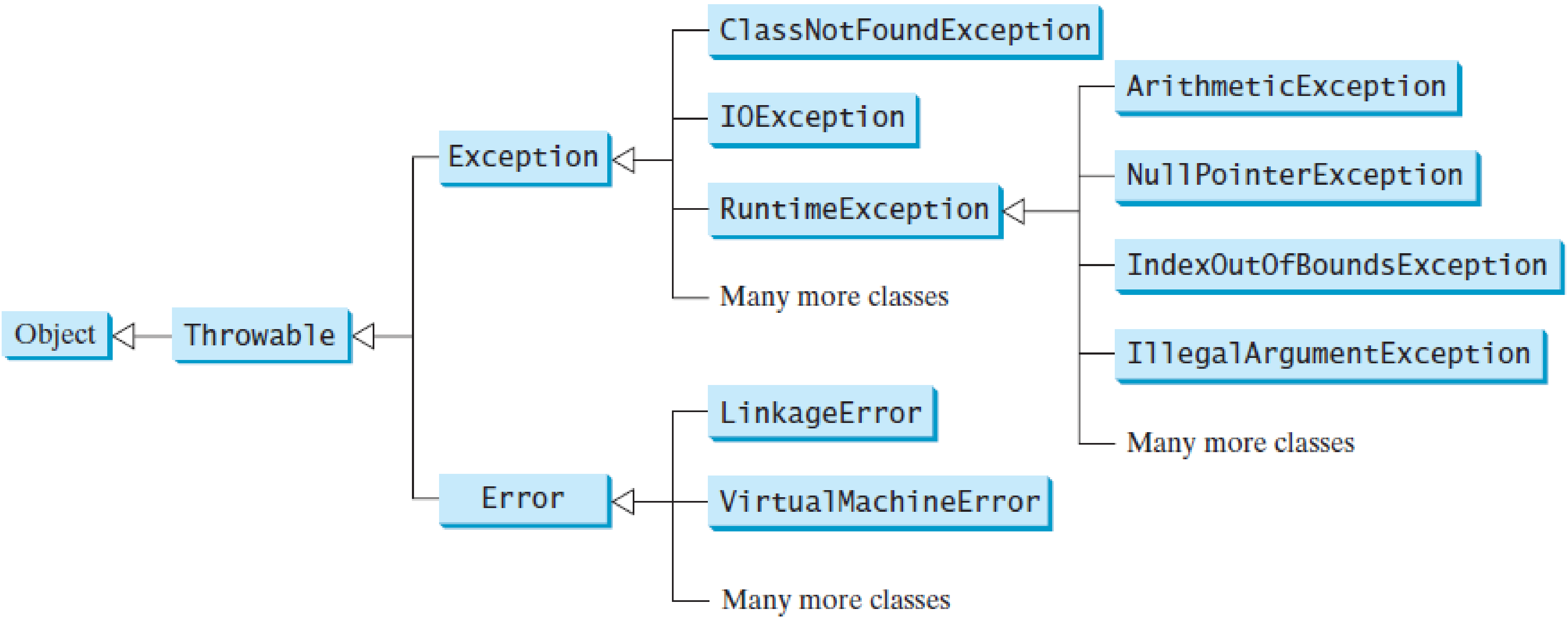


```
Enter an integer: 3.5 ↵Enter
Try again. (Incorrect input: an integer is required)
Enter an integer: 4 ↵Enter
The number entered is 4
```

Exception Types

- Exceptions are objects, and objects are defined using classes.
- The root class for all exceptions is *java.lang.Throwable*.
- There are many predefined exception classes in the Java API.
- You can also define your own exception classes.

Exception Types (Cont.)

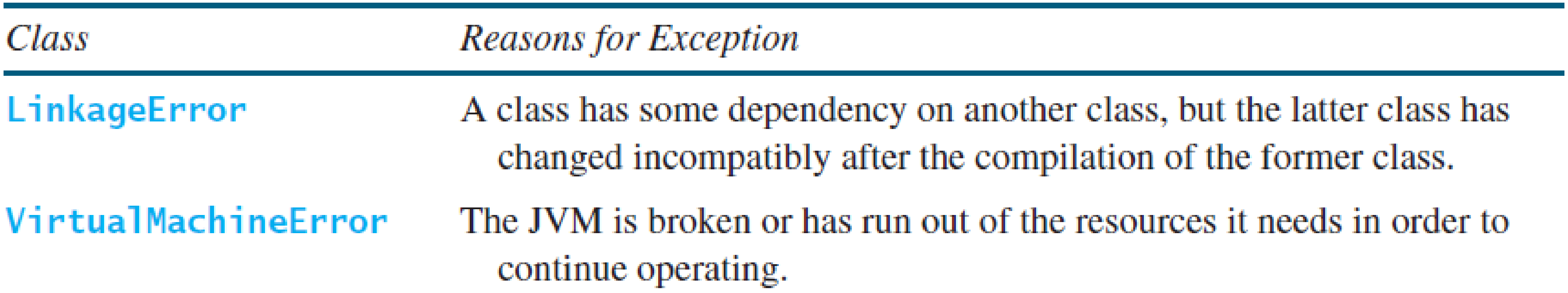


Exception Types (Cont.)

- The *Throwable* class is the root of all exception classes.
  - All Java exception classes inherit directly or indirectly from *Throwable*.
  - You can create your own exception classes by extending *Exception* or a subclass of *Exception*.
- The exception classes can be classified into three major types:
  - System Errors.
  - Runtime Exceptions.
  - Other exceptions.

Exception Types: System Errors

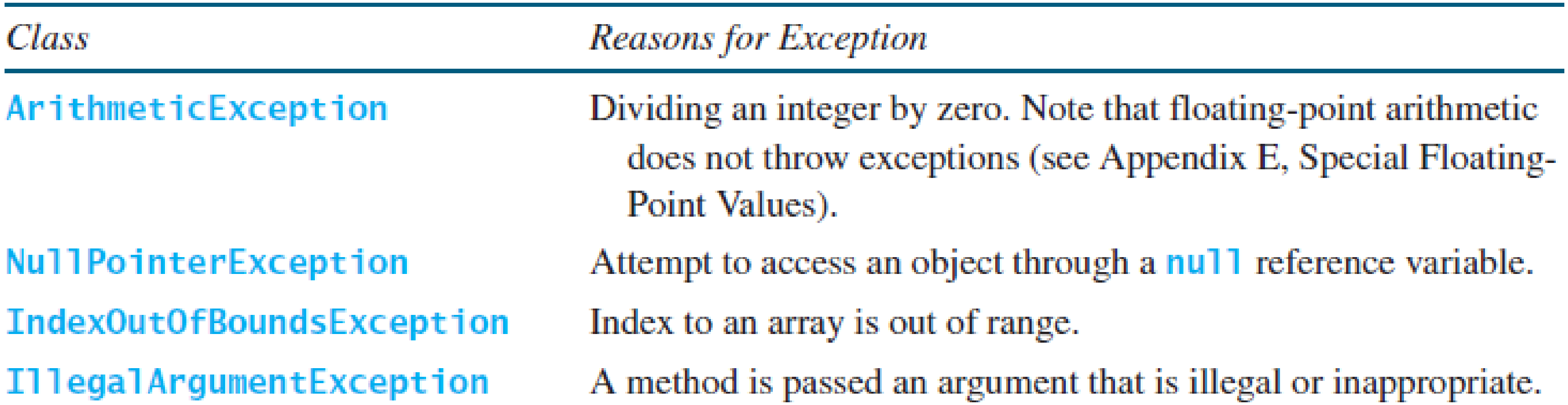
- *System errors* are thrown by the JVM and are represented in the *Error* class.
- The *Error* class describes internal system errors, though such errors rarely occur.
  - If one occurs, there is little you can do beyond notifying the user and trying to terminate the program gracefully.



Class	*Reasons for Exception*
`LinkageError`	A class has some dependency on another class, but the latter class has changed incompatibly after the compilation of the former class.
`VirtualMachineError`	The JVM is broken or has run out of the resources it needs in order to continue operating.

Exception Types: Runtime Exceptions

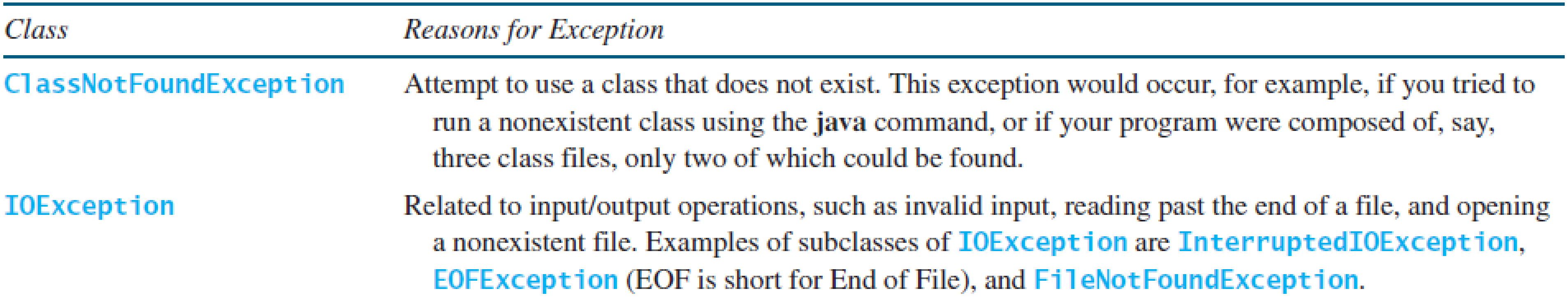
- Runtime exceptions are represented in the RunTimeException class.
  - Describes programming errors, such as bad casting, accessing an out-of-bounds array, and numeric errors.



