COURSE MATERIAL

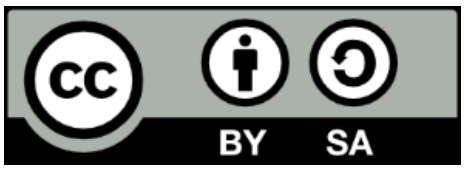
Programming Using Java

National Open University of Nigeria
Faculty of Sciences, Department of Computer Science

Copyright

This course has been developed as part of the collaborative advanced ICT course development project of the Commonwealth of Learning (COL). COL is an intergovernmental organization created by Commonwealth Heads of Government to promote the development and sharing of open learning and distance education knowledge, resources and technologies.

The National Open University of Nigeria (NOUN) is a fully fledged, autonomous and accredited public University. It offers its certificate, diploma, degree and postgraduate courses through the open and distance learning system which includes various means of communication such as face-to-face, broadcasting, telecasting, correspondence, seminars, e-learning as well as a blended mode. The NOUN's academic programmes are quality-assured and centrally regulated by the National Universities Commission (NUC).



National Open University of Nigeria
Plot 91 Cadastral Zone,
University Village,
Jabi - Abuja,
Nigeria
Phone: +234806 310 2206
Email: info@noun.edu.ng
Website: www.noun.edu.ng

Commonwealth of Learning
4710 Kingsway, Suite 2500,
Burnaby V5H 4M2,
British Columbia,
Canada
Phone: +1 604 775 8200
Fax: +1 604 775 8210
Email: info@col.org
Website: www.col.org

Acknowledgements

The National Open University of Nigeria, Faculty of Sciences and the Department of Computer Science wish to thank those below for their contribution to this to the production of this course material and video lectures:

Authors:

Ass. Prof. MuhtarAlhassan (Director, Management Information Systems, NOUN)
Dr. Juliana Ndunagu (Lecturer, Computer Science Department, NOUN)
Mr. Abdul-JalilChobe (Lecturer, Computer Science Department, NOUN)

Copy Editor: Prof. Adekunle A. Adebowale and Dr. Olujimi Alao

Reviewer: Dr. Greg O. Onwodi

Contents

About this COURSE MATERIAL

Programming using Java has been produced by The National Open University of Nigeria. All Course Materials produced by The National Open University of Nigeria are structured in the same way, as outlined below.

The course overview

The course overview gives you a general introduction to the course. Information contained in the course overview will help you determine:

- If the course is suitable for you.
- What you will already need to know.
- What you can expect from the course.
- How much time you will need to invest to complete the course.

The overview also provides guidance on:

- Study skills.
- Where to get help.
- Course assignments and assessments.
- Activity icons.
- Units.

We strongly recommend that you read the overview *carefully* before starting your study.

The course content

The course is broken down into units. Each unit comprises:

- An introduction to the unit content.
- Unit Objectives
- Unit outcomes.
- New terminology.
- Core content of the unit with a variety of learning activities.
- A unit summary.
- Assignments and/or assessments, as applicable.
- Answers to Assignment and/or assessment, as applicable

Resources

For those interested in learning more on this subject, we provide you with a list of additional resources at the end of this course material; these may be books, articles or websites.

Your comments

After completing Programming using Java we would appreciate it if you would take a few moments to give us your feedback on any aspect of this course. Your feedback might include comments on:

- Course content and structure.
- Course reading materials and resources.
- Course assignments.
- Course assessments.
- Course duration.
- Course support (assigned tutors, technical help, etc.)

Your constructive feedback will help us to improve and enhance this course.

Course overview

Welcome to Programming using Java

Java is a general-purpose computer programming language that is concurrent, class-based, object-oriented, and specifically designed to have as few implementation dependencies as possible. It is intended to let application developers "write once, run anywhere" (WORA), meaning that compiled Java code can run on all platforms that support Java without the need for recompilation.

Programming Using Java-is this course for you?

This course is intended for people who desire to earn a living through the use of advance ICT skills.
Candidates of this course must have at least a pass in Mathematics at O' levels.

Course objectives

Objectives

The objectives of this course are:

- Explain Java as a platform consisting of virtual machine and execution environment.
- Understand and explain the different editions of java platform that can be used to create Java programs
- Understand and explain Java Database Connectivity and how to achieve such connectivity.
- Understand the proper installation of JDK and setting up the environment for creating Java programs
- Explain the use of classes and objects and identify how they simply process of creating complex program
- Proper understanding and use of selection, decision and repetition

Course outcomes

Outcomes

Upon completion of Programming Using Java you will be able to:

- Explain Java as a platform consisting of virtual machine and execution environment.
- Understand and explain the different editions of java platform that can be used to create Java programs
- Understand and explain Java Database Connectivity and how to achieve such connectivity.
- Understand the proper installation of JDK and setting up the environment for creating Java programs
- Explain the use of classes and objects and identify how

they simply process of creating complex program

- Proper understanding and use of selection, decision and repetition
- Understand and explain the importance of IDE and how it is a preliminary requirement for any mobile applications
- Understand how to download, install and run any required IDE for mobile application development. Emphasis would be on Eclipse, SDK and Android studio.
- Proper understanding and use of software development kit, android and iOS.

Timeframe

How long?

The expected duration of this course is eight weeks
The course should be lectured at least 2 hours per week
2 hours weekly of self study time is recommended

Need help?

