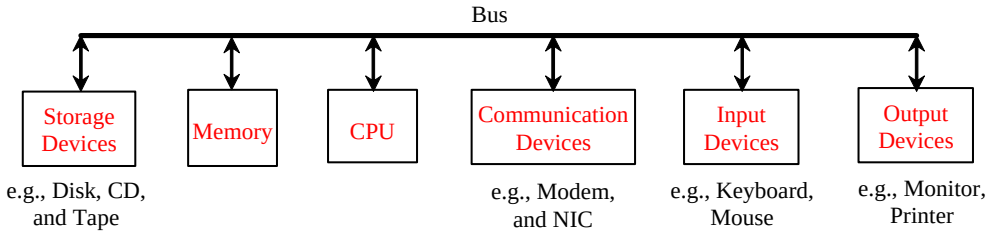
JAVA Programming 2013-2 Part-I

1. Introduction

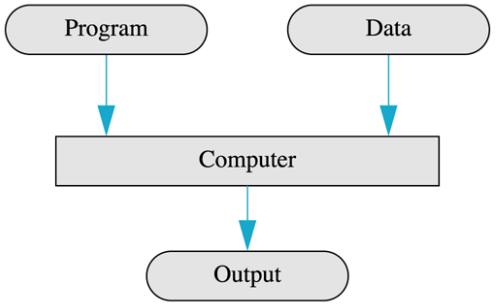
1.1. Computer Systems

- Computer system includes **Software** and **Hardware**
- Computer hardware includes data processing unites (CPU, GPU), data storage devices (RAM, hard disk, flash disk, tape, floppy disk), Input devices (keyboard, mouse), output devices (monitor, printer) and communication devices(NIC, modem)



- Computer software is the collection of **programs** used by a computer. For example
  - Editors
  - Translators
  - System Managers
  - MS office
- Computer ***program*** is a set of instructions to the computer that perform a specific task.
- You tell a computer what to do through programs. Without programs, a computer is an empty machine. Computers do not understand human languages, so you need to use computer languages to communicate with them.

Simple View of Running a Program



- Programs are written using programming languages.
- programming languages can be divided into three categories
  1. Machine Language
  2. Assembly Language
  3. High-Level Language

- **Machine language** is a set of primitive instructions built into every computer. The instructions are in the form of **binary code**, so you have to enter binary codes for various instructions. Program with native machine language is a tedious process. Moreover the programs are **difficult to read and modify**. For example, to add two numbers, you might write an instruction in binary like this: 1101101010011010
- **Assembly languages** were developed to make programming easy. Since the computer cannot understand assembly language, however, a program called **assembler** is used to convert assembly language programs into **machine code**. For example, to add two numbers, you might write an instruction in assembly code like this: ADDF3 R1, R2, R3
- The **high-level languages** are English-like and **easy to learn and program**. For example, the following is a high-level language statement that computes the area of a circle with radius 5:
  area = 5 * 5 * 3.1415;
- A program written in a high-level language is called a ***source program*** or ***source code***. Because a computer cannot understand a source program, a source program must be translated into machine code for execution. The translation can be done using another programming tool called an ***interpreter*** or a ***compiler***.
- Few popular High-Level Languages are listed below

Programming language	Description
Fortran	Fortran (FORmula TRANslator) was developed by IBM Corporation in the mid-1950s to be used for scientific and engineering applications that require complex mathematical computations. It's still widely used and its latest versions are object oriented.
COBOL	COBOL (COmmon Business Oriented Language) was developed in the late 1950s by computer manufacturers, the U.S. government and industrial computer users based on a language developed by Grace Hopper, a career U.S. Navy officer and computer scientist. COBOL is still widely used for commercial applications that require precise and efficient manipulation of large amounts of data. Its latest version supports object-oriented programming.
Pascal	Research in the 1960s resulted in *structured programming*—a disciplined approach to writing programs that are clearer, easier to test and debug and easier to modify than large programs produced with previous techniques. One of the more tangible results of this research was the development of Pascal by Professor Niklaus Wirth in 1971. It was designed for teaching structured programming and was popular in college courses for several decades.
Ada	Ada, based on Pascal, was developed under the sponsorship of the U.S. Department of Defense (DOD) during the 1970s and early 1980s. The DOD wanted a single language that would fill most of its needs. The Pascal-based language was named after Lady Ada Lovelace, daughter of the poet Lord Byron. She's credited with writing the world's first computer program in the early 1800s (for the Analytical Engine mechanical computing device designed by Charles Babbage). Its latest version supports object-oriented programming.
Basic	Basic was developed in the 1960s at Dartmouth College to introduce novices to programming. Many of its latest versions are object oriented.
C	C was implemented in 1972 by Dennis Ritchie at Bell Laboratories. It initially became widely known as the UNIX operating system's development language. Today, most of the code for general-purpose operating systems is written in C or C++.

C++	C++, an extension of C, was developed by Bjarne Stroustrup in the early 1980s at Bell Laboratories. C++ provides a number of features that "spruce up" the C language, but more important, it provides capabilities for object-oriented programming.
Objective-C	Objective-C is an object-oriented language based on C. It was developed in the early 1980s and later acquired by Next, which in turn was acquired by Apple. It has become the key programming language for the Mac OS X operating system and all iOS-powered devices (such as iPods, iPhones and iPads).
Visual Basic	Microsoft's Visual Basic language was introduced in the early 1990s to simplify the development of Microsoft Windows applications. Its latest versions support object-oriented programming.
Visual C#	Microsoft's three primary object-oriented programming languages are Visual Basic, Visual C++ (based on C++) and C# (based on C++ and Java, and developed for integrating the Internet and the web into computer applications).
PHP	PHP is an object-oriented, "open-source" (see Section 1.7) "scripting" language supported by a community of users and developers and is used by numerous websites including Wikipedia and Facebook. PHP is platform independent—implementations exist for all major UNIX, Linux, Mac and Windows operating systems. PHP also supports many databases, including MySQL.
Python	Python, another object-oriented scripting language, was released publicly in 1991. Developed by Guido van Rossum of the National Research Institute for Mathematics and Computer Science in Amsterdam (CWI), Python draws heavily from Modula-3—a systems programming language. Python is "extensible"—it can be extended through classes and programming interfaces.
JavaScript	JavaScript is the most widely used scripting language. It's primarily used to add programmability to web pages—for example, animations and interactivity with the user. It's provided with all major web browsers.
Ruby on Rails	Ruby—created in the mid-1990s by Yukihiro Matsumoto—is an open-source, object-oriented programming language with a simple syntax that's similar to Python. Ruby on Rails combines the scripting language Ruby with the Rails web application framework developed by 37Signals. Their book, *Getting Real* (`gettingreal.37signals.com/toc.php`), is a must read for web developers. Many Ruby on Rails developers have reported productivity gains over other languages when developing database-intensive web applications. Ruby on Rails was used to build Twitter's user interface.
Scala	Scala (`www.scala-lang.org/node/273`)—short for "scalable language"—was designed by Martin Odersky, a professor at École Polytechnique Fédérale de Lausanne (EPFL) in Switzerland. Released in 2003, Scala uses both the object-oriented and functional programming paradigms and is designed to integrate with Java. Programming in Scala can reduce the amount of code in your applications significantly. Twitter and Foursquare use Scala.

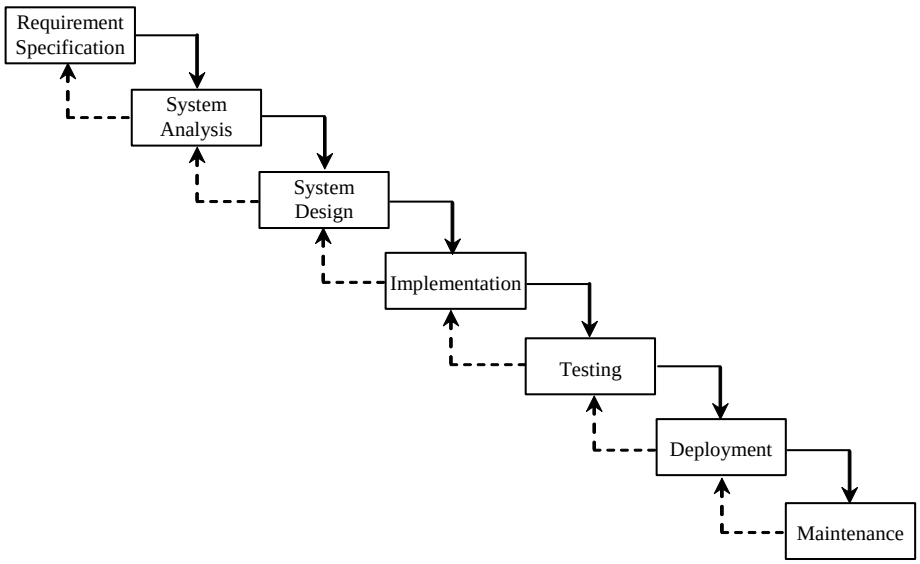
1.2. Software Development

- Software development is a complex process that is both an art and a science.