Class	*Reasons for Exception*
`ArithmeticException`	Dividing an integer by zero. Note that floating-point arithmetic does not throw exceptions (see Appendix E, Special Floating-Point Values).
`NullPointerException`	Attempt to access an object through a `null` reference variable.
`IndexOutOfBoundsException`	Index to an array is out of range.
`IllegalArgumentException`	A method is passed an argument that is illegal or inappropriate.

Exception Types: Other Exceptions

- Other exceptions are represented in the *Exception* class.
  - Describes errors caused by your program and by external circumstances.



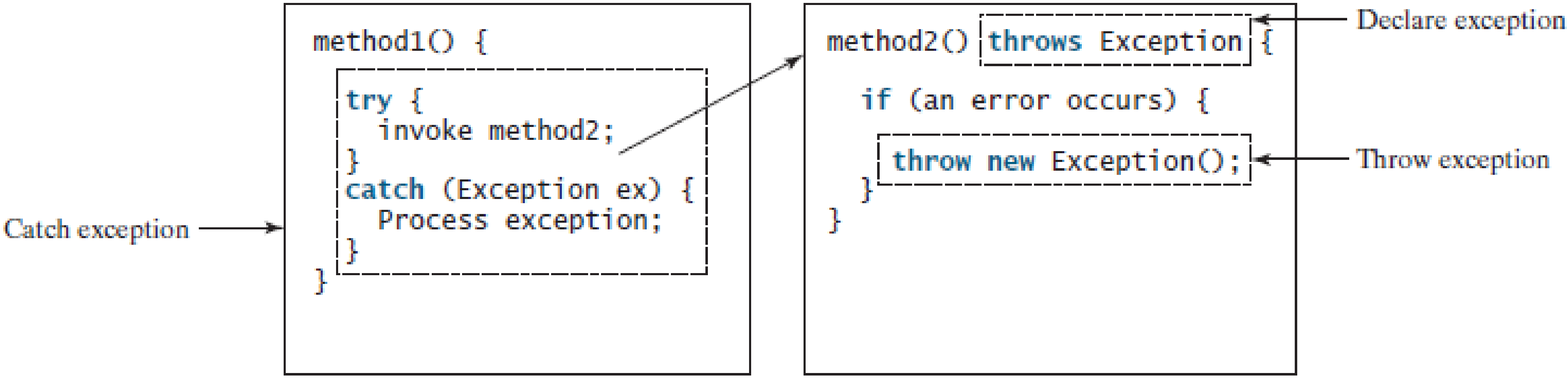
Class	*Reasons for Exception*
ClassNotFoundException	Attempt to use a class that does not exist. This exception would occur, for example, if you tried to run a nonexistent class using the **java** command, or if your program were composed of, say, three class files, only two of which could be found.
IOException	Related to input/output operations, such as invalid input, reading past the end of a file, and opening a nonexistent file. Examples of subclasses of IOException are InterruptedIOException, EOFException (EOF is short for End of File), and FileNotFoundException.

Exception Types: Checked and Unchecked Exceptions

- *RunTimeException*, *Error* and their subclasses are known as *unchecked exceptions*.
  - In most cases, unchecked exceptions reflect programming logic errors that are unrecoverable.
  - To avoid cumbersome overuse of try-catch blocks, Java does not mandate that you write code to catch or declare unchecked exceptions.
- All other exceptions are known as *checked exceptions*.
  - The compiler forces the programmer to check and deal with them in a try-catch block or declare it in the method header.

More On Exception Handling

- Java exception handling model is based on three operations:
  - Declaring an exception.
  - Throwing an exception.
  - Catching an exception.



```
method1() {

  try {
    invoke method2;
  }
  catch (Exception ex) {
    Process exception;
  }
}
```

```
method2() throws Exception {

  if (an error occurs) {

    throw new Exception();
  }
}
```

More On Exception Handling
Declaring Exceptions

- Every method must state the types of *checked exceptions* it might throw.
  - This is known as *declaring exceptions*.
  - Java does not require that you declare unchecked exceptions explicitly in the method.
- To declare an exception in a method, use the *throws* keyword in the method header.
- Example:

public void myMethod() throws IOException

More On Exception Handling Declaring Exceptions (Cont.)

- If the method might throw multiple exceptions, add a list of the exceptions, separated by commas after throws:

  public void myMethod() throws Exception1, Exception2, …, ExceptionN

- If a method does not declare exceptions in the superclass, you cannot override it to declare exceptions in the subclass.

More On Exception Handling
Throwing Exceptions

- A program that detects an error can create an instance of an appropriate exception type and throw it.
  - This is known as *throwing an exception*.
- Example:

  Suppose the program detects that a negative argument is passed when it should be nonnegative, the program can create an instance of *IllegalArgumentException* and throw it as follows:

```
IllegalArgumentException ex = new
IllegalArgumentException (“Wrong Argument”);

throw ex;
```

OR

```
throw new IllegalArgumentException (“Wrong Argument”);
```

More On Exception Handling Throwing Exceptions (Cont.)

- In general, each exception class in the Java API has at least two constructors:
  - A no-arg constructor, and
  - A constructor with a String argument that describes the exception.
    - The argument is called the *exception message*, which can be obtained using *getMessage()*;
- Note that:
  - The keyword to declare an exception is *throws*.
  - The keyword to throw an exception is *throw*.

More On Exception Handling
Catching Exceptions

- When an exception is thrown, it can be caught and handled in a *try-catch* block, as follows:

```
try{
        statements; //statements that may throw exception
}
catch (Exception1 exVar1){
        handler for exception1;
}
catch (Exception2 exVar2){
        handler for exception2;
}
…
catch (ExceptionN exVarN){
        handler for exceptionN;
}
```

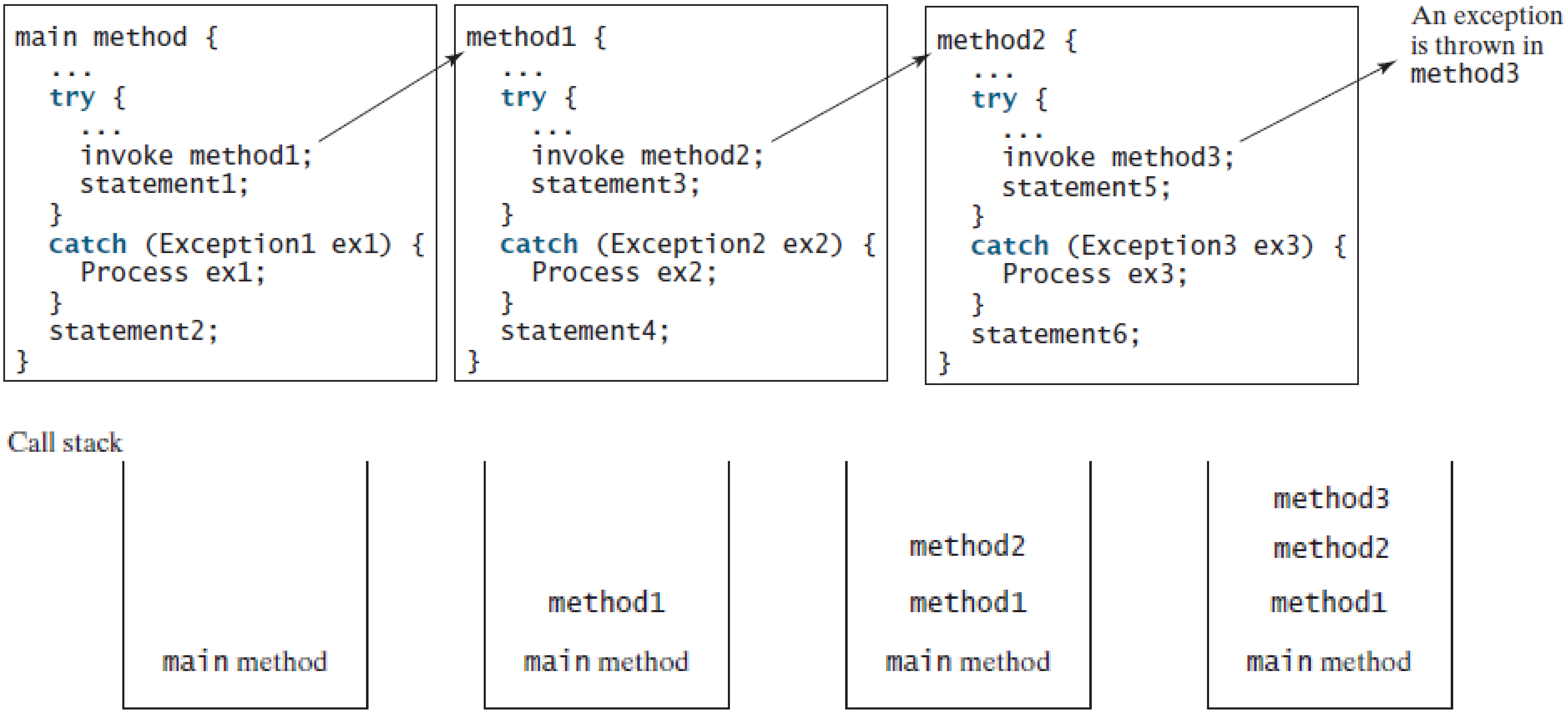
More On Exception Handling Catching Exceptions (Cont.)