This course is offered at The National Open University of Nigeria, Computer Science Department.
If you need help regarding this course, please contact:

Dr. Greg Onwodi
Head of Department, Computer Science department
The National Open University of Nigeria
Plot 91 Cadastral Zone,
University Village,
Jabi - Abuja,
Nigeria
Phone: +2347032022265
Email: gonwodi@noun.edu.ng
Website: www.noun.edu.ng

Assessments

Assessments

There are activities, case studies, assignments and review questions in the units of this course. All these learner's activities are assessed in three modalities

- Peer – review
- Self – assessment
- Instructor – marked assessment

NB: Review questions are for self – assessment

Unit 1

Introduction

Java is as an object oriented programming language that was designed to meet the need for a platform independent language. Java programming language was originally developed by Sun Microsystems which was initiated by James Gosling and released in 1995 as core component of Sun Microsystems' Java platform. In order to be used to handle various requirements of Java platform; three editions of Java have been organized by Sun Microsystems. These editions are: Java SE, Java EE and Java ME. Java is guaranteed to be Write Once, Run Anywhere.

Outcomes

Upon completion of this unit you will be able to:

- At the end of this unit, you should be able to:
- Understand the concepts of Java programming
- Understand the need for Java as a platform independent language
- Understand the different editions of Java
- Understand Java databases

Terminology

Object Oriented Programming Language(OOP**):**	OOP refers to a type of computer programming (software design) in which programmers define not only the data type of a data structure, but also the types of operations (functions) that can be applied to the data structure.
Platform:	The underlying hardware or software for a system. The platform defines a standard around which a system can be developed.
Bytecode:	Programming code that, once compiled, is run through a virtual machine instead of the computer's processor. By using this approach, source code can be run on any platform once it has been compiled and run through the virtual machine.
Compiler:	Compiler is a program that translates source code into object code. The compiler looks at the entire piece of source code and collects and reorganizes the instructions.
JVM:	Java Virtual Machine, a platform-independent execution environment that converts Java bytecode into machine language and executes it.

Sourcecode:	Program instructions in their original form. The word source differentiates code from various other forms that it can have.

Getting Started with Java

Java as a Platform Independent Language

Most programming languages are platform dependent. Applications developed by using such programming languages can run only on those types of hardware and software platforms on which the applications are compiled. Java is a platform independent language that enables you to compile an application on one platform and execute it on any platform. This saves your effort to write and compile the same application for different platforms. Java programs are saved with an extension, .java. A .java file is compiled to generate the .class file, which contains the Bytecode. The JVM converts the Bytecode contained in the .class file to machine object code. The JVM needs to be implemented for each platform running on a different operating system.

Figure 1.1 shows the relationship among various components of the Java programming environment

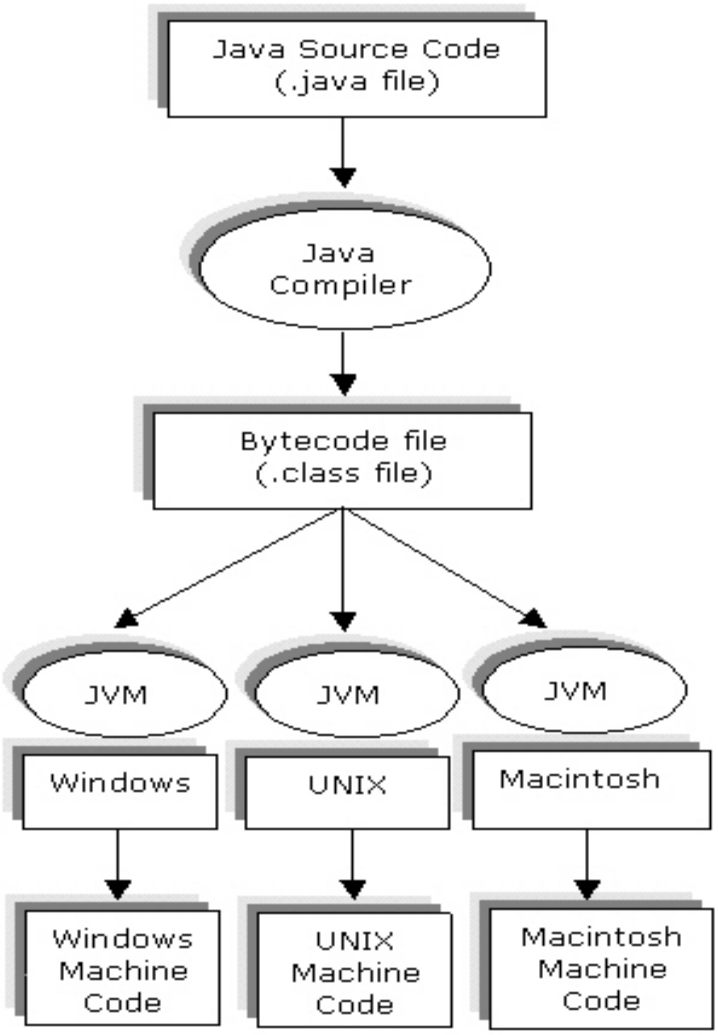


Figure 1.1: The Java compiler translates Java source code into Java object code consisting of bytecode and associated data

The JVM forms the base for the Java platform and is convenient to use on various hardware-based platforms. JVM for different platforms uses different techniques to execute the Bytecode. The major components of JVM are:

- Class loader: loads the class files, which are required by a program running in the memory.
- Execution engine: converts the Bytecode to the machine object code and runs it.
- Just In Time (JIT) compiler: compiles the Bytecode into executable code.