- It is an art in that it requires a good deal of imagination, creativity, and ingenuity.
- It is also a science in that it uses certain standard techniques and methodologies.

- The term software engineering/development has come to be applied to the study and use of these techniques.
- The software development process is involved certain steps.

- **Requirement Specification:** A formal process that seeks to understand the problem and document in detail what the software system needs to do. This phase involves close interaction between users and designers.
- **System Analysis:** This phase analyzes the business process in terms of data flow, and to identify the system's input and output. Part of the analysis entails modeling the system's behavior. The model is intended to capture the essential elements of the system and to define services to the system.
- **System Design:** This phase involves the use of many levels of abstraction to decompose the problem into manageable components, identify classes and interfaces, and establish relationships among the classes and interfaces.
- **Implementation:** The process of translating the system design into programs. Separate programs are written for each component and put to work together. This phase requires the use of a programming language like Java. The implementation involves coding, testing, and debugging.
- **Testing:** Ensures that the code meets the requirements specification and weeds out bugs.
- **Deployment:** Deployment makes the project available for use. For a Java applet, this means installing it on a Web server; for a Java application, installing it on the client's computer.
- **Maintenance:** Maintenance is concerned with changing and improving the product. A software product must continue to perform and improve in a changing environment. This requires periodic upgrades of the product to fix newly discovered bugs and incorporate changes.



Introduction to JAVA Language

- Java is one of the world's most widely used computer programming language.
- 1991 - Microprocessors are having a profound impact in intelligent consumer-electronic devices. Recognizing this, Sun Microsystems funded an internal corporate research project, which resulted in a C++-based object-oriented programming language Sun named it Java.
- 1993 - The web exploded in popularity, Sun saw the potential of using Java to add dynamic content to web pages. Java garnered the attention of the business community because of the phenomenal interest in the web.
- 2009 - Oracle acquired Sun Microsystems in 2009.

1.2.1. Characteristics of Java

- Java is one of the world's most widely used computer programming language. It is
- **Simple:** Java is partially modeled on C++, but greatly simplified and improved. Some people refer to Java as "C++--" because it is like C++ but with more functionality and fewer negative aspects.
- **Object-Oriented:** Java is inherently object-oriented. Although many object-oriented languages began strictly as procedural languages, Java was designed from the start to be object-oriented. Object-oriented programming (OOP) is a popular programming approach that is replacing traditional procedural programming techniques. One of the central issues in software development is how to reuse code. Object-oriented programming provides great flexibility, modularity, clarity, and reusability through encapsulation, inheritance, and polymorphism.
- **Distributed:** Distributed computing involves several computers working together on a network. Java is designed to make distributed computing easy. Since networking capability is inherently integrated into Java, writing network programs is like sending and receiving data to and from a file.
- **Interpreted:** You need an interpreter to run Java programs. The programs are compiled into the Java Virtual Machine code called bytecode. The bytecode is machine-independent and can run on any machine that has a Java interpreter, which is part of the Java Virtual Machine (JVM).
- **Robust:** Java compilers can detect many problems that would first show up at execution time in other languages. Java has eliminated certain types of error-prone programming constructs found in other languages. Java has a runtime exception-handling feature to provide programming support for robustness.
- **Secure:** Java implements several security mechanisms to protect your system against harm caused by stray programs.
- **Architecture-Neutral:** Write once, run anywhere. With a Java Virtual Machine (JVM), you can write one program that will run on any platform.
- **Portable:** Because Java is architecture neutral, Java programs are portable. They can be run on any platform without being recompiled.
- **Performance:** Java's performance Because Java is architecture neutral, Java programs are portable. They can be run on any platform without being recompiled.
- **Multithreaded**: Multithread programming is smoothly integrated in Java, whereas in other languages you have to call procedures specific to the operating system to enable multithreading.
- **Dynamic:** Java was designed to adapt to an evolving environment. New code can be loaded on the fly without recompilation. There is no need for developers to create, and for users to install, major new software versions. New features can be incorporated transparently as needed.

1.3.2 Java Editions

- **Java Standard Edition:** Used for developing cross-platform, general-purpose applications.
- **Java Enterprise Edition:** Geared toward developing large-scale, distributed networking applications and web-based applications.
- **Java Micro Edition: G**eared toward developing applications for small, memory-constrained devices, such as smartphones, tablets (small, lightweight mobile computers with touch screens) and e-readers
- Java Development Kit (JDK) Versions

  JDK 1.02 (1995)
  JDK 1.1 (1996)
  JDK 1.2 (1998)
  JDK 1.3 (2000)
  JDK 1.4 (2002)
  JDK 1.5 (2004) a. k. a. JDK 5 or Java 5

JDK 1.6 (2006) a. k. a. JDK 6 or Java 6
JDK 1.7 (2011) a. k. a. JDK 7 or Java 7

1.3.3 A Typical Java Development Environment

- Java programs normally go through five phases
  1. Edit
  2. compile
  3. load
  4. verify
  5. execute
- **Phase 1:** editing a file
  - Type a Java program (source code) using an **editor**.
  - Make any necessary corrections.
  - Save the program.
  - A file name ending with the .java extension indicates that the file contains Java source code.

- Common windows editors:
  - Notepad
  - EditPlus (www.editplus.com)
  - TextPad (www.textpad.com)
  - jEdit (www.jedit.org).
- Integrated development environments (IDEs)
  - Provide tools that support the software development process, including editors for writing and editing programs and debuggers for locating logic errors—errors that cause programs to execute incorrectly.
- Popular IDEs
  - Eclipse (www.eclipse.org)
  - NetBeans (www.netbeans.org).
  - jGRASP™ IDE (www.jgrasp.org)
  - DrJava IDE (www.drjava.org/download.shtml)
  - BlueJ IDE (www.bluej.org/)
  - TextPad® Text Editor for Windows® (www.textpad.com/)

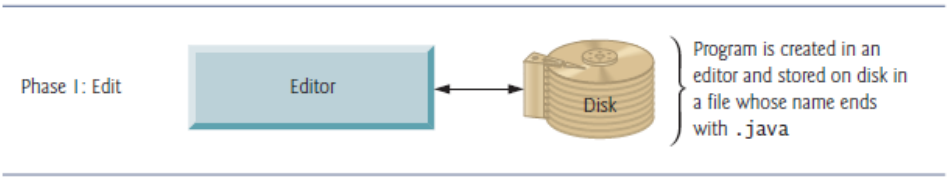


Fig. 1.6 | Typical Java development environment—editing phase.

- **Phase 2:** Compiling a Java Program into Bytecodes
  - Use the command javac (the Java compiler) to compile a program. For example, to compile a program called Welcome.java, you'd type javac Welcome.java
  - If the program compiles, the compiler produces a .class file called Welcome.class that contains the compiled version of the program.
  - Java compiler translates Java source code into bytecodes that represent the tasks to execute.
  - **Bytecodes** are executed by the **Java Virtual Machine (JVM)**

- Virtual machine (VM) is a software application that simulates a computer and hides the underlying operating system and hardware from the programs that interact with it.
- Bytecodes are platform independent. They do not depend on a particular hardware platform.
- Bytecodes are portable. The same bytecodes can execute on any platform containing a JVM that understands the version of Java in which the bytecodes were compiled.
- The JVM is invoked by the java command. For example, to execute a Java application called Welcome, you'd type the command java Welcome

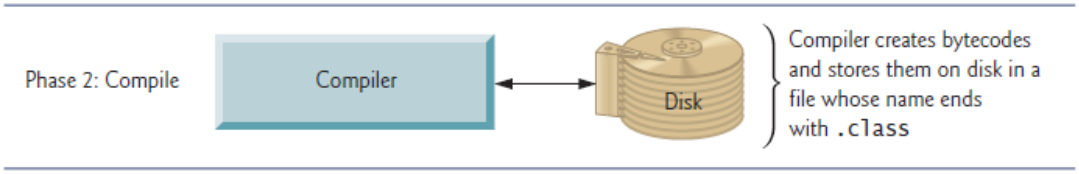


Fig. 1.7 | Typical Java development environment—compilation phase.

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

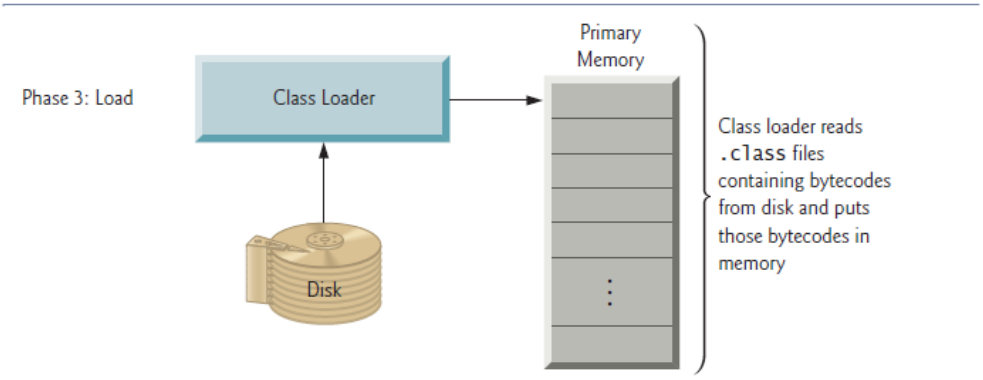


Fig. 1.8 | Typical Java development environment—loading phase.