- If no exceptions arise during the execution of the *try* block, the *catch* blocks are skipped.
- If one of the statements inside the *try* block throws an exception:
  - Java skips the remaining statements in the *try* block, and
  - Starts the process of finding the code to handle the exception, which is called *catching an exception*.
    - The code that handles the exception is called the *exception handler*.

More On Exception Handling Catching Exceptions (Cont.)

- An *exception handler* is found by *propagating the exception* backward through a chain of method calls, starting from the current method.
- Each catch block is examined in turn, from first to last, to see whether the type of the exception object is an instance of the exception class in the catch block
  - If so, the exception object is assigned to the variable declared, and the code in the catch block is executed.
  - If no handler is found, Java exits this method, passes the exception to the method that invoked the method, and continues the same process to find a handler.
  - If no handler is found in the chain of methods being invoked, the program terminates and prints an error message to the console.

Catching Exceptions: An Example



```
main method {
  ...
  try {
    ...
    invoke method1;
    statement1;
  }
  catch (Exception1 ex1) {
    Process ex1;
  }
  statement2;
}
```

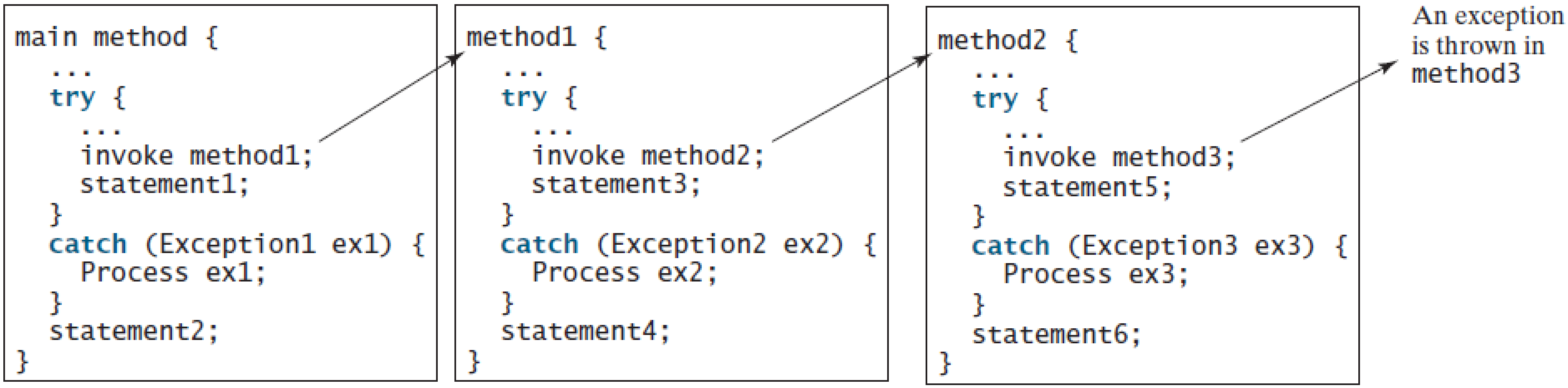
```
method1 {
  ...
  try {
    ...
    invoke method2;
    statement3;
  }
  catch (Exception2 ex2) {
    Process ex2;
  }
  statement4;
}
```

```
method2 {
  ...
  try {
    ...
    invoke method3;
    statement5;
  }
  catch (Exception3 ex3) {
    Process ex3;
  }
  statement6;
}
```

An exception is thrown in method3

Call stack

Catching Exceptions: An Example (Case 1)



```
main method {
  ...
  try {
    ...
    invoke method1;
    statement1;
  }
  catch (Exception1 ex1) {
    Process ex1;
  }
  statement2;
}
```

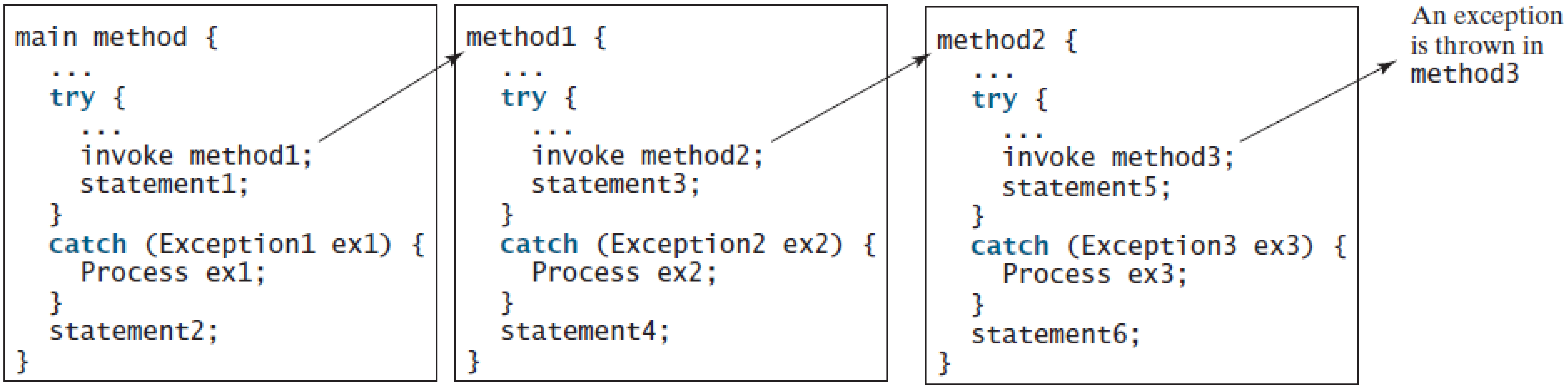
```
method1 {
  ...
  try {
    ...
    invoke method2;
    statement3;
  }
  catch (Exception2 ex2) {
    Process ex2;
  }
  statement4;
}
```

```
method2 {
  ...
  try {
    ...
    invoke method3;
    statement5;
  }
  catch (Exception3 ex3) {
    Process ex3;
  }
  statement6;
}
```

An exception is thrown in method3

- If the exception type is Exception3:
  - It is caught by the catch block for handling exception ex3 in method2.
  - Statement 5 is skipped, and statement6 is executed.

Catching Exceptions: An Example (Case 2)



```
main method {
  ...
  try {
    ...
    invoke method1;
    statement1;
  }
  catch (Exception1 ex1) {
    Process ex1;
  }
  statement2;
}
```

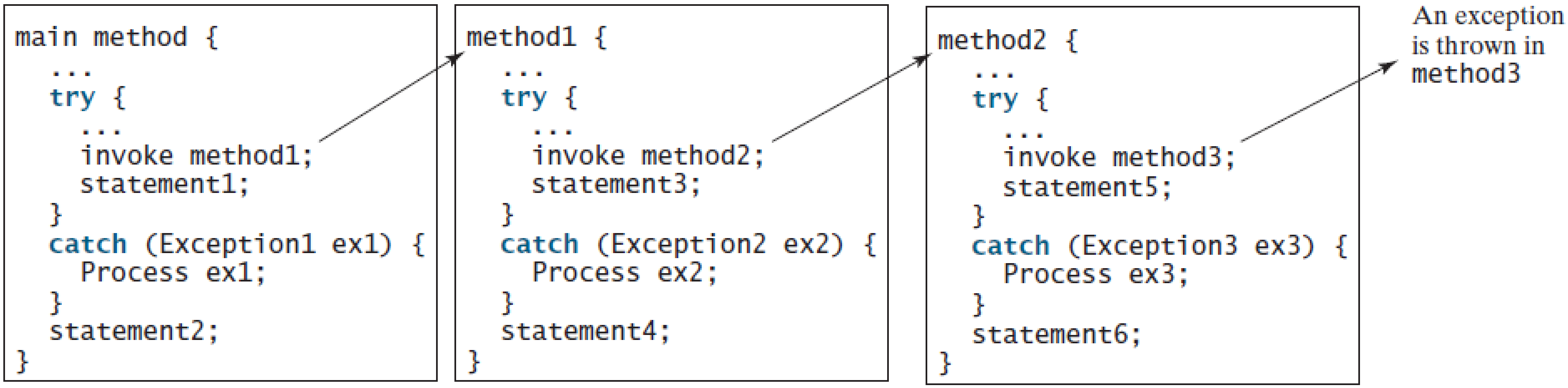
```
method1 {
  ...
  try {
    ...
    invoke method2;
    statement3;
  }
  catch (Exception2 ex2) {
    Process ex2;
  }
  statement4;
}
```

```
method2 {
  ...
  try {
    ...
    invoke method3;
    statement5;
  }
  catch (Exception3 ex3) {
    Process ex3;
  }
  statement6;
}
```

An exception is thrown in method3

- If the exception type is Exception2:
  - Method2 is aborted, the control is returned to method1.
  - The exception is caught by the catch block for handling exception ex2 in method1.
  - Statement3 is skipped, and statement4 is executed.