A Java program executes through a tool that will load and start the Java Virtual Machine and it will pass the program's main classfile to the machine. The Java Virtual Machine will use its classloader component to load the classfile into memory.

When a classfile is loaded, the Java Virtual Machine's bytecode verifier component ensures that the classfile'sbytecode is valid and

doesn’t compromise security in any way. The verifier usually terminates the J virtual machine when there is a problem with the bytecode. If it is all well with the classfile’s bytecode, the virtual machine’s interpreter component interprets the bytecode one instruction at a time. While Interpretation consists of identifying bytecode instructions and executing equivalent native instructions.

The moment the interpreter learns that a sequence of bytecode instructions is executed repeatedly, it informs the virtual machine’s just-in-time (JIT) compiler to compile the instructions into native code.

Just-In-Time compilation is done strictly once for a given sequence of bytecode instructions. This is because the native instructions execute instead of the associated bytecode instruction sequence, the program perform the execution very fast.

In the process of execution, interpreter might encounter a request to execute another classfile’sbytecode. In a scenario like that the interpreter asks the classloader to load the classfile and the bytecode verifier to verify the bytecode before executing tthebytecode. Moreover during execution, bytecode instructions might request that the Java Virtual Machine open a file and display on the screen, or request the virtual machine to perform another task that would require the native platform. JVM responds by transferring the request to the platform via its Java Native Interface bridge to the native platform. Figure 1.2 shows an illustration of the Java Virtual Machine tasks.

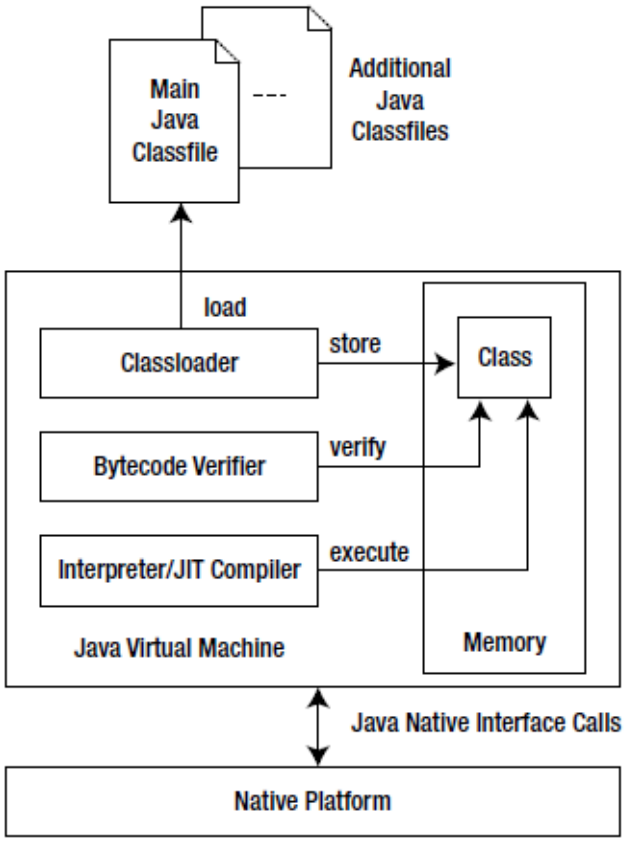


Figure 1.2: The JVM provides all of the necessary components for loading, verifying, and executing a classfile

Java as a platform independent language ensures *portability* whichrefers to the ability of a program to run on any platform without changing the source code of the program. Due to such portability, the same bytecode runs unchanged on Windows, Linux, Mac OS X, and other platforms.

Java platform also ensures *security* by providing a secure environment such as the bytecode verifier in which code executes. The ultimate goal is to prevent malicious code from affecting the underlying platform.

Java SE, Java EE, Java ME

Various developers use different editions of the Java platform to create Java programs that run on desktop computers, web browsers, web servers, mobile information devices and other embedded devices. Below are the different editions of java platform:

Java Platform, Standard Edition (Java SE): This Java platform edition is used for developing applications, which are stand-alone

programs that run on desktop. This edition is also used to develop applets, programs that usually run in web browsers.

Java Platform, Enterprise Edition (Java EE): This is also another Java platform edition for developing enterprise-oriented applications and servlets, these are server programs that conform to Java EE's Servlet application program interface. Java Enterprise Edition is built on top of Java Software Edition.

Java Platform, Micro Edition (Java ME): The Java Micro Edition platform for developing MIDlets, these are programs that run on mobile information devices, and also Xlets, programs that run on embedded devices.

Java Database

Java database was first introduced by Sun Microsystems as part of Java Development Kit 6 to provide developers with a relational database management system RDMS to test Java database connectivity JDBC code. Java database is a distribution of Apache's open-source Derby product, which is based on IBM's Cloudscape RDBMS code base. The Java RDBMS is also combined with JDK 7, It is considered secure, and supports JDBC and SQL, and has a small footprint, its core engine. Java database connectivity driver occupy approximately 2MB.