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

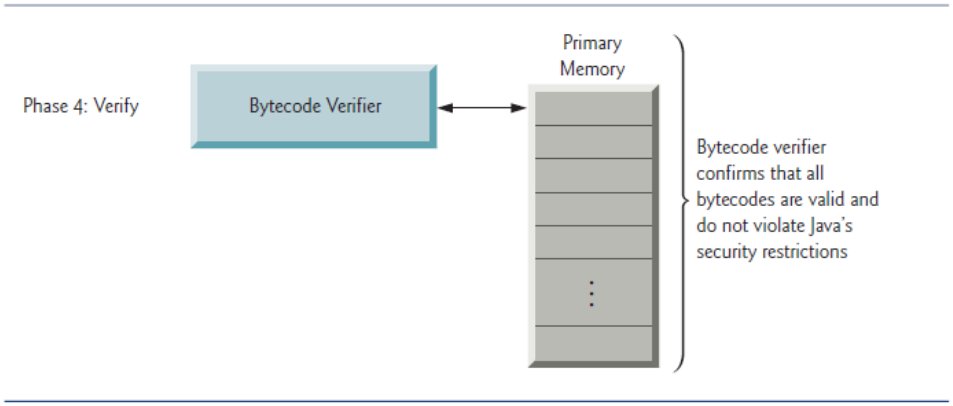


Fig. 1.9 | Typical Java development environment—verification phase.

- Phase 5: Execution
  - The JVM executes the program's bytecodes.
  - JVMs typically execute bytecodes using a combination of interpretation and so-called **just-in-time** (JIT) compilation.
  - Analyzes the bytecodes as they're interpreted
  - A just-in-time (JIT) compiler—known as the Java HotSpot compiler—translates the bytecodes into the underlying computer's machine language.
  - When the JVM encounters these compiled parts again, the faster machine-language code executes.
  - Java programs go through *two compilation phases*
  - One in which source code is translated into bytecodes (for portability across JVMs on different computer platforms) and
  - A second in which, during execution, the bytecodes are translated into machine language for the actual computer on which the program executes.

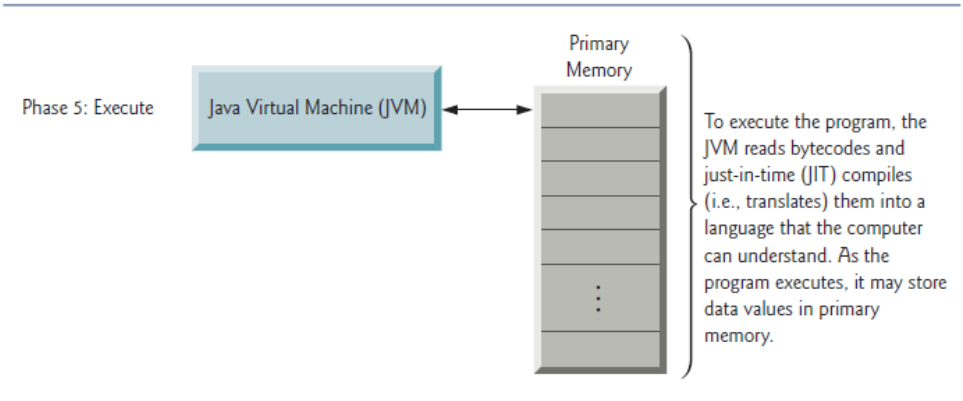


Fig. 1.10 | Typical Java development environment—execution phase.

2. Java Applications

2.1 Our First Program in Java

- Figure 2.1 shows the program that prints a line of text followed by a box that displays its output.

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

```
Welcome to Java Programming!
```

Fig. 2.1 | Text-printing program.

2.2 Discussion about Fig. 2.1

Java Keywords

abstract	assert	boolean	break	byte
case	catch	char	class	continue
default	do	double	else	enum
extends	final	finally	float	for
if	implements	import	instanceof	int
interface	long	native	new	package
private	protected	public	return	short
static	strictfp	super	switch	synchronized
this	throw	throws	transient	try
void	volatile	while		

- Comments

  // Fig. 2.1: Welcome1.java
  - // indicates that the line is a comment.
  - Used to document programs and improve their readability.
  - Compiler ignores comments.
  - A comment that begins with // is an end-of-line comment—it terminates at the end of the line on which it appears.
  - Traditional comment, can be spread over several lines as in /* This is a traditional comment. It can be split over multiple lines */ . This type of comment begins with /* and ends with */. All text between the delimiters is ignored by the compiler.
- Blank lines and space characters
  - Make programs easier to read.
  - Blank lines, spaces and tabs are known as **white space**.

- White space is ignored by the compiler.

- Class declaration

  public class Welcome1

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

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

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

- Declaring the main Method

  public static void main(String[] args)

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

- Statement

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

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

- System.out object
  - Standard output object.
  - Allows Java applications to display strings in the command window from which the Java application executes.

- System.out.println method
  - Displays (or prints) a line of text in the command window.
  - The string in the parentheses the argument to the method.
  - Positions the output cursor at the beginning of the next line in the command window.

- Most statements end with a semicolon (;).

2.3 Compiling and Executing Your First Java Application

- Open a command window and change to the directory where the program is stored.
- Many operating systems use the command cd to change directories.

- To compile the program, type
  javac Welcome1.java
- If the program contains no syntax errors, preceding command creates a.class file (known as the class file) containing the platform-independent Java bytecodes that represent the application.
- When we use the java command to execute the application on a given platform, these bytecodes will be translated by the JVM into instructions that are understood by the underlying operating system.
- To execute the program, type
  java Welcome1
- Launches the JVM, which loads the .class file for class Welcome1.
- Note that the .class file-name extension is omitted from the preceding command; otherwise, the JVM will not execute the program.
- The JVM calls method main to execute the program.

2.4 Modified Your First Java Program (Modification -1)

```
1  // Fig. 2.3: Welcome2.java
2  // Printing a line of text with multiple statements.
3
4  public class Welcome2
5  {
6     // main method begins execution of Java application
7     public static void main( String[] args )
8     {
9        System.out.print( "Welcome to " );
10       System.out.println( "Java Programming!" );
11    } // end method main
12 } // end class Welcome2
```

Prints `Welcome to` and leaves cursor on same line

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

```
Welcome to Java Programming!
```

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

2.5 Discussion about Fig. 2.3

- Class Welcome2, shown in Fig. 2.3, uses two statements to produce the same output as that shown in Fig. 2.1.
- New and key features in each code listing are highlighted.
- System.out's method print displays a string.
- Unlike println, print does not position the output cursor at the beginning of the next line in the command window. The next character the program displays will appear immediately after the last character that print displays
- Newline characters indicate to System.out's print and println methods when to position the output cursor at the beginning of the next line in the command window.
- Newline characters are white-space characters.
- The backslash (\) is called an escape character.
- Indicates a "special character"
- Backslash is combined with the next character to form an escape sequence.
- The escape sequence \n represents the newline character.

```
1  // Fig. 2.4: Welcome3.java
2  // Printing multiple lines of text with a single statement.
3
4  public class Welcome3
5  {
6     // main method begins execution of Java application
7     public static void main( String[] args )
8     {
9        System.out.println( "Welcome\nto\nJava\nProgramming!" );
10    } // end method main
11 } // end class Welcome3
```

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

```
Welcome
to
Java
Programming!
```

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

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

Fig. 2.5 | Some common escape sequences.

2.6 Modified Your First Java Program using printf function (Modification -2)

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

```
1  // Fig. 2.6: Welcome4.java
2  // Displaying multiple lines with method System.out.printf.
3
4  public class Welcome4
5  {
6     // main method begins execution of Java application
7     public static void main( String[] args )
8     {
9        System.out.printf( "%s\n%s\n",
10          "Welcome to", "Java Programming!" );
11    } // end method main
12 } // end class Welcome4
```

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

Statements can be split over multiple lines.

```
Welcome to
Java Programming!
```

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

2.7 Another Application: Adding Integers

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

Imports class Scanner for use in this program

Creates Scanner for reading data from the user

Variables that are declared but not initialized

Reads an int value from the user

Reads another int value from the user

Sums the values of number1 and number2

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

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

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

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

2.8 Discussion about Fig. 2.7

JAVA Programming Part-I

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

- Variable declaration statement

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

  Specifies the name (input) and type (Scanner) of a variable that is used in this program.
- **Variable**
  - A location in the computer's memory where a value can be stored for use later in a program.
  - Must be declared with a **name** and a **type** before they can be used.
  - A variable's name enables the program to access the value of the variable in memory.
  - The name can be any valid identifier.
  - A variable's type specifies what kind of information is stored at that location in memory.
  - Java requires all variables to have a type.
  - Java is a strongly typed language.
  - Primitive types in Java are portable across all platforms.
  - Instance variables of types char, byte, short, int, long, float and double are all given the value 0 by default. Instance variables of type boolean are given the value false by default.
  - Reference-type instance variables are initialized by default to the value null.