Catching Exceptions: An Example (Case 3)



```
main method {
  ...
  try {
    ...
    invoke method1;
    statement1;
  }
  catch (Exception1 ex1) {
    Process ex1;
  }
  statement2;
}
```

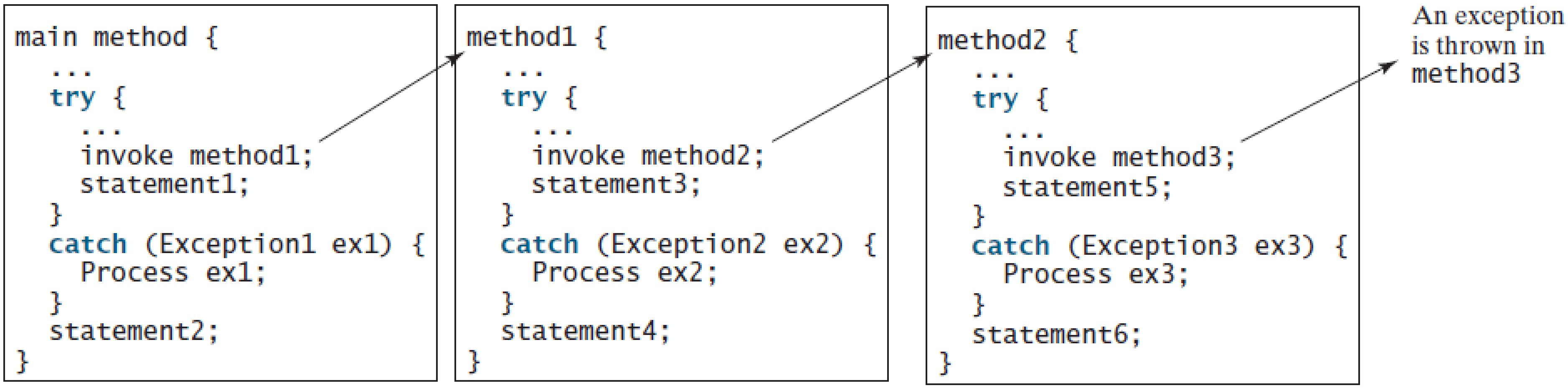
```
method1 {
  ...
  try {
    ...
    invoke method2;
    statement3;
  }
  catch (Exception2 ex2) {
    Process ex2;
  }
  statement4;
}
```

```
method2 {
  ...
  try {
    ...
    invoke method3;
    statement5;
  }
  catch (Exception3 ex3) {
    Process ex3;
  }
  statement6;
}
```

An exception is thrown in method3

- If the exception type is Exception1:
  - Method2 and method1 are aborted, the control is returned to the main method.
  - The exception is caught by the catch block for handling exception ex1 in the main method.
  - Statement1 is skipped, and statement2 is executed.

Catching Exceptions: An Example (Case 4)



```
main method {
   ...
   try {
     ...
     invoke method1;
     statement1;
   }
   catch (Exception1 ex1) {
     Process ex1;
   }
   statement2;
}
```

```
method1 {
   ...
   try {
     ...
     invoke method2;
     statement3;
   }
   catch (Exception2 ex2) {
     Process ex2;
   }
   statement4;
}
```

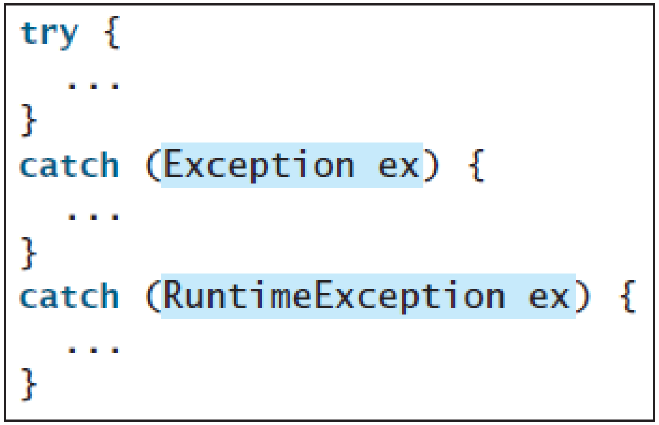
```
method2 {
   ...
   try {
     ...
     invoke method3;
     statement5;
   }
   catch (Exception3 ex3) {
     Process ex3;
   }
   statement6;
}
```

An exception is thrown in method3

- If the exception type is not caught in method2, method1, or the main method:
  - Program terminates, and statement1 and statement2 are not executed.

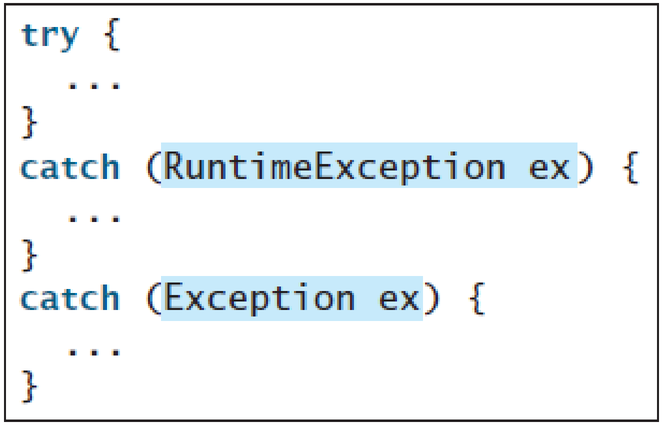
More on Catching Exceptions

- Various exception classes can be derived from a common superclass.
  - If a *catch* block catches exception objects of a superclass, it can catch all the exception objects of the subclasses of the superclass.
- The order in which exceptions are specified in *catch* blocks is important.
  - A compile error will result if a *catch* block for a superclass type appears before a *catch* block for a subclass type.



```
try {
  ...
}
catch (Exception ex) {
  ...
}
catch (RuntimeException ex) {
  ...
}
```

(a) Wrong order

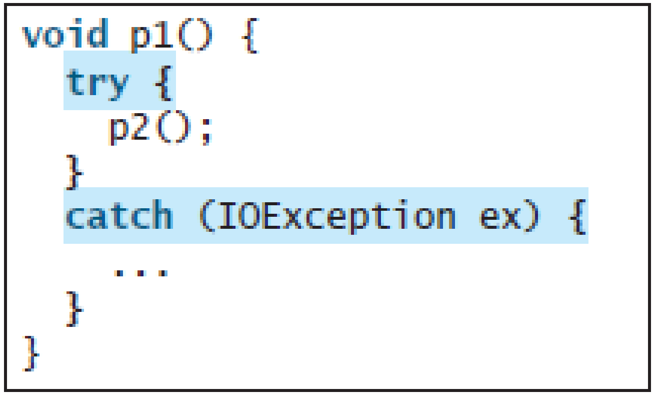


```
try {
  ...
}
catch (RuntimeException ex) {
  ...
}
catch (Exception ex) {
  ...
}
```

(b) Correct order

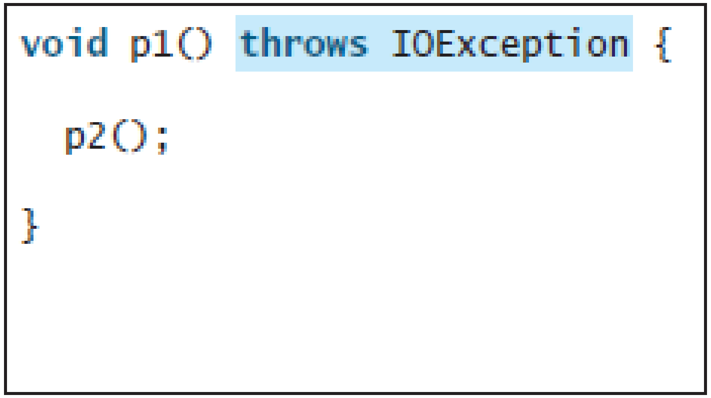
More on Catching Exceptions (Cont.)

- Java forces you to deal with checked exceptions.
- If a method declares a checked exception (i.e., an exception other than *Error* or *RuntimeException*), you must invoke it in a *try-catch* block or declare to throw the exception in the calling method.
  - For example, suppose that method *p1* invokes method *p2*, and *p2* may throw a checked exception (e.g., *IOException*); you have to write the code as shown in (a) or (b) below.



```
void p1() {
  try {
    p2();
  }
  catch (IOException ex) {
    ...
  }
}
```

(a) Catch exception

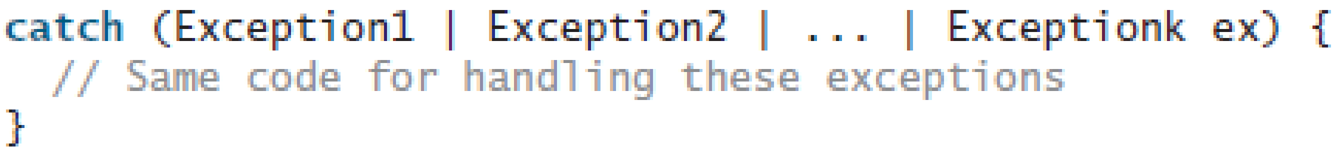


```
void p1() throws IOException {

  p2();

}
```

(b) Throw exception

More on Catching Exceptions (Cont.)