Java DB is capable of running in an embedded environment or in a client/server environment. In an embedded environment, where an application accesses the database engine via Java DB's *embedded driver*, the database engine runs in the same virtual machine as the application. Figure 1.3 illustrates the embedded environment architecture, where the database engine is embedded in the application.

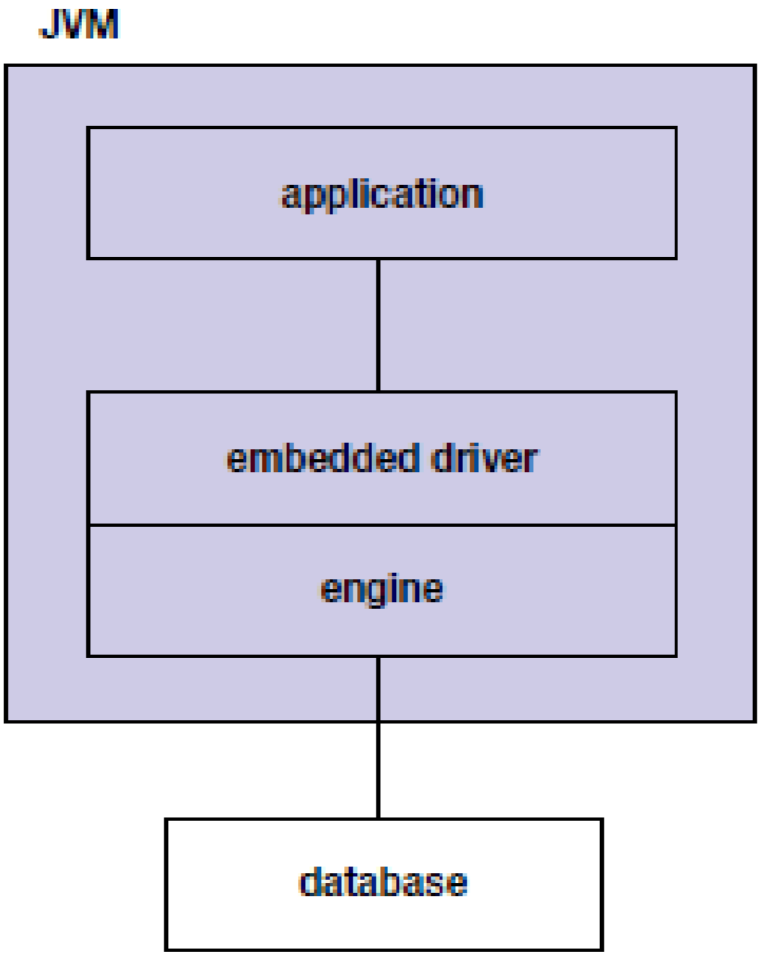


Figure 1.3: No separate processes are required to start up or shut down an embedded database engine.

Client applications and the database engine run in separate virtual machines in a client/server environment. A client application can also access the network server through Java database *client driver*. The network server, which runs in the same virtual machine as the database engine, accesses the database engine through the embedded driver. Figure 1.4 illustrates this architecture.

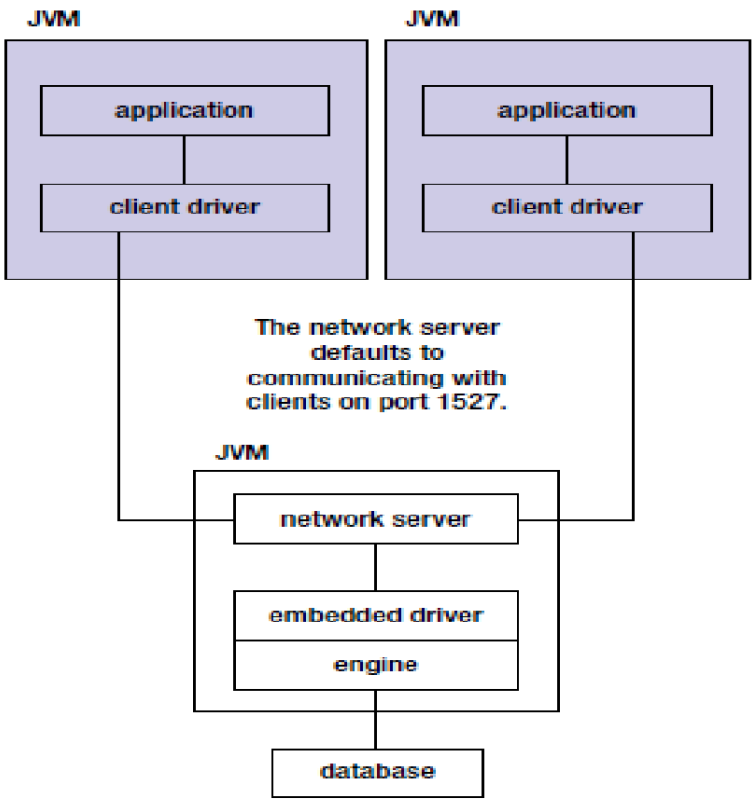


Fig 1.4: Multiple clients communicate with the same database engine through the network server.

Java DB implements the database portion of the architectures shown in Figures 1.3 and 1.4 as a directory with the same name as the database. Within this directory, Java DB creates a log directory to store transaction logs, a seg0 directory to store the data files, and a service.properties file to store configuration parameters.