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

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

- Scanner object translates these bytes into types that can be used in a program.
- Variable declaration statements

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

  declare that variables number1, number2 and sum hold data of type int
- Several variables of the same type may be declared in one declaration with the variable names separated by commas.

```
System.out.print( "Enter first integer: " ); // prompt
```

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

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

  - Obtains an integer from the user at the keyboard.
  - Program waits for the user to type the number and press the Enter key to submit the number to the program.
  - Use the methods next(), nextByte(), nextShort(), nextInt(), nextLong(), nextFloat(), nextDouble(), or nextBoolean() to obtain to a string, byte, short, int, long, float, double, or boolean value.
- The result of the call to method **nextInt** is placed in variable number1 by using the assignment operator, =.
  - "number1 gets the value of **input.nextInt()**."
  - Operator = is called a binary operator—it has two operands.
  - Everything to the right of the assignment operator, =, is always evaluated before the assignment is performed.
- Arithmetic

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

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

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

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

2.9 Arithmetic

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

- Arithmetic expressions in Java must be written in straight-line form to facilitate entering programs into the computer.
- Expressions such as “a divided by b” must be written as a / b, so that all constants, variables and operators appear in a straight line.
- Parentheses are used to group terms in expressions in the same manner as in algebraic expressions.
- If an expression contains nested parentheses, the expression in the innermost set of parentheses is evaluated first.
- Rules of operator precedence
    - Multiplication, division and remainder operations are applied first.
    - If an expression contains several such operations, they are applied from left to right.
    - Multiplication, division and remainder operators have the same level of precedence.
    - Addition and subtraction operations are applied next.
    - If an expression contains several such operations, the operators are applied from left to right.
    - Addition and subtraction operators have the same level of precedence.
- When we say that operators are applied from left to right, we are referring to their associativity.
- Some operators associate from right to left.
- As in algebra, it’s acceptable to place redundant parentheses (unnecessary parentheses) in an expression to make the expression clearer.
- Condition
- An expression that can be true or false.
- Equality operators (== and !=)
- Relational operators (>, <, >= and <=)
- Both equality operators have the same level of precedence, which is lower than that of the relational operators.
- The equality operators associate from left to right.
- The relational operators all have the same level of precedence and also associate from left to right.

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

Fig. 2.11 | Arithmetic operators.

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

Fig. 2.12 | Precedence of arithmetic operators.

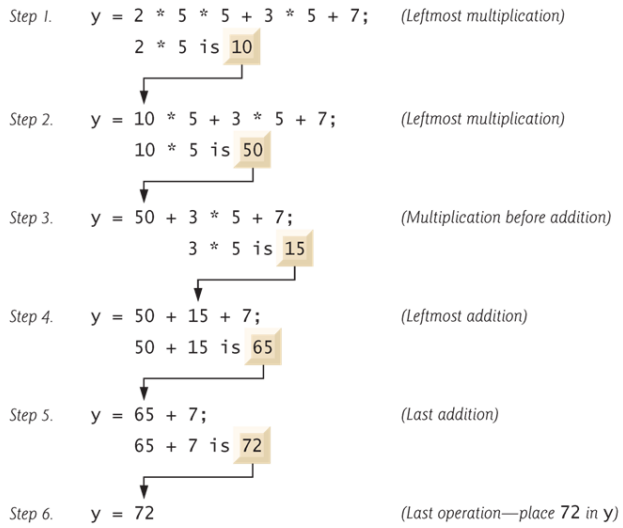


Fig. 2.13 | Order in which a second-degree polynomial is evaluated.

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

Fig. 2.14 | Equality and relational operators.

3. Control Statements: Part I

3.1 Algorithms

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

3.2 Pseudocode

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

3.3 Control Structures

- **Sequential execution**: Statements in a program execute one after the other in the order in which they are written.
- **Transfer of control**: Various Java statements, enable you to specify that the next statement to execute is not necessarily the next one in sequence.
- Bohm and Jacopini
  - Demonstrated that programs could be written without any goto statement.
  - All programs can be written in terms of only three control structures—the sequence structure, the selection structure and the repetition structure.
- Every program is formed by combining the sequence statement, selection statements (three types) and repetition statements (three types) as appropriate for the algorithm the program implements.

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

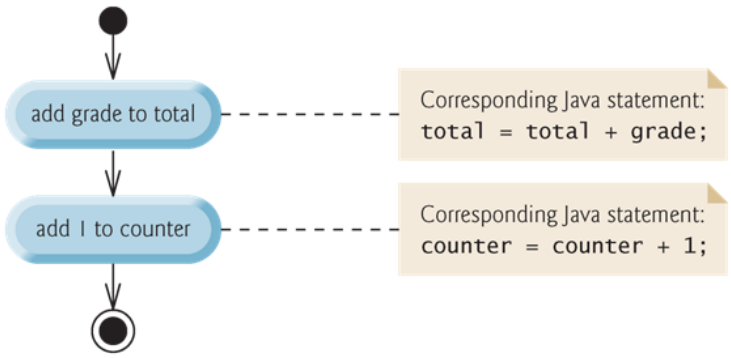


Fig. 4.1 | Sequence structure activity diagram.

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

- **Arrows** represent transitions (order in which the actions represented by the action states occur).
- **Solid circle** at the top represents the initial state—the beginning of the workflow before the program performs the modeled actions.
- **Solid circle surrounded by a hollow circle** at the bottom represents the final state—the end of the workflow after the program performs its actions.

- **UML notes**
  - Like comments in Java.
  - Rectangles with the upper-right corners folded over.
  - Dotted line connects each note with the element it describes.
  - Activity diagrams normally do not show the Java code that implements the activity. We do this here to illustrate how the diagram relates to Java code.

3.4 Selection Structure

- Three types of selection statements.
- **if statement:**
- **if…else statement:**
- **switch statement**
  - Performs one of several actions, based on the value of an expression.
  - Multiple-selection statement—selects among many different actions (or groups of actions).

3.4.1 if - Single-Selection Statement

- Performs an action, if a condition is true; skips it, if false.
- Single-selection statement—selects or ignores a single action (or group of actions).
- Pseudocode

  If student's grade is greater than or equal to 60
  Print "Passed"
- If the condition is false, the Print statement is ignored, and the next pseudocode statement in order is performed.
- Indentation is optional, but recommended. it emphasizes the inherent structure of structured programs
- The preceding pseudocode *if* in Java:

```
if ( studentGrade >= 60 )
        System.out.println( "Passed" );
```

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

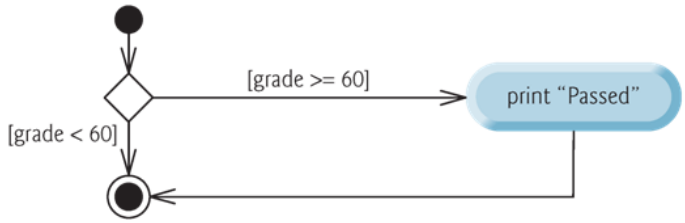


Fig. 4.2 | `if` single-selection statement UML activity diagram.

3.4.2 if…else - Double-Selection Statement

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

  If student's grade is greater than or equal to 60
  Print "Passed"
  Else
  Print "Failed"
- The preceding *If…Else pseudocode statement in Java:*

```
if ( grade >= 60 )
  System.out.println( "Passed" );
else
  System.out.println( "Failed" );
```

- Note that the body of the else is also indented.
- Figure 4.3 illustrates the flow of control in the if…else statement.

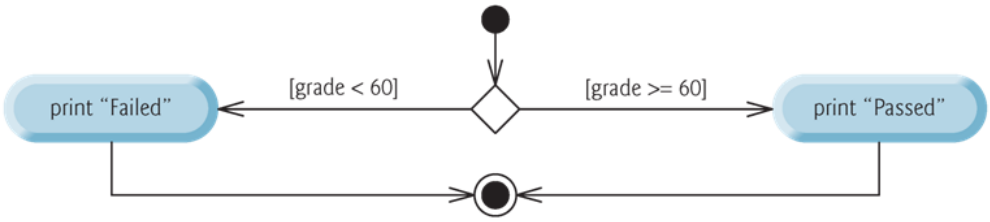


Fig. 4.3 | `if…else` double-selection statement UML activity diagram.

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

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

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

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

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

- This pseudocode may be written in Java as

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

- If studentGrade >= 90, the first four conditions will be true, but only the statement in the if part of the first if…else statement will execute. After that, the else part of the "outermost" if…else statement is skipped.
- The Java compiler always associates an else with the immediately preceding if unless told to do otherwise by the placement of braces ({ and }).
- Referred to as the dangling-else problem.
- The following code is not what it appears:

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

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

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

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

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

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

3.5 Repetition structure

- Three repetition statements **while, for, do… while** (also called looping statements)
  - Perform statements repeatedly while a loop-continuation condition remains true.
- while and for statements perform the action(s) in their bodies zero or more times
  - if the loop-continuation condition is initially false, the body will not execute.
- The do…while statement performs the action(s) in its body one or more times.