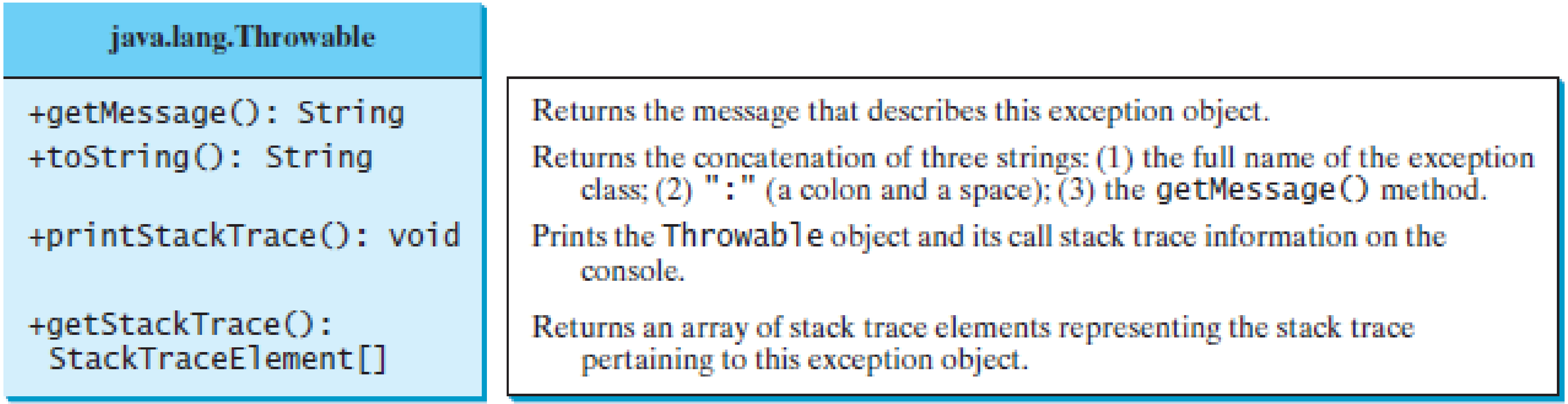
- You can use the new JDK 7 multi-catch feature to simplify coding for the exceptions with the same handling code.
- The syntax is:



```
catch (Exception1 | Exception2 | ... | Exceptionk ex) {
  // Same code for handling these exceptions
}
```

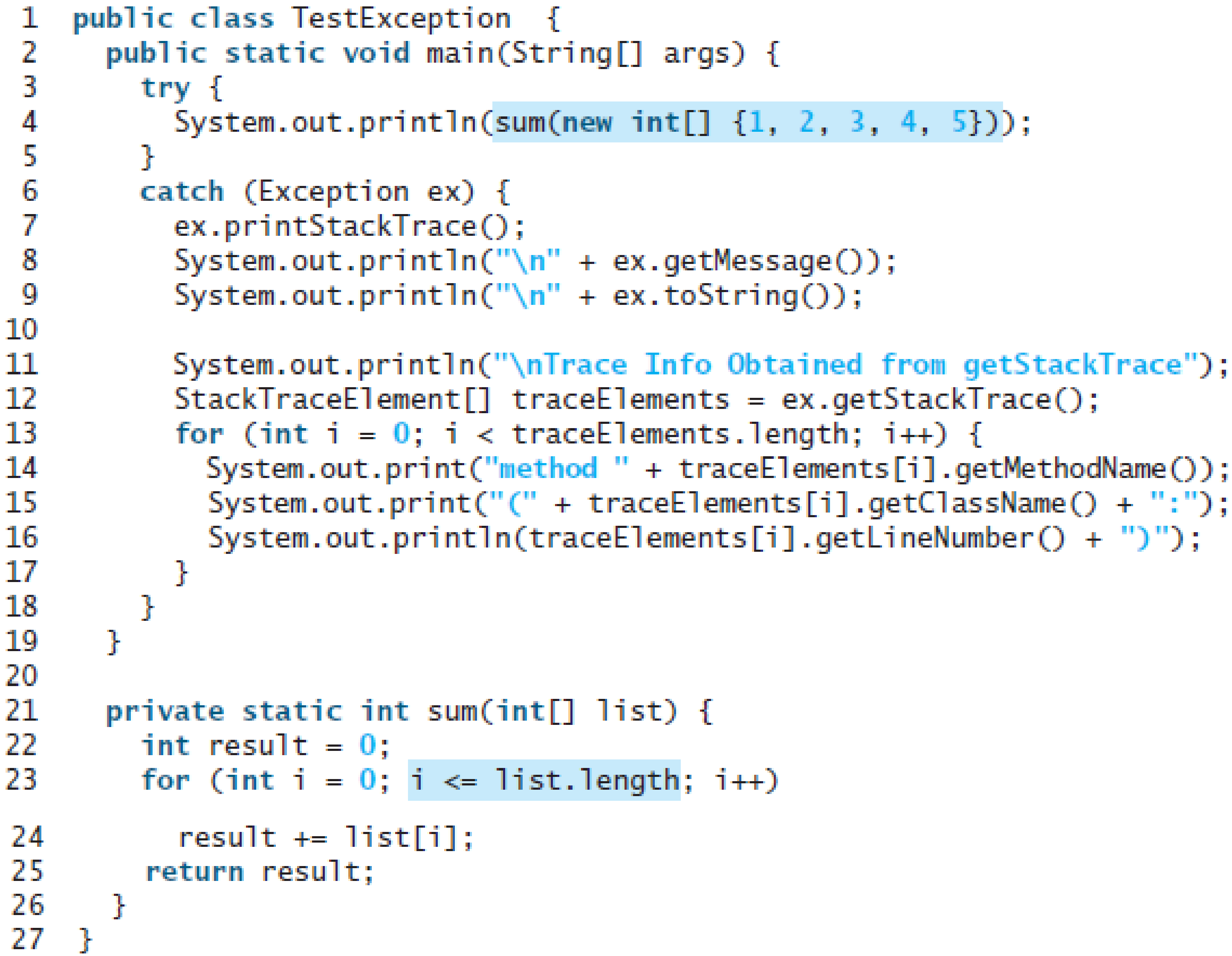
Getting Information from Exceptions

- An exception object contains valuable information about the exception.



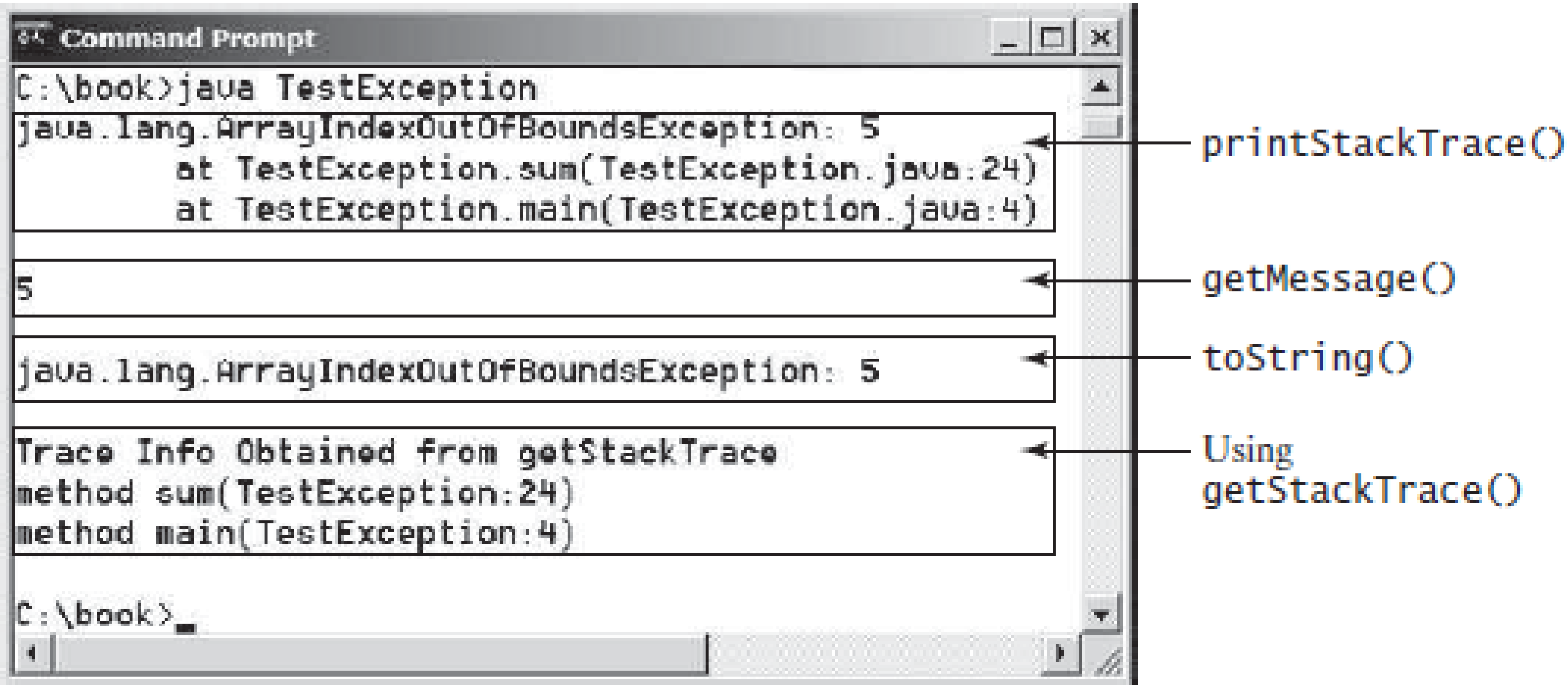
java.lang.Throwable	
+getMessage(): String	Returns the message that describes this exception object.
+toString(): String	Returns the concatenation of three strings: (1) the full name of the exception class; (2) ":" (a colon and a space); (3) the getMessage() method.
+printStackTrace(): void	Prints the Throwable object and its call stack trace information on the console.
+getStackTrace(): StackTraceElement[]	Returns an array of stack trace elements representing the stack trace pertaining to this exception object.