Java DB Installation and Configuration

When you install JDK 7 with the default settings, the bundled Java DB is installed into %JAVA_HOME%\db on Windows platforms, or into the db subdirectory in the equivalent location on Unix/Linux platforms.

The db directory always contains five files and the below mentioned pair of subdirectories:

- The bin directory contains scripts for setting up embedded and client/server environments, running command-line tools, and starting/stopping the network server. You should add this directory to your PATH environment variable so that you can conveniently execute its scripts from anywhere in the file system.

- The lib directory contains various JAR files that house the engine library (derby.jar), the command-line tools libraries (derbytools.jar and derbyrun.jar), the network server library (derbynet.jar), the network client library (derbyclient.jar), and various locale-specific libraries. This directory also contains derby.war, which is used to register the network *servlet*

(see http://en.wikipedia.org/wiki/Java_Servlet) at the /derbynet relative path—it's also possible to manage the Java DB network server remotely via the servlet interface
(see
http://db.apache.org/derby/docs/10.8/adminguide/cadminservlet98430.html)

Conclusion

Java is object oriented, platform independent, easy to learn and with secure feature which enables you to develop virus-free, tamper-free systems. The development process of Java is more rapid and analytical since the linking is an incremental and light-weight process.

It comes in three editions: Java Standard Edition (Java SE), Java Enterprise Edition (Java EE), and JavaMicro Edition (Java ME). Java SE can be used to develop client-side standalone applications or applets. Java EE can be used to develop server-side applications, such as Java servlets and Java Server Pages. Java ME can be used to develop applications for mobile devices, such as cell phones.

Unit summary

In this unit, you learned that:

- Java is a powerful programming language that is built on object oriented programming language and was designed to meet the need for a platform independent language.
- The JVM converts the bytecode contained in the .class file to machine objectcode.
- Java comes in three editions: Java Standard Edition (Java SE), Java Enterprise Edition (Java EE), and JavaMicro Edition (Java ME).
- Java database is included as part of Java Development Kit 6 to provide developers with a relational database management system RDMS to test Java database connectivity JDBC code.

Assessment

1. Which of the following is correct extension of a Java file?
    a. .jav
    b. .java
    c. .JAVA
    d. .class
2. The Java compiler reads the Java source file and converts it to a file having an extension:
    a. .class
    b. .java
    c. .prg
    d. .file
3. JVM stands for:
    a. Java Virtual Machine
    b. Java Virtual Model
    c. Java Virtual Mechanism
    d. Java Virtual Methodology
4. JIT stands for:
    a. Java In Time
    b. Just In Time
    c. Java Is Tested
    d. Java Is Time saving
5. What is Java?
6. Discuss Java as a platform Independent language.

Videos http://tinyurl.com/y77op5de

http://tinyurl.com/ycqoy2ux

http://tinyurl.com/ydd2lxpl

Unit 2

Installing the Java Development Kit (JDK)

This unit basically focuses on you knowing and understanding your working environment, particularly your system requirements before you set up and development environment. Identifying such requirements will allow you to understand how to set up system for development regardless of the operating system.

Outcomes

Upon completion of this unit you will be able to:

- Understand system requirement for different operating systems
- Understand how to set up a working Java platform and environment.

Terminology

Java ME:	Short for ***Java 2 Platform Micro Edition***. J2ME is Sun Microsystems' answer to a consumer wireless device platform.
Java EE:	J2EE is a platform-independent, Java-centric environment from Sun for developing, building and deploying Web-based enterprise applications online.
RDBMS:	Relational database management system (RDBMS) is a type of database management system (DBMS) that stores data in the form of related tables
JDK:	Short for ***Java Development Kit***, a software development kit (SDK) for producing Java programs. The JDK is developed by Sun Microsystem's JavaSoft division

System Requirement

The software and hardware prerequisites for installing Java Development Kit on a Windows system are as follows:

Microsoft Windows

- Microsoft Windows XP SP3, Vista, Windows Server 2008, Windows 7, Windows 8, Windows Server 2012, or Windows 10
- Pentium-compatible PC (minimum a Pentium 2 266 MHz processor)
- 128 MB RAM (256 MB RAM recommended)
- Disk space: 124 MB for JRE; 2 MB for Java Update

Administrator rights are needed for the installation process. It is a recommended best practice to back up your system and data before you remove or install software.

Apple Mac OS X

- Intel-based Mac running Mac OS X 10.8.3+, 10.9+
- 128 MB RAM (256 MB RAM recommended)
- Disk space: 124 MB for JRE; 2 MB for Java Update

It is a recommended best practice to back-up your system and data before you remove or install software.