3.5.1 while Repetition Statement

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

```
While there are more items on my shopping list
        Purchase next item and cross it off my list
```

- The repetition statement's body may be a single statement or a block.
- Eventually, the condition will become false. At this point, the repetition terminates, and the first statement after the repetition statement executes.
- Example of Java's while repetition statement: find the first power of 3 larger than 100. Assume int variable product is initialized to 3.

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

- Each iteration multiplies product by 3, so product takes on the values 9, 27, 81 and 243 successively.
- When variable product becomes 243, the while-statement condition—product <= 100—becomes false.
- Repetition terminates. The final value of product is 243.
- Program execution continues with the next statement after the while statement.
- The UML activity diagram in Fig. 4.4 illustrates the flow of control in the preceding while statement.
- The decision and merge symbols can be distinguished by the number of "incoming" and "outgoing" transition arrows.
  - A decision symbol has one transition arrow pointing to the diamond and two or more pointing out from it to indicate possible transitions from that point. Each transition arrow pointing out of a decision symbol has a guard condition next to it.

- A merge symbol has two or more transition arrows pointing to the diamond and only one pointing from the diamond, to indicate multiple activity flows merging to continue the activity. None of the transition arrows associated with a merge symbol has a guard condition.

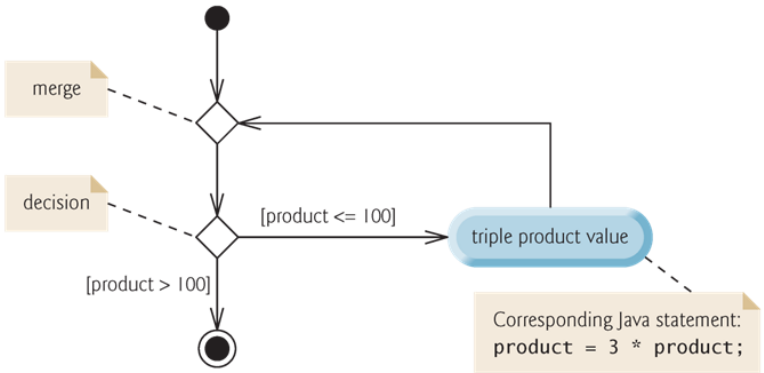


Fig. 4.4 | `while` repetition statement UML activity diagram.

3.5.2 Type of repetitions

3.5.2.1 Example

- *Problem Statement*
  - *A class of ten students took a quiz. The grades (integers in the range 0 to 100) for this quiz are available to you. Determine the class average on the quiz.*
  - We solve this problem by implementing java class GradeBook.
- Pseudocode Algorithm with Counter-Controlled Repetition in Class GradeBook

```
1  Set total to zero
2  Set grade counter to one
3
4  While grade counter is less than or equal to ten
5     Prompt the user to enter the next grade
6     Input the next grade
7     Add the grade into the total
8     Add one to the grade counter
9
10 Set the class average to the total divided by ten
11 Print the class average
```

Fig. 4.5 | Pseudocode algorithm that uses counter-controlled repetition to solve the class-average problem.

- Implementing Counter-Controlled Repetition in Class GradeBook

```
1  // Fig. 4.6: GradeBook.java
2  // GradeBook class that solves class-average problem using
3  // counter-controlled repetition.
4  import java.util.Scanner; // program uses class Scanner
5
6  public class GradeBook
7  {
8     private String courseName; // name of course this GradeBook represents
9
10    // constructor initializes courseName
11    public GradeBook( String name )
12    {
13       courseName = name; // initializes courseName
14    } // end constructor
15
16    // method to set the course name
17    public void setCourseName( String name )
18    {
19       courseName = name; // store the course name
20    } // end method setCourseName
21
```

Fig. 4.6 | GradeBook class that solves class-average problem using counter-controlled repetition. (Part 1 of 3.)

```
22    // method to retrieve the course name
23    public String getCourseName()
24    {
25       return courseName;
26    } // end method getCourseName
27
28    // display a welcome message to the GradeBook user
29    public void displayMessage()
30    {
31       // getCourseName gets the name of the course
32       System.out.printf( "Welcome to the grade book for\n%s!\n\n",
33          getCourseName() );
34    } // end method displayMessage
35
36    // determine class average based on 10 grades entered by user
37    public void determineClassAverage()
38    {
39       // create Scanner to obtain input from command window
40       Scanner input = new Scanner( System.in );
41
42       int total; // sum of grades entered by user
43       int gradeCounter; // number of the grade to be entered next
```

Fig. 4.6 | GradeBook class that solves class-average problem using counter-controlled repetition. (Part 2 of 3.)

```
44         int grade; // grade value entered by user
45         int average; // average of grades
46
47         // initialization phase
48         total = 0; // initialize total
49         gradeCounter = 1; // initialize loop counter
50
51         // processing phase uses counter-controlled repetition
52         while ( gradeCounter <= 10 ) // loop 10 times
53         {
54            System.out.print( "Enter grade: " ); // prompt
55            grade = input.nextInt(); // input next grade
56            total = total + grade; // add grade to total
57            gradeCounter = gradeCounter + 1; // increment counter by 1
58         } // end while
59
60         // termination phase
61         average = total / 10; // integer division yields integer result
62
63         // display total and average of grades
64         System.out.printf( "\nTotal of all 10 grades is %d\n", total );
65         System.out.printf( "Class average is %d\n", average );
66      } // end method determineClassAverage
67   } // end class GradeBook
```

Fig. 4.6 | GradeBook class that solves class-average problem using counter-controlled repetition. (Part 3 of 3.)

- Class GradeBookTest for testing class GradeBook

```
1  // Fig. 4.7: GradeBookTest.java
2  // Create GradeBook object and invoke its determineClassAverage method.
3
4  public class GradeBookTest
5  {
6     public static void main( String[] args )
7     {
8        // create GradeBook object myGradeBook and
9        // pass course name to constructor
10       GradeBook myGradeBook = new GradeBook(
11          "CS101 Introduction to Java Programming" );
12
13       myGradeBook.displayMessage(); // display welcome message
14       myGradeBook.determineClassAverage(); // find average of 10 grades
15    } // end main
16 } // end class GradeBookTest
```

Fig. 4.7 | GradeBookTest class creates an object of class GradeBook (Fig. 4.6) and invokes its determineClassAverage method. (Part 1 of 2.)

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

Enter grade: 67
Enter grade: 78
Enter grade: 89
Enter grade: 67
Enter grade: 87
Enter grade: 98
Enter grade: 93
Enter grade: 85
Enter grade: 82
Enter grade: 100

Total of all 10 grades is 846
Class average is 84
```

Fig. 4.7 | `GradeBookTest` class creates an object of class `GradeBook` (Fig. 4.6) and invokes its `determineClassAverage` method. (Part 2 of 2.)

3.5.2.2 Discussion on implementation of GradeBook class

- Variables declared in a method body are local variables and can be used only from the line of their declaration to the closing right brace of the method declaration.
- A local variable's declaration must appear before the variable is used in that method.
- A local variable cannot be accessed outside the method in which it's declared.
- The program's output indicates that the sum of the grade values in the sample execution is 846, which, when divided by 10, should yield the floating-point number 84.6.
- The result of the calculation total / 10 (line 61 of Fig. 4.6) is the integer 84, because total and 10 are both integers.
- Dividing two integers results in integer division—any fractional part of the calculation is lost (i.e., truncated).

```
1  // Fig. 5.1: WhileCounter.java
2  // Counter-controlled repetition with the while repetition statement.
3
4  public class WhileCounter
5  {
6     public static void main( String[] args )
7     {
8        int counter = 1; // declare and initialize control variable
9
10       while ( counter <= 10 ) // loop-continuation condition
11       {
12          System.out.printf( "%d  ", counter );
13          ++counter; // increment control variable by 1
14       } // end while
15
16       System.out.println(); // output a newline
17    } // end main
18 } // end class WhileCounter
```

Declares and initializes control variable `counter` to 1

Loop-continuation condition tests for `count`'s final value

Initializes `gradeCounter` to 1; indicates first grade about to be input

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

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

3.5.2.3 Counter-controlled repetition summary

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

3.5.3 Sentinel-Controlled Repetition

- Sentinel-controlled repetition is often called indefinite repetition because the number of repetitions is not known before the loop begins executing.
- A special value called a sentinel value (also called a signal value, a dummy value or a flag value) can be used to indicate "end of data entry."
- A sentinel value must be chosen that cannot be confused with an acceptable input value.

3.5.3.1 Example

- Problem Statement
  - *Develop a class-averaging program that processes grades for an arbitrary number of students each time it is run.*
- Pseudocode Algorithm with Sentinel-Controlled Repetition in Class GradeBook

```
1  Initialize total to zero
2  Initialize counter to zero
3
4  Prompt the user to enter the first grade
5  Input the first grade (possibly the sentinel)
6
7  While the user has not yet entered the sentinel
8      Add this grade into the running total
9      Add one to the grade counter
10     Prompt the user to enter the next grade
11     Input the next grade (possibly the sentinel)
12
13 If the counter is not equal to zero
14     Set the average to the total divided by the counter
15     Print the average
16 else
17     Print "No grades were entered"
```

Fig. 4.8 | Class-average problem pseudocode algorithm with sentinel-controlled repetition.

- Implementation of Class GradeBook using sentinel-controlled repetition

```
1  // Fig. 4.9: GradeBook.java
2  // GradeBook class that solves the class-average problem using
3  // sentinel-controlled repetition.
4  import java.util.Scanner; // program uses class Scanner
5
6  public class GradeBook
7  {
8     private String courseName; // name of course this GradeBook represents
9
10    // constructor initializes courseName
11    public GradeBook( String name )
12    {
13       courseName = name; // initializes courseName
14    } // end constructor
15
16    // method to set the course name
17    public void setCourseName( String name )
18    {
19       courseName = name; // store the course name
20    } // end method setCourseName
21
```

Fig. 4.9 | GradeBook class that solves the class-average problem using sentinel-controlled repetition. (Part 1 of 4.)