TestException.java



```java
public class TestException  {
  public static void main(String[] args) {
    try {
      System.out.println(sum(new int[] {1, 2, 3, 4, 5}));
    }
    catch (Exception ex) {
      ex.printStackTrace();
      System.out.println("\n" + ex.getMessage());
      System.out.println("\n" + ex.toString());

      System.out.println("\nTrace Info Obtained from getStackTrace");
      StackTraceElement[] traceElements = ex.getStackTrace();
      for (int i = 0; i < traceElements.length; i++) {
        System.out.print("method " + traceElements[i].getMethodName());
        System.out.print("(" + traceElements[i].getClassName() + ":");
        System.out.println(traceElements[i].getLineNumber() + ")");
      }
    }
  }

  private static int sum(int[] list) {
    int result = 0;
    for (int i = 0; i <= list.length; i++)
      result += list[i];
    return result;
  }
}
```

TestException.java (Output)



```
C:\book>java TestException
java.lang.ArrayIndexOutOfBoundsException: 5
        at TestException.sum(TestException.java:24)
        at TestException.main(TestException.java:4)

5

java.lang.ArrayIndexOutOfBoundsException: 5

Trace Info Obtained from getStackTrace
method sum(TestException:24)
method main(TestException:4)

C:\book>_
```

printStackTrace()

getMessage()

toString()

Using getStackTrace()

Example: Declaring, Throwing, and Catching Exceptions (CircleWithException.java)

```
public class CircleWithException{
    private double radius;
    private static int numberOfObjects=0;

    public CircleWithException(double newRadius){
        setRadius(newRadius);
        numberOfObjects++;
    }

    public void setRadius(double newRadius) throws IllegalArgumentException{
        if (newRadius>=0) radius = newRadius;
        else throw new IllegalArgumentException("Radius cannot be negative!");
    }

    public static int getNumberOfObjects(){
        return numberOfObjects;
    }
}
```

Example: Declaring, Throwing, and Catching Exceptions (TestCircleWithException.java)

```
public class TestCircleWithException{
   public static void main (String [] args){
      try{
         CircleWithException C1 = new CircleWithException(5);
         CircleWithException C2 = new CircleWithException(-5);
         CircleWithException C3 = new CircleWithException(0);
      }
      catch (IllegalArgumentException ex){
         System.out.println(ex);
      }
      System.out.println("Number of circle objects created: "+
            CircleWithException.getNumberOfObjects());
   }
}
```

Output:

```
java.lang.IllegalArgumentException: Radius cannot be negative!
Number of circle objects created: 1
```

The *finally* Clause

- The *finally* clause is executed under all circumstances, regardless of whether an exception occurs in the *try* block or is caught.
- The syntax for the *finally* clause is as follows:

```
try {
        Statements
}
catch (TheException ex){
        handling ex;
}
finally{
        finalStatements;
}
```

The *finally* Clause (Cont.)

- If no exception arises in the *try* block:
  - The *finally* clause is executed, and
  - The next statement after the *try* statement is executed.
- If a statement causes an exception in the *try* block that is caught in the *catch* block:
  - The rest of the statements in the *try* block are skipped,
  - The *catch* block is executed,
  - The *finally* clause is executed, and
  - The next statement after the *try* statement is executed.
- If a statement causes an exception that is not caught in any *catch* block:
  - The other statements in the *try* block are skipped,
  - The *finally* clause is executed, and
  - The exception is passed to the caller of this method.
- Note: the *finally* block executes even if there is a *return* statement prior to reaching the *finally* block.

When to Use Exceptions?

- The *try* block contains the code that is executed in normal circumstances.
- The *catch* block contains the code that is executed in exceptional circumstances.
- Exception handling separates error-handling code from normal programming tasks, thus making programs easier to read and to modify.
- Be aware, however, that exception handling usually requires more time and resources.
  - Requires instantiating a new exception object,
  - Rolling back the call stack, and
  - Propagating the exception through the chain of methods invoked to search for the handler.

When to Use Exceptions? (Cont.)

- An exception occurs in a method:
  - If you want the exception to be processed by the method's caller, you should create an exception object and throw it.
  - If you can handle the exception in the method where it occurs, there is no need to throw or use exception objects.
    - Simple errors that may occur in individual methods are best handled without throwing exceptions.
    - This can be done by using *if* statements to check for errors.

Rethrowing Exceptions

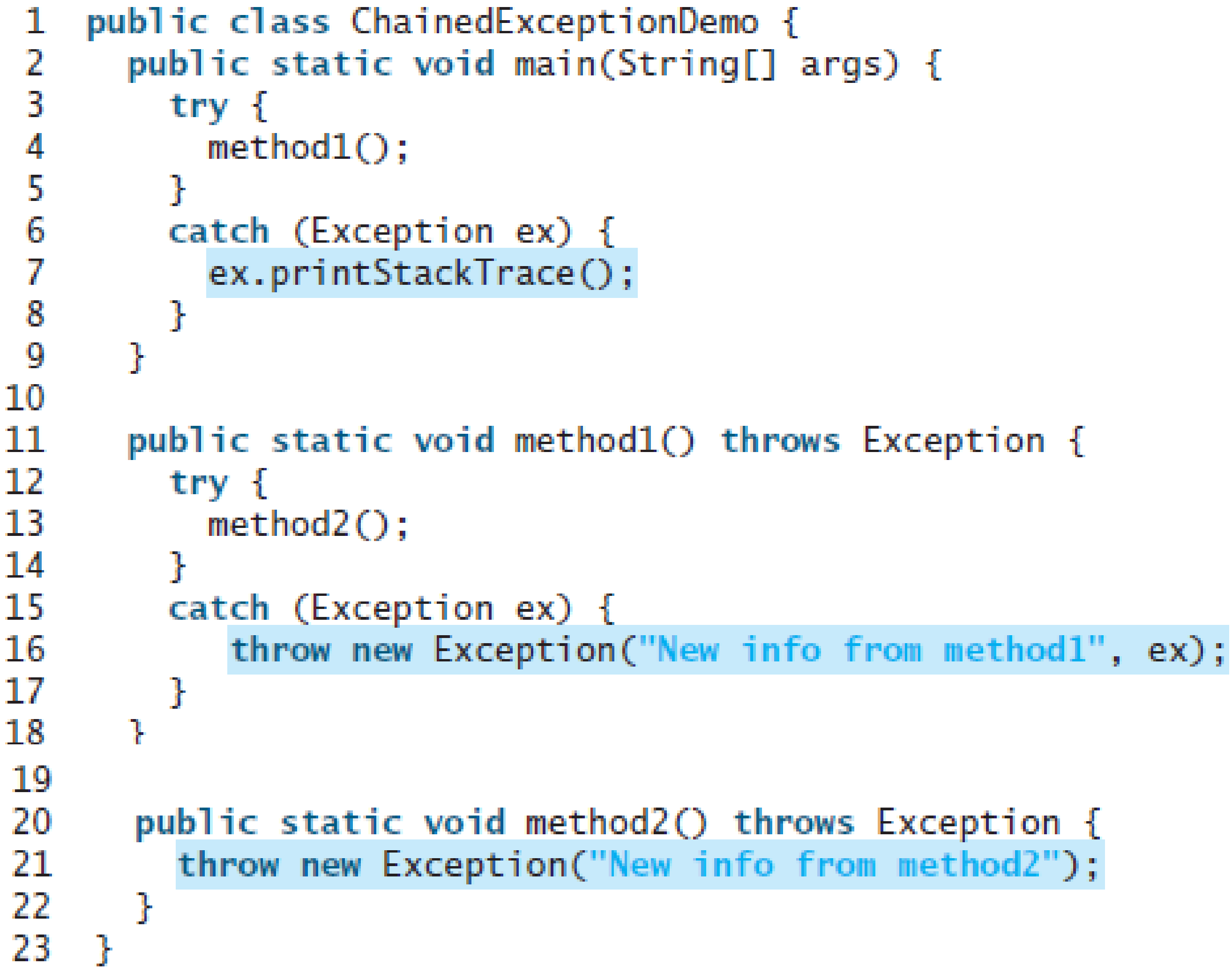
- Java allows an exception handler to *rethrow* the exception if 1) the handler cannot fully process the exception or 2) simply wants to let its caller be notified of the exception.
- The syntax for re-throwing an exception is as follows:

```
try{
        statements;
}
catch (TheException ex){
        perform operations;
        throw ex;
}
```