Installing the Java Development Kit on Windows

This type of JDK installation is for Windows users. There would be a section for installation of JDK on Mac system. Integrated development environment such as Android Studio uses the Java tool chain to build, so you need to make sure that you have the Java Development Kit (JDK) installed on your computer before you start using IDEs like Android Studio. There is a possibility that you already have the JDK installed on your computer, you might have tried some installations for developing java. If you already have the JDK installed on your computer, and you're running JDK version 1.6 or higher, then you can skip this section. However it is advisable to download, install, and configure the latest JDK anyway. You can download the JDK from the following Oracle website.
www.oracle.com/technetwork/java/javase/downloads/index.html

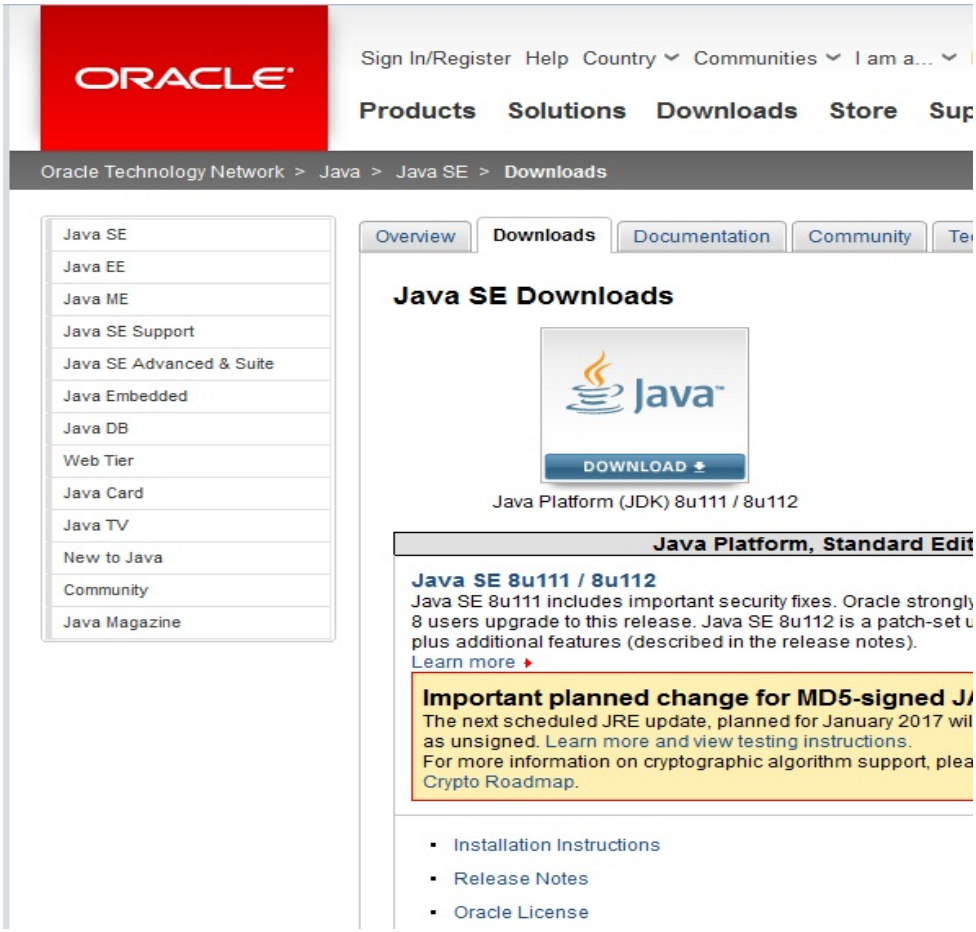


Fig 1.5: Java SE Download Environment

Downloading the JDK on Windows

The next step of installation as shown in Figure 1.6 requires that you accept a license agreement by clicking the Accept License Agreement radio button. Then you must choose the appropriate JDK for your operating system. If your operating system is Windows 7, Windows 8, or Windows 10 click the file link to the right of the Windows x64 label, also shown in Figure 1.6. Oracle makes frequent release updates to the JDK, so it is always important to go for the latest version. Allow the installation file to download.

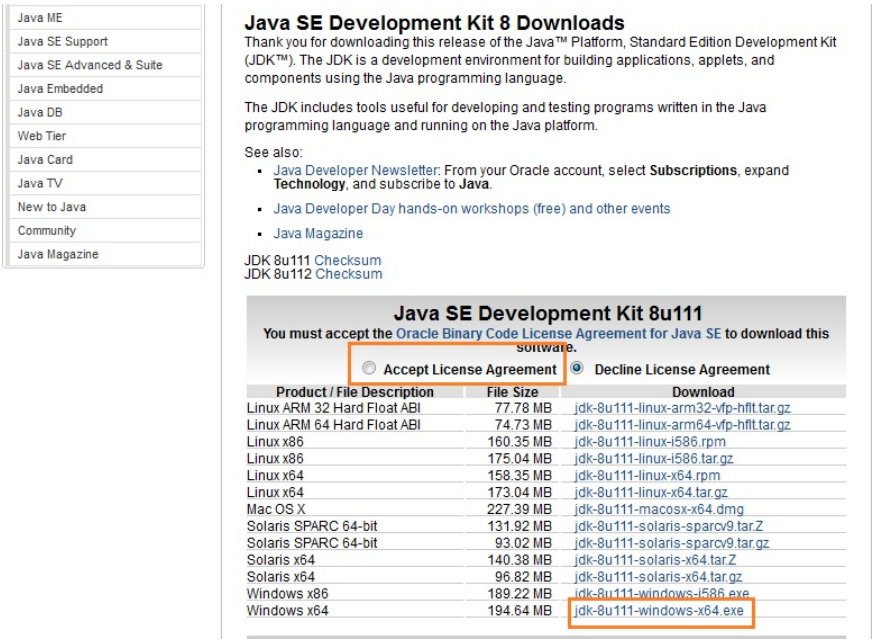


Figure 1.6: Download options for JDK

Running the JDK Wizard on Windows

It is important that before you install the JDK, you create a directory in the root of your C: drive called Java. The name of this directory is arbitrary, though we call it Java because many of the tools we are going to install here are related to Java, including the JDK, Android Studio, and the Android SDK. Consistently installing the tools related to Android Studio in the C:\Java directory also keeps your development environment organized and well identified.