```
22    // method to retrieve the course name
23    public String getCourseName()
24    {
25       return courseName;
26    } // end method getCourseName
27
28    // display a welcome message to the GradeBook user
29    public void displayMessage()
30    {
31       // getCourseName gets the name of the course
32       System.out.printf( "Welcome to the grade book for\n%s!\n\n",
33          getCourseName() );
34    } // end method displayMessage
35
36    // determine the average of an arbitrary number of grades
37    public void determineClassAverage()
38    {
39       // create Scanner to obtain input from command window
40       Scanner input = new Scanner( System.in );
41
42       int total; // sum of grades
43       int gradeCounter; // number of grades entered
```

Fig. 4.9 | GradeBook class that solves the class-average problem using sentinel-controlled repetition. (Part 2 of 4.)

```
44          int grade; // grade value
45          double average; // number with decimal point for average
46
47          // initialization phase
48          total = 0; // initialize total
49          gradeCounter = 0; // initialize loop counter
50
51          // processing phase
52          // prompt for input and read grade from user
53          System.out.print( "Enter grade or -1 to quit: " );
54          grade = input.nextInt();
55
56          // loop until sentinel value read from user
57          while ( grade != -1 )
58          {
59             total = total + grade; // add grade to total
60             gradeCounter = gradeCounter + 1; // increment counter
61
62             // prompt for input and read next grade from user
63             System.out.print( "Enter grade or -1 to quit: " );
64             grade = input.nextInt();
65          } // end while
```

Fig. 4.9 | GradeBook class that solves the class-average problem using sentinel-controlled repetition. (Part 3 of 4.)

```
66
67          // termination phase
68          // if user entered at least one grade...
69          if ( gradeCounter != 0 )
70          {
71             // calculate average of all grades entered
72             average = (double) total / gradeCounter;
73
74             // display total and average (with two digits of precision)
75             System.out.printf( "\nTotal of the %d grades entered is %d\n",
76                gradeCounter, total );
77             System.out.printf( "Class average is %.2f\n", average );
78          } // end if
79          else // no grades were entered, so output appropriate message
80             System.out.println( "No grades were entered" );
81       } // end method determineClassAverage
82    } // end class GradeBook
```

Fig. 4.9 | GradeBook class that solves the class-average problem using sentinel-controlled repetition. (Part 4 of 4.)

- Class GradeBookTest

```
1  // Fig. 4.10: GradeBookTest.java
2  // Create GradeBook object and invoke its determineClassAverage method.
3
4  public class GradeBookTest
5  {
6     public static void main( String[] args )
7     {
8        // create GradeBook object myGradeBook and
9        // pass course name to constructor
10       GradeBook myGradeBook = new GradeBook(
11          "CS101 Introduction to Java Programming" );
12
13       myGradeBook.displayMessage(); // display welcome message
14       myGradeBook.determineClassAverage(); // find average of grades
15    } // end main
16 } // end class GradeBookTest
```

Fig. 4.10 | `GradeBookTest` class creates an object of class `GradeBook` (Fig. 4.9) and invokes its `determineClassAverage` method. (Part 1 of 2.)

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

Enter grade or -1 to quit: 97
Enter grade or -1 to quit: 88
Enter grade or -1 to quit: 72
Enter grade or -1 to quit: -1

Total of the 3 grades entered is 257
Class average is 85.67
```

Fig. 4.10 | `GradeBookTest` class creates an object of class `GradeBook` (Fig. 4.9) and invokes its `determineClassAverage` method. (Part 2 of 2.)

3.5.3.2 Discussion

- Program logic for sentinel-controlled repetition
  - Reads the first value before reaching the while.
  - This value determines whether the program's flow of control should enter the body of the while. If the condition of the while is false, the user entered the sentinel value, so the body of the while does not execute (i.e., no grades were entered).
  - If the condition is true, the body begins execution and processes the input.
  - Then the loop body inputs the next value from the user before the end of the loop.
- Integer division yields an integer result.
- To perform a floating-point calculation with integers, temporarily treat these values as floating-point numbers for use in the calculation.
- The unary cast operator (double) creates a temporary floating-point copy of its operand.
- Cast operator performs explicit conversion (or type cast).
- The value stored in the operand is unchanged.
- Java evaluates only arithmetic expressions in which the operands' types are identical.
- Promotion (or implicit conversion) performed on operands.
- In an expression containing values of the types int and double, the int values are promoted to double values for use in the expression.
- Cast operators are available for any type.

- Cast operator formed by placing parentheses around the name of a type.
- The operator is a unary operator (i.e., an operator that takes only one operand).
- Java also supports unary versions of the plus (+) and minus (–) operators.
- Cast operators associate from right to left; same precedence as other unary operators, such as unary + and unary -.
- This precedence is one level higher than that of the multiplicative operators *, / and %.
- Appendix A: Operator precedence chart

3.5.4 Examination Results Problem - Nested Control Statements

- This case study examines nesting one control statement within another.
- A college offers a course that prepares students for the state licensing exam for real estate brokers. Last year, ten of the students who completed this course took the exam. The college wants to know how well its students did on the exam. You've been asked to write a program to summarize the results. You've been given a list of these 10 students. Next to each name is written a 1 if the student passed the exam or a 2 if the student failed.
- Your program should analyze the results of the exam as follows:
  - Input each test result (i.e., a 1 or a 2). Display the message "Enter result" on the screen each time the program requests another test result.
  - Count the number of test results of each type.
  - Display a summary of the test results, indicating the number of students who passed and the number who failed.
  - If more than eight students passed the exam, print the message "Bonus to instructor!"

- Pseudocode Algorithm for examination-results problem

```
1  Initialize passes to zero
2  Initialize failures to zero
3  Initialize student counter to one
4
5  While student counter is less than or equal to 10
6      Prompt the user to enter the next exam result
7      Input the next exam result
8
9      If the student passed
10         Add one to passes
11     Else
12         Add one to failures
13
14     Add one to student counter
15
16 Print the number of passes
17 Print the number of failures
18
19 If more than eight students passed
20     Print "Bonus to instructor!"
```

Fig. 4.11 | Pseudocode for examination-results problem.

- Implementation of examination-results problem

```
1  // Fig. 4.12: Analysis.java
2  // Analysis of examination results using nested control statements.
3  import java.util.Scanner; // class uses class Scanner
4
5  public class Analysis
6  {
7     public static void main( String[] args )
8     {
9        // create Scanner to obtain input from command window
10       Scanner input = new Scanner( System.in );
11
12       // initializing variables in declarations
13       int passes = 0; // number of passes
14       int failures = 0; // number of failures
15       int studentCounter = 1; // student counter
16       int result; // one exam result (obtains value from user)
17
```

Fig. 4.12 | Analysis of examination results using nested control statements. (Part 1 of 4.)

```
18       // process 10 students using counter-controlled loop
19       while ( studentCounter <= 10 )
20       {
21          // prompt user for input and obtain value from user
22          System.out.print( "Enter result (1 = pass, 2 = fail): " );
23          result = input.nextInt();
24
25          // if...else is nested in the while statement
26          if ( result == 1 )          // if result 1,
27             passes = passes + 1;     // increment passes;
28          else                        // else result is not 1, so
29             failures = failures + 1; // increment failures
30
31          // increment studentCounter so loop eventually terminates
32          studentCounter = studentCounter + 1;
33       } // end while
34
```

Fig. 4.12 | Analysis of examination results using nested control statements. (Part 2 of 4.)

```
35         // termination phase; prepare and display results
36         System.out.printf( "Passed: %d\nFailed: %d\n", passes, failures );
37
38         // determine whether more than 8 students passed
39         if ( passes > 8 )
40            System.out.println( "Bonus to instructor!" );
41      } // end main
42   } // end class Analysis
```

```
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Passed: 9
Failed: 1
Bonus to instructor!
```

Fig. 4.12 | Analysis of examination results using nested control statements. (Part 3 of 4.)

```
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Passed: 6
Failed: 4
```

Fig. 4.12 | Analysis of examination results using nested control statements. (Part 4 of 4.)