Chained Exceptions

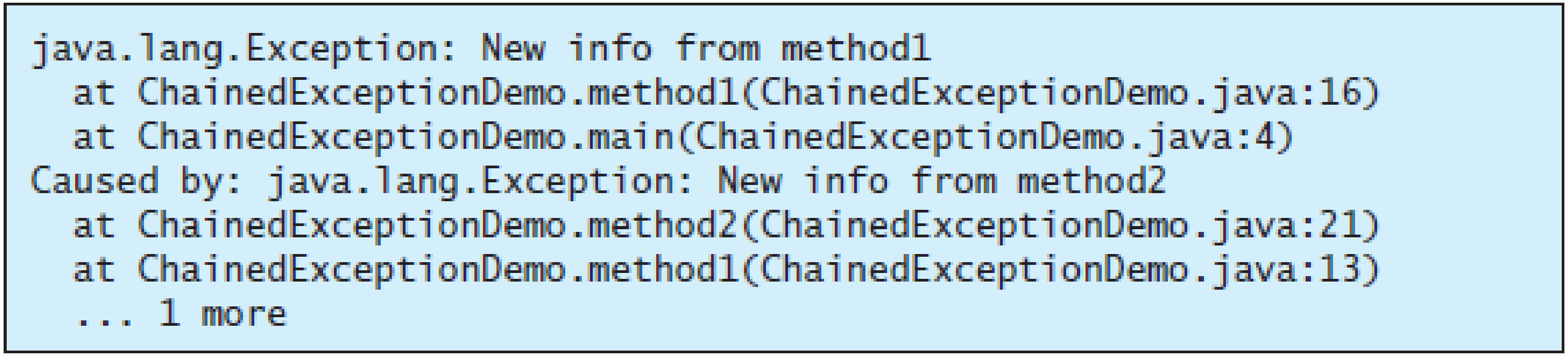
- Throwing an exception along with another exception forms a chained exception.
- Sometimes, you may need to throw a new exception (with additional information) along with the original exception.
- This is called *chained exceptions*.

Chained Exceptions ChainedExceptionDemo.java



```java
public class ChainedExceptionDemo {
  public static void main(String[] args) {
    try {
      method1();
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }
  }

  public static void method1() throws Exception {
    try {
      method2();
    }
    catch (Exception ex) {
       throw new Exception("New info from method1", ex);
    }
  }

  public static void method2() throws Exception {
    throw new Exception("New info from method2");
  }
}
```

Chained Exceptions ChainedExceptionDemo.java (Output)

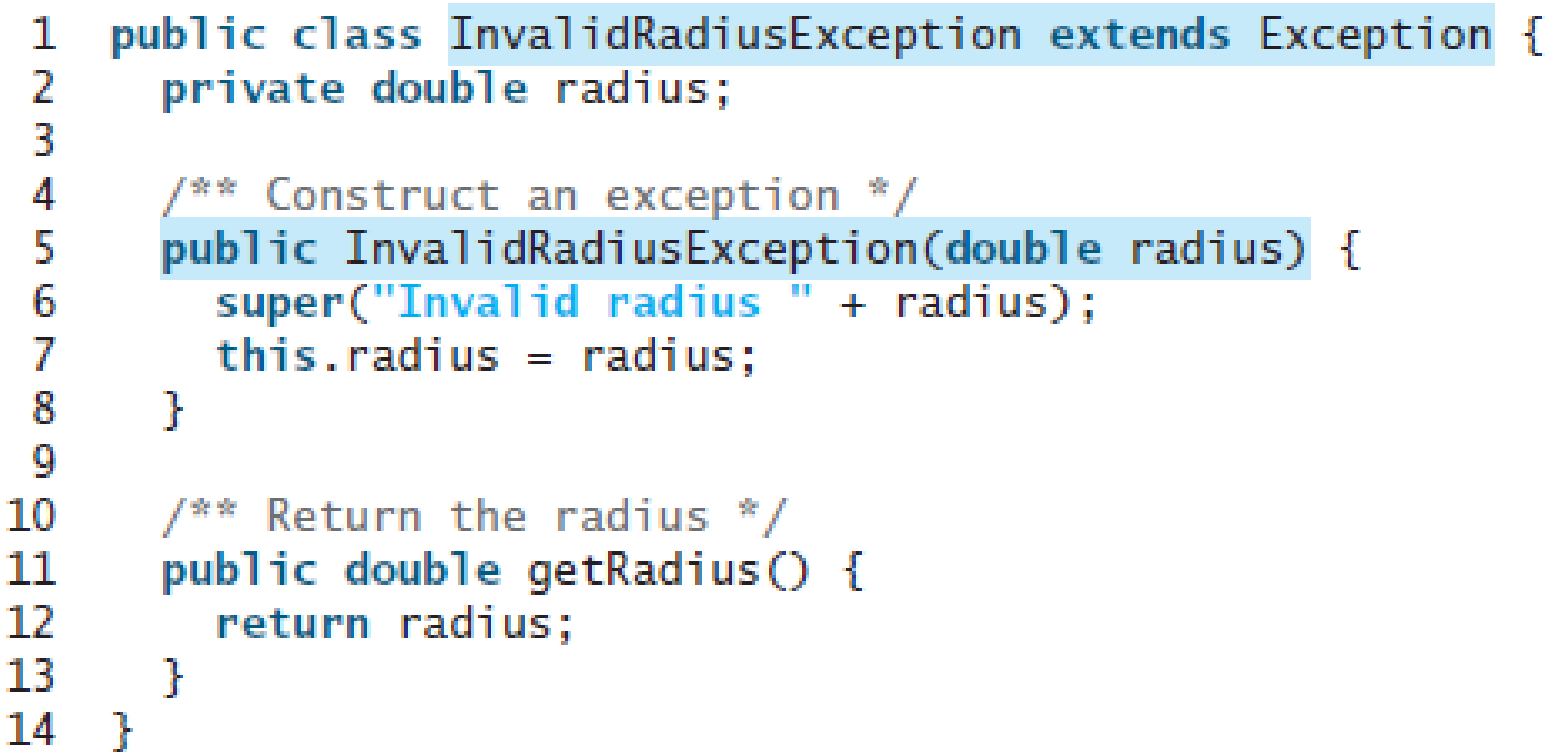


```
java.lang.Exception: New info from method1
  at ChainedExceptionDemo.method1(ChainedExceptionDemo.java:16)
  at ChainedExceptionDemo.main(ChainedExceptionDemo.java:4)
Caused by: java.lang.Exception: New info from method2
  at ChainedExceptionDemo.method2(ChainedExceptionDemo.java:21)
  at ChainedExceptionDemo.method1(ChainedExceptionDemo.java:13)
  ... 1 more
```

Defining Custom Exception Classes

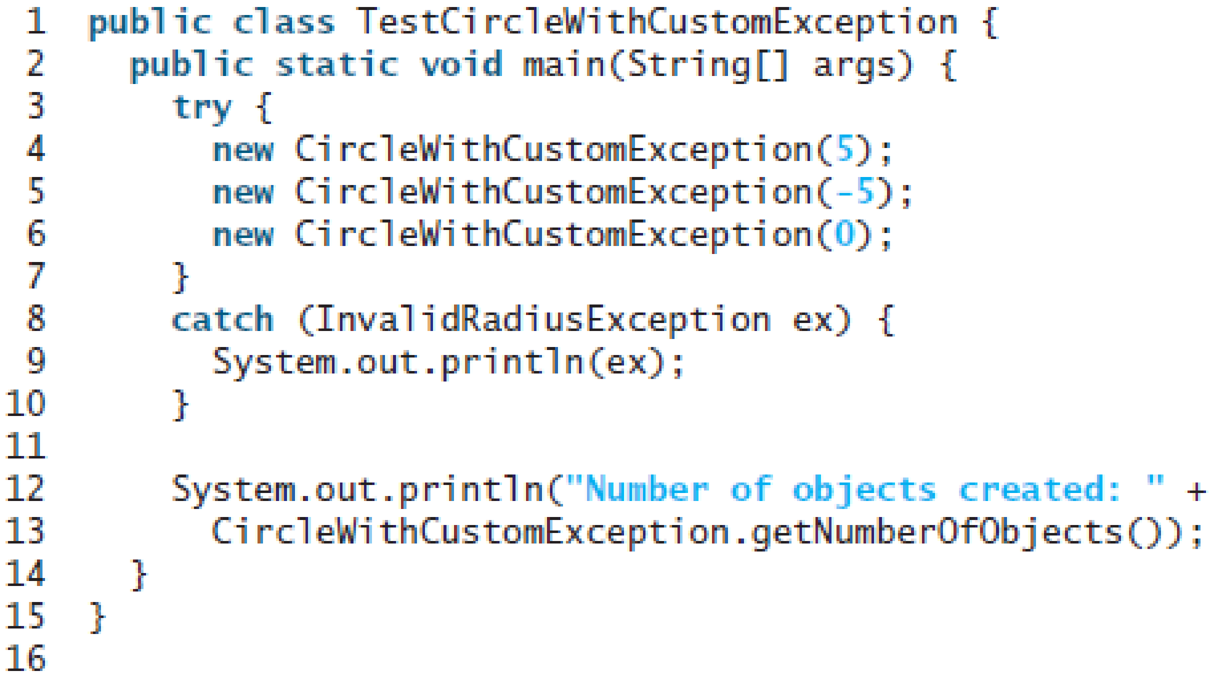
- Java provides quite a few exception classes.
- Use them whenever possible instead of defining your own exception classes.
- However, if you run into a problem that cannot be adequately described by the predefined exception classes, you can create your own exception class, derived from *Exception* or from a subclass of *Exception*, such as *IOException*.