Navigate to the location where your installation file is downloaded and run the file by double-clicking it. Once the installation begins, Installation Wizard will show as shown in Figure 1.7. In Windows, the JDK installer defaults to C:\Program Files\Java\. To change the installation directory location, click the Change button. It is always recommended that your JDK is installed in the C:\Java directory because it contains no spaces in the path name and it's easy to remember. See Figure 1.8.

Figure 1.7: *Installation Wizard for the JDK on Windows*

Figure 1.8: *Select the JDK installation directory*

Take note of where you are installing your JDK. Follow the prompts until the installation is complete. If prompted to install the Java Runtime Edition (JRE), choose the same directory where you installed the JDK.

Configuring Java Development Kit on Windows

This section shows you how to configure Windows so that the JDK is found by Android Studio. On a computer running Windows, hold down the Windows key and press the Pause key to open the System

window. Click the Advanced System Settings option, shown in Figure 1.9.

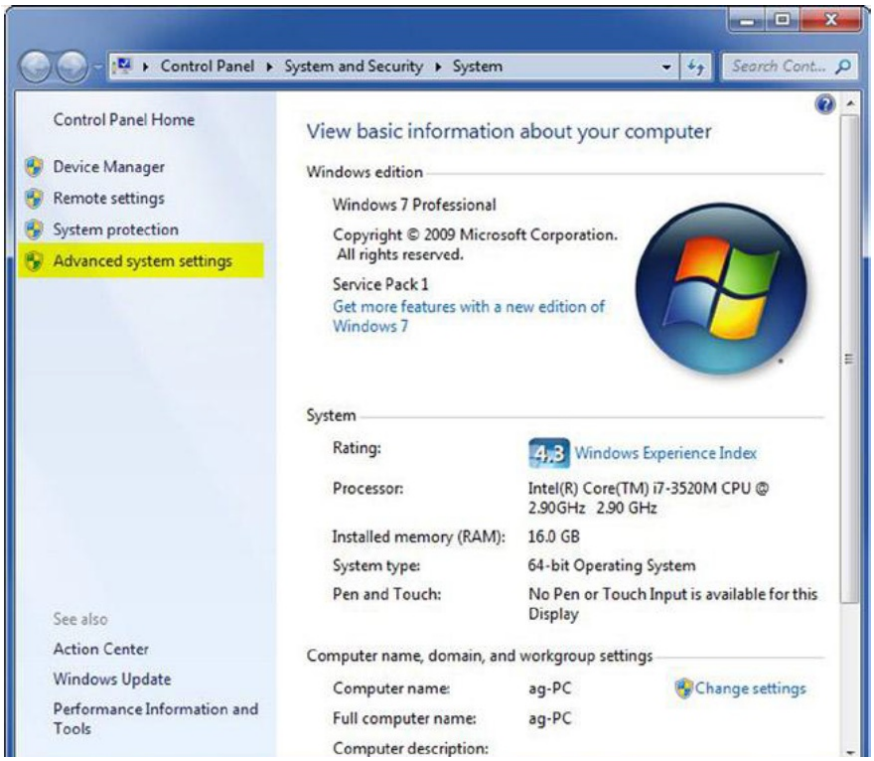


Figure 1.9: *Windows System window*

Click the Environmental Variables button, shown in Figure 1.10. In the System Variables list along the bottom, shown in Figure 1.11, navigate to the JAVA_HOME item. If the JAVA_HOME item does not exist, click New to create it. Otherwise, click Edit.