3.6 Compound Assignment Operators

- Compound assignment operators abbreviate assignment expressions.
- Statements like

  variable = variable operator expression;

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

  variable operator= expression;
- Example:

  c = c + 3;

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

c += 3;

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

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

Fig. 4.13 | Arithmetic compound assignment operators.

3.6.1 Increment and Decrement Operators

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

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

Fig. 4.14 | Increment and decrement operators.

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

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

Uses current value, then increments c

Increments c then uses new value

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

Control Statements: Part II

4.1 for Repetition Statement

- for **repetition statement** specifies the counter-controlled-repetition details in a single line of code.

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

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

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

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

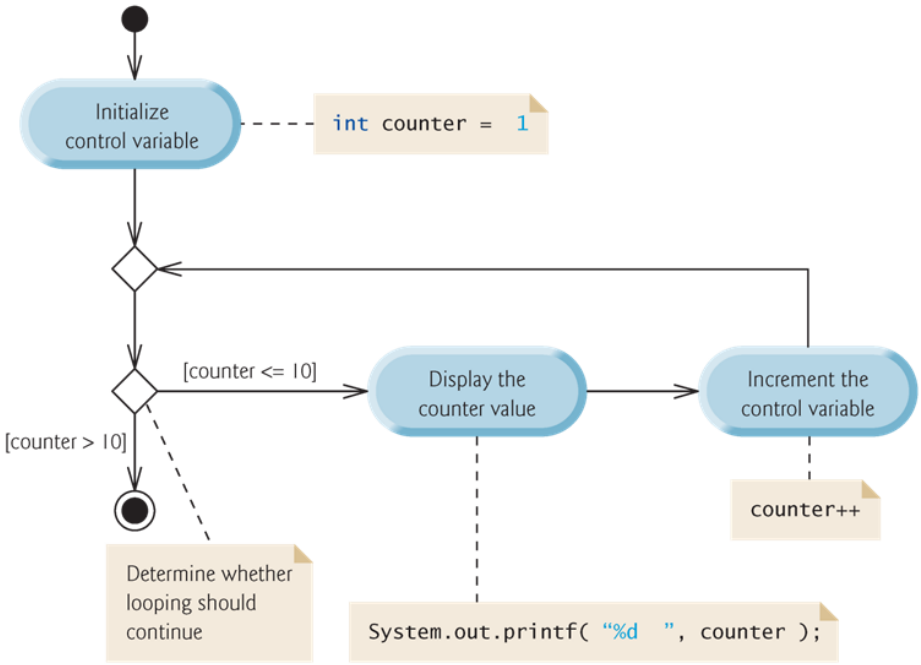


Fig. 5.4 | UML activity diagram for the `for` statement in Fig. 5.2.

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

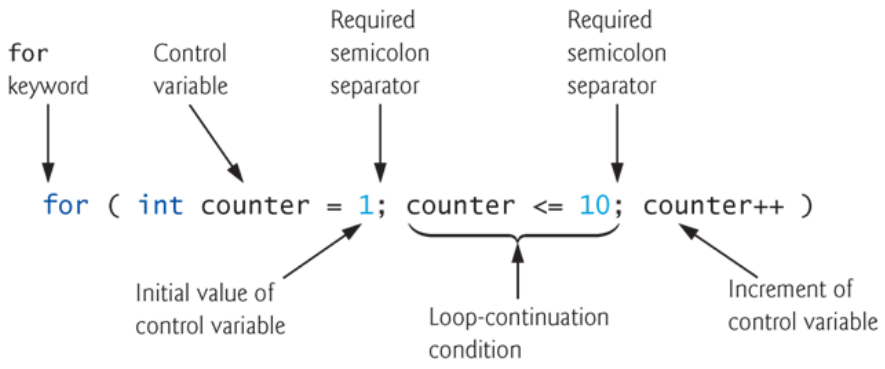


Fig. 5.3 | `for` statement header components.

- The general format of the for statement is

for (*initialization; loopContinuationCondition; increment)*
statement

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

- The two semicolons in the for header are required.
- In most cases, the for statement can be represented with an equivalent while statement as follows:

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

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

```
counter = counter + 1
counter += 1
++counter
counter++
```

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

```
for (int j = x; j <= 4 * x * y; j += y / x)
```

  is equivalent to the statement

```
for (int j = 2; j <= 80; j += 5)
```

- The increment of a for statement may be negative, in which case it's a decrement, and the loop counts downward.
- **Examples**
  a) Vary the control variable from 1 to 100 in increments of 1.
     `for ( int i = 1; i <= 100; i++ )`
  b) Vary the control variable from 100 to 1 in decrements of 1.
     `for ( int i = 100; i >= 1; i-- )`
  c) Vary the control variable from 7 to 77 in increments of 7.
     `for ( int i = 7; i <= 77; i += 7 )`
  d) Vary the control variable from 20 to 2 in decrements of 2.

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

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

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

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

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

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

```
Sum is 110
```

Fig. 5.5 | Summing integers with the `for` statement.

4.1.1 Problem Example: Compound interest application

- Problem Statement

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

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

  where

  p is the original amount invested (i.e., the principal)
  r is the annual interest rate (e.g., use 0.05 for 5%)
  n is the number of years
  a is the amount on deposit at the end of the nth year.

- Implementation

```
1  // Fig. 5.6: Interest.java
2  // Compound-interest calculations with for.
3
4  public class Interest
5  {
6     public static void main( String[] args )
7     {
8        double amount; // amount on deposit at end of each year
9        double principal = 1000.0; // initial amount before interest
10       double rate = 0.05; // interest rate
11
12       // display headers
13       System.out.printf( "%s%20s\n", "Year", "Amount on deposit" );
14
15       // calculate amount on deposit for each of ten years
16       for ( int year = 1; year <= 10; year++ )
17       {
18          // calculate new amount for specified year
19          amount = principal * Math.pow( 1.0 + rate, year );
20
```

Java treats floating-point literals as `double` values

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

Fig. 5.6 | Compound-interest calculations with `for`. (Part 1 of 2.)

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

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

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

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

4.1.1.1 Discussion

- Java treats floating-point constants like 1000.0 and 0.05 as type double.
- Java treats whole-number constants like 7 and -22 as type int.
- In the format specifier %20s, the integer 20 between the % and the conversion character s indicates that the value output should be displayed with a **field width** of 20—that is, printf displays the value with at least 20 character positions.
- If the value to be output is less than 20 character positions wide, the value is **right justified** in the field by default.
- If the year value to be output were more thanhas more characters than the field width, the field width would be extended to the right to accommodate the entire value.
- To indicate that values should be output **left justified**, precede the field width with the **minus sign (–) formatting flag** (e.g., %-20s).
- Classes provide methods that perform common tasks on objects.
- Most methods must be called on a specific object.

- Many classes also provide methods that perform common tasks and do not require objects. These are called static methods.
- Java does not include an exponentiation operator—Math class static method pow can be used for raising a value to a power.
- You can call a static method by specifying the class name followed by a dot (.) and the method name, as in

  ClassName.methodName(arguments)
- Math.pow(*x*, *y*) calculates the value of x raised to the y^{th} power. The method receives two double arguments and returns a double value.
- In the format specifier %,20.2f, the **comma (,) formatting flag** indicates that the floating-point value should be output with a **grouping separator.**
- Separator is specific to the user's locale (i.e., country).
- In the United States, the number will be output using commas to separate every three digits and a decimal point to separate the fractional part of the number, as in 1,234.45.
- The number 20 in the format specification indicates that the value should be output right justified in a field width of 20 characters.
- The .2 specifies the formatted number's precision—in this case, the number is rounded to the nearest hundredth and output with two digits to the right of the decimal point.