Defining Custom Exception Classes InvalidRadiusException.java



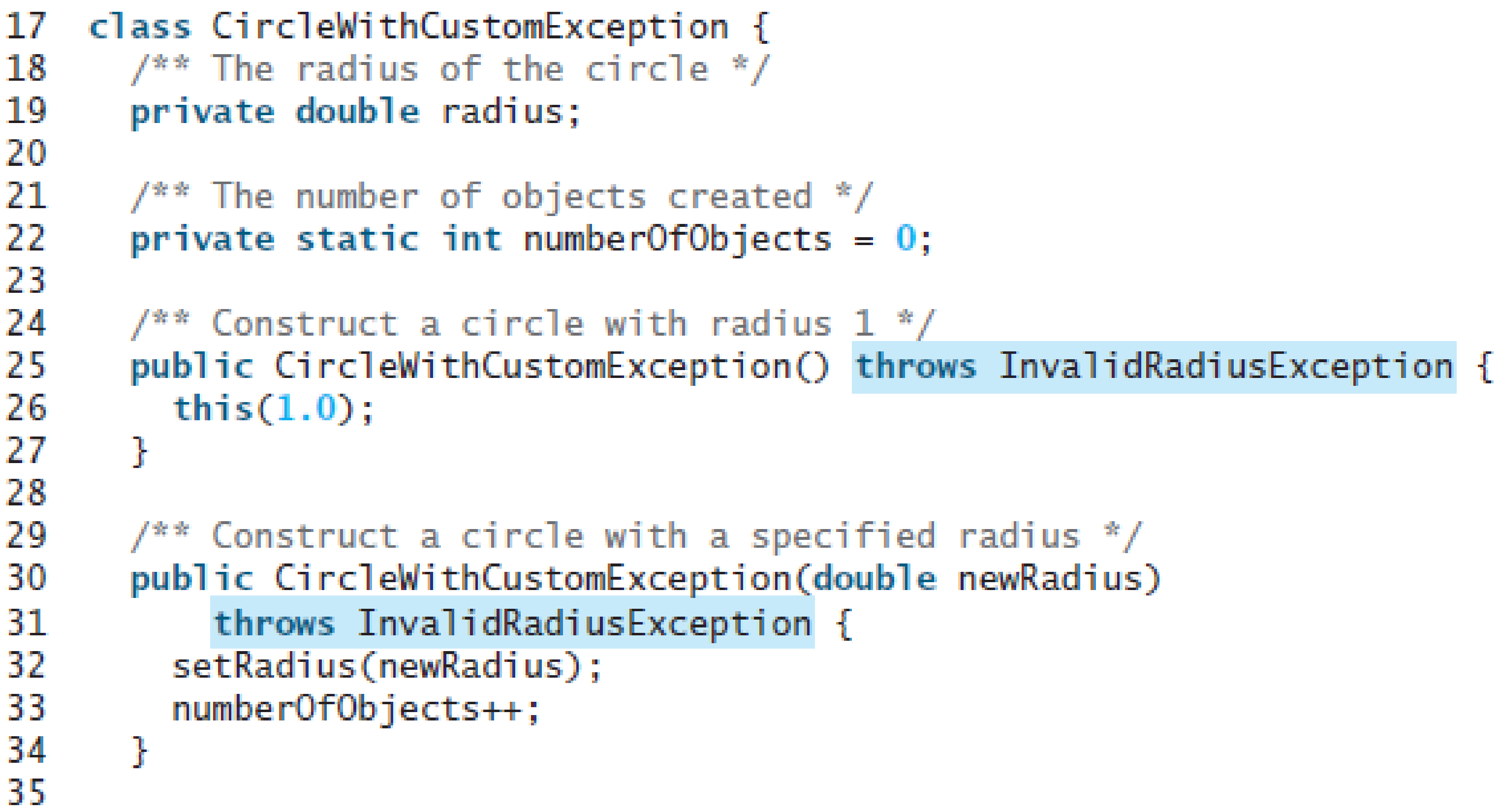
```java
public class InvalidRadiusException extends Exception {
  private double radius;

  /** Construct an exception */
  public InvalidRadiusException(double radius) {
    super("Invalid radius " + radius);
    this.radius = radius;
  }

  /** Return the radius */
  public double getRadius() {
    return radius;
  }
}
```

Defining Custom Exception Classes TestCircleWithCustomException.java



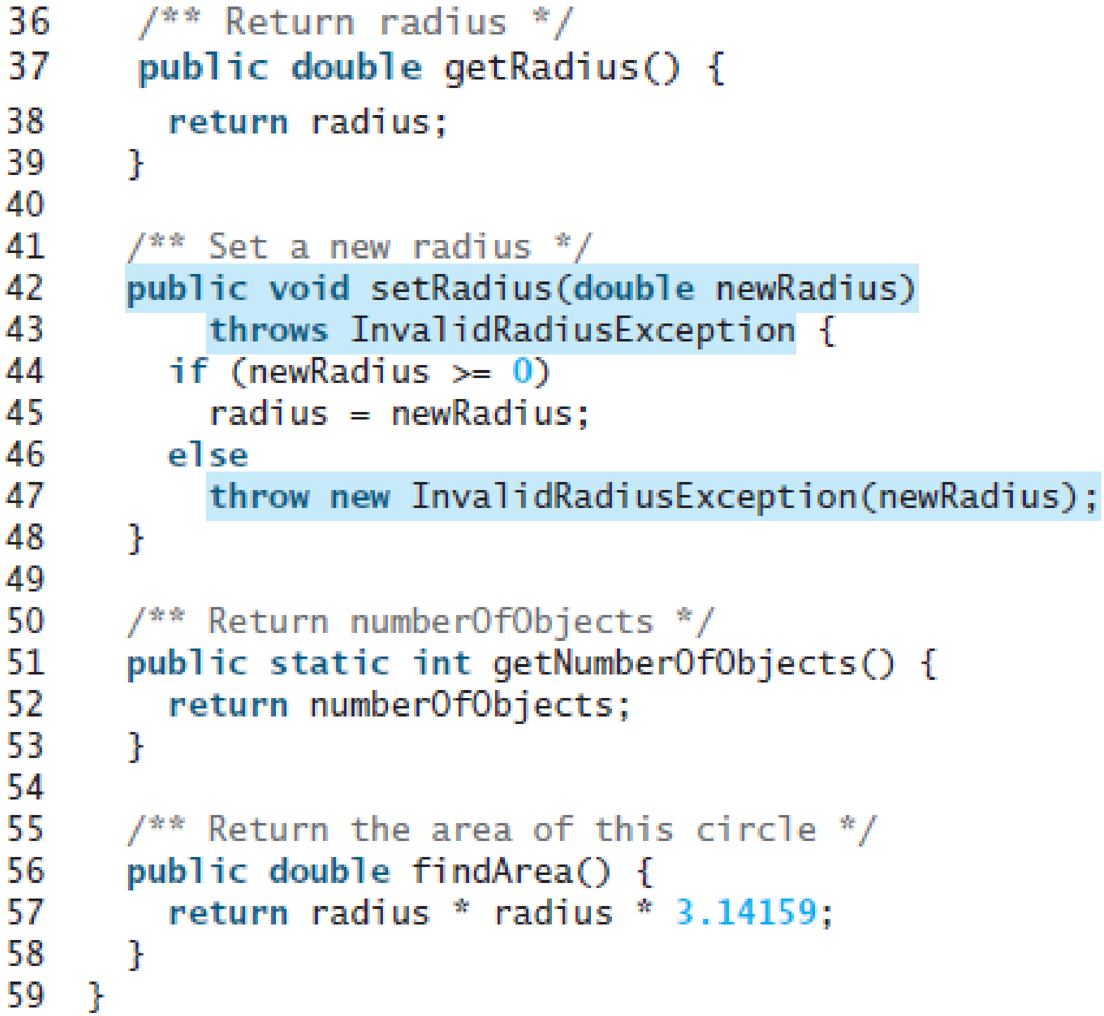
```java
public class TestCircleWithCustomException {
  public static void main(String[] args) {
    try {
      new CircleWithCustomException(5);
      new CircleWithCustomException(-5);
      new CircleWithCustomException(0);
    }
    catch (InvalidRadiusException ex) {
      System.out.println(ex);
    }

    System.out.println("Number of objects created: " +
      CircleWithCustomException.getNumberOfObjects());
  }
}
```

Defining Custom Exception Classes TestCircleWithCustomException.java (Cont.)



```java
class CircleWithCustomException {
  /** The radius of the circle */
  private double radius;

  /** The number of objects created */
  private static int numberOfObjects = 0;

  /** Construct a circle with radius 1 */
  public CircleWithCustomException() throws InvalidRadiusException {
    this(1.0);
  }

  /** Construct a circle with a specified radius */
  public CircleWithCustomException(double newRadius)
      throws InvalidRadiusException {
    setRadius(newRadius);
    numberOfObjects++;
  }

```

Defining Custom Exception Classes TestCircleWithCustomException.java (Cont.)



```java
  /** Return radius */
  public double getRadius() {
    return radius;
  }

  /** Set a new radius */
  public void setRadius(double newRadius)
      throws InvalidRadiusException {
    if (newRadius >= 0)
      radius = newRadius;
    else
      throw new InvalidRadiusException(newRadius);
  }

  /** Return numberOfObjects */
  public static int getNumberOfObjects() {
    return numberOfObjects;
  }

  /** Return the area of this circle */
  public double findArea() {
    return radius * radius * 3.14159;
  }
}
```