4.2 do…while Repetition Statement

- The do…while **repetition statement** is similar to the while statement.
- In the while, the program tests the loop-continuation condition at the beginning of the loop, before executing the loop's body; if the condition is false, the body never executes.
- The do…while statement tests the loop-continuation condition *after executing the loop's body*; therefore, the body always executes at least once.
- When a do…while statement terminates, execution continues with the next statement in sequence.
- Figure 5.8 contains the UML activity diagram for the do…while statement.
- The diagram makes it clear that the loop-continuation condition is not evaluated until after the loop performs the action state at least once.
- Braces are not required in the do…while repetition statement if there's only one statement in the body.
- Most programmers include the braces, to avoid confusion between the while and do…while statements.
- Thus, the do…while statement with one body statement is usually written as follows:

```
do
{
  statement
} while ( condition );
```

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

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

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

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

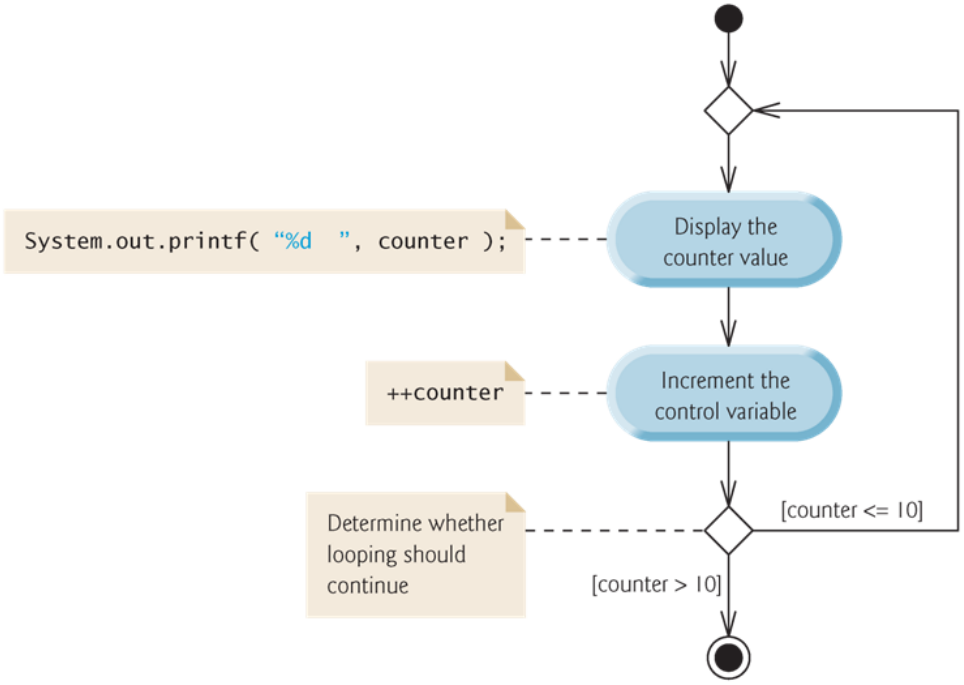


Fig. 5.8 | do...while repetition statement UML activity diagram.

4.3 switch Multiple-Selection Statement

- switch **multiple-selection statement** performs different actions based on the possible values of a **constant integral expression** of type byte, short, int or char.
- Figure 5.11 shows the UML activity diagram for the general switch statement.
- Most switch statements use a break in each case to terminate the switch statement after processing the case.
- The break statement is not required for the switch's last case (or the optional default case, when it appears last), because execution continues with the next statement after the switch.

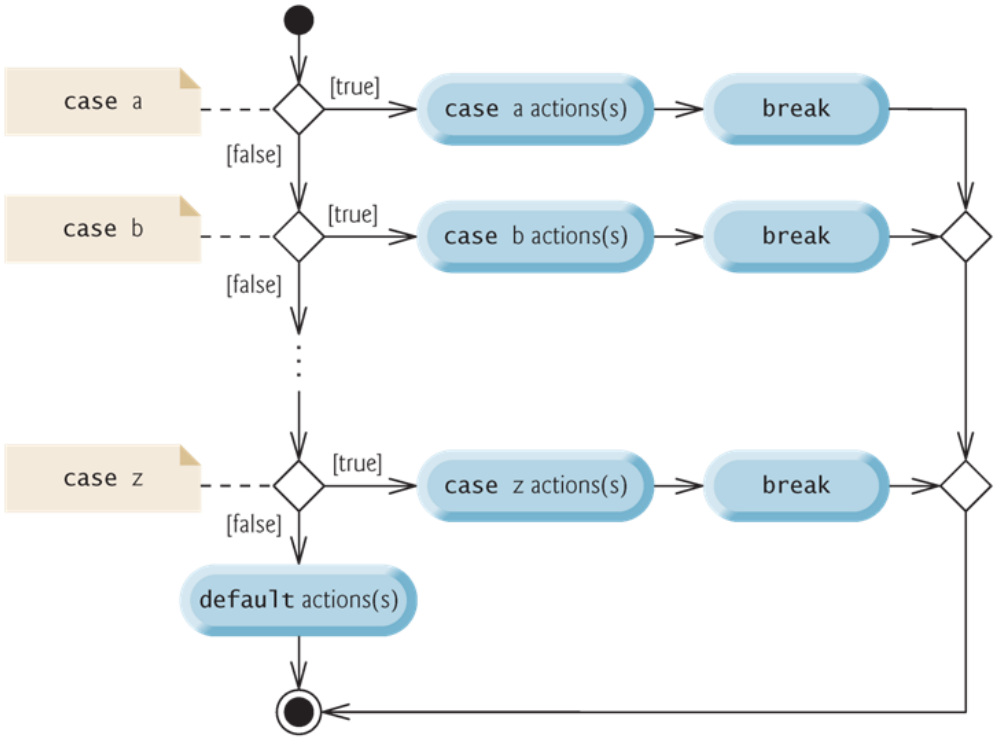


Fig. 5.11 | `switch` multiple-selection statement UML activity diagram with `break` statements.

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

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

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

Fig. 5.9 | `GradeBook` class uses `switch` statement to count letter grades. (Part 1 of 7.)

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

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

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

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

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

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

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

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

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

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

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

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

- Testing class

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

Fig. 5.10 | Create `GradeBook` object, input grades and display grade report. (Part 1 of 3.)

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

Enter the integer grades in the range 0-100.
Type the end-of-file indicator to terminate input:
   On UNIX/Linux/Mac OS X type <Ctrl> d then press Enter
   On Windows type <Ctrl> z then press Enter
99
92
45
57
63
71
76
85
90
100
^Z

Grade Report:
Total of the 10 grades entered is 778
Class average is 77.80
```

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

```
Number of students who received each grade:
A: 4
B: 1
C: 2
D: 1
F: 2
```

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

4.3.1 Discussion

- The **end-of-file indicator** is a system-dependent keystroke combination which the user enters to indicate that there is no more data to input.
- On UNIX/Linux/Mac OS X systems, end-of-file is entered by typing the sequence
  - *<Ctrl> d*
- on a line by itself. This notation means to simultaneously press both the *Ctrl* key and the *d* key.
- On Windows systems, end-of-file can be entered by typing
  - *<Ctrl> z*
- On some systems, you must press *Enter* after typing the end-of-file key sequence.
- Windows typically displays the characters ^Z on the screen when the end-of-file indicator is typed.
- Scanner method hasNext determine whether there is more data to input. This method returns the boolean value true if there is more data; otherwise, it returns false.
- As long as the end-of-file indicator has not been typed, method hasNext will return true.
- The switch statement consists of a block that contains a sequence of case **labels** and an optional default **case.**
- The program evaluates the **controlling expression** in the parentheses following keyword switch.
- The program compares the controlling expression's value (which must evaluate to an integral value of type byte, char, short or int) with each case label.
- If a match occurs, the program executes that case's statements.
- The break **statement** causes program control to proceed with the first statement after the switch.
- switch does not provide a mechanism for testing ranges of values—every value must be listed in a separate case label.
- Note that each case can have multiple statements.

- switch differs from other control statements in that it does not require braces around multiple statements in a case.
- Without break, the statements for a matching case and subsequent cases execute until a break or the end of the switch is encountered. This is called “falling through.”
- If no match occurs between the controlling expression’s value and a case label, the default case executes.
- If no match occurs and there is no default case, program control simply continues with the first statement after the switch.

4.3.2 break and continue Statements

- The break statement, when executed in a while, for, do…while or switch, causes immediate exit from that statement.
- Execution continues with the first statement after the control statement.
- Common uses of the break statement are to escape early from a loop or to skip the remainder of a switch.
- The continue statement, when executed in a while, for or do…while, skips the remaining statements in the loop body and proceeds with the next iteration of the loop.
- In while and do…while statements, the program evaluates the loop-continuation test immediately after the continue statement executes.
- In a for statement, the increment expression executes, then the program evaluates the loop-continuation test.
-

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

Terminates the loop immediately and program control continues at line 17

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

Fig. 5.12 | break statement exiting a for statement.

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

Terminates current iteration of loop and proceeds to increment

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

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

4.4 Logical Operators

- Java's **logical operators** enable you to form more complex conditions by combining simple conditions.
- The logical operators are
  - && (conditional AND)
  - || (conditional OR)
  - & (boolean logical AND)
  - | (boolean logical inclusive OR)
  - ^ (boolean logical exclusive OR)
  - ! (logical NOT).

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

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

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

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

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

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

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

- The **boolean logical AND** (**&**) and **boolean logical inclusive OR (|)** operators are identical to the && and || operators, except that the & and | operators *always evaluate both of their operands* (i.e., they do not perform short-circuit evaluation).
- This is useful if the right operand of the boolean logical AND or boolean logical inclusive OR operator has a required **side effect**—a modification of a variable's value.
- A simple condition containing the **boolean logical exclusive OR** (**^**) operator is true *if and only if* one of its operands is true and the other is false.
- If both are true or both are false, the entire condition is false.
- Figure 5.16 is a truth table for the boolean logical exclusive OR operator (^).
- This operator is guaranteed to evaluate both of its operands.

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

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

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

expression	!expression
false	true
true	false

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

- Summary

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

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

Fig. 5.18 | Logical operators. (Part 1 of 4.)

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

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

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

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

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

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

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

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

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

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

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