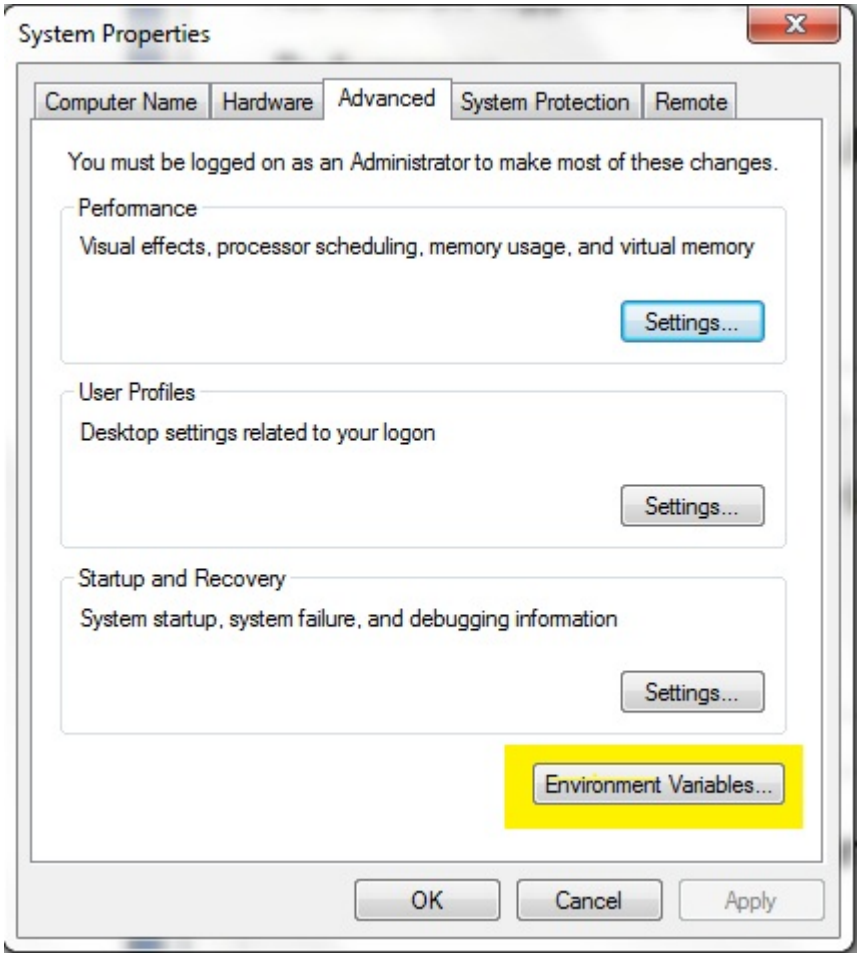


Figure 1.10: *System properties*

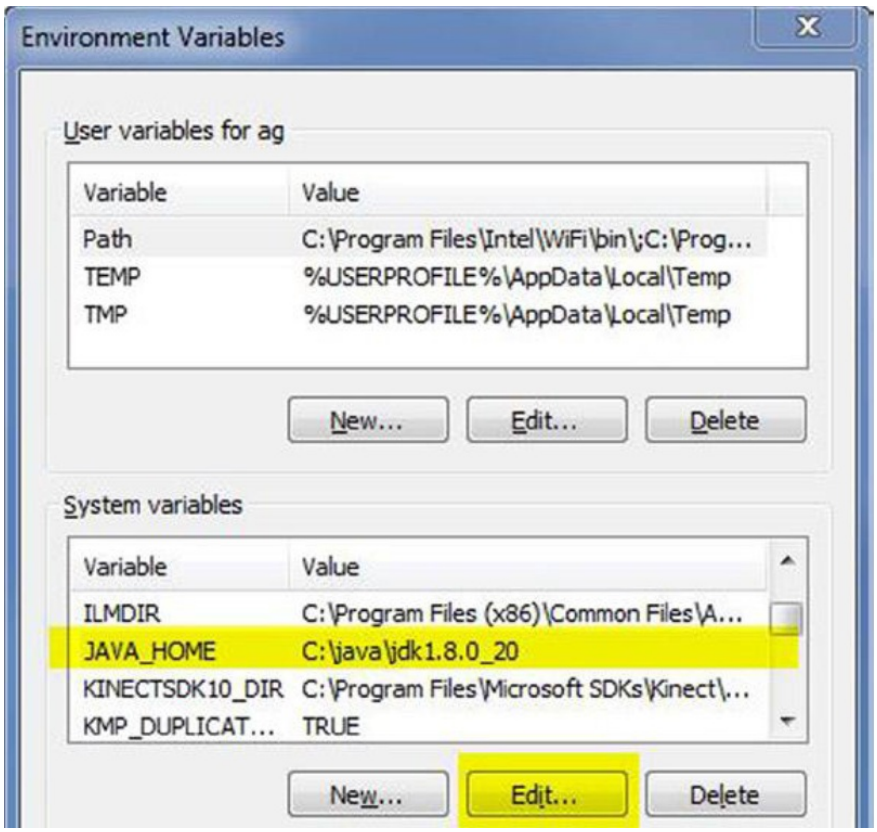


Figure 1.11: *Environmental variables*

Clicking either New or Edit displays a dialog box similar to Figure 1.12. Be sure to type JAVA_HOME in the Variable Name field. In the Variable Value field, type the location where you installed the JDK earlier. Now click OK.

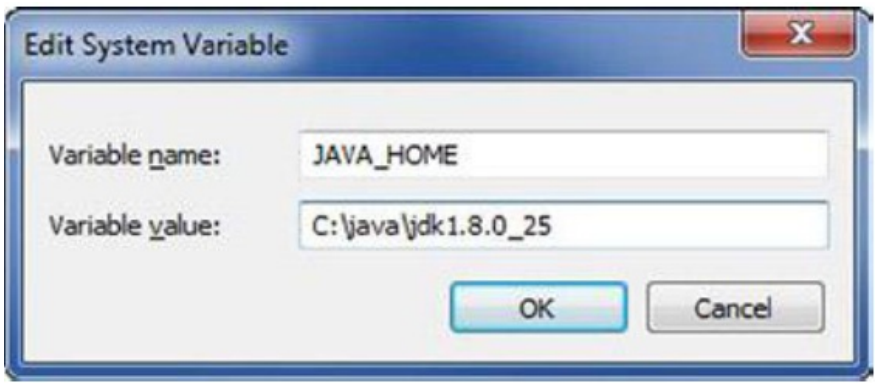


Figure 1.12: *Edit the JAVA_HOME environmental variable*

Just as you did with the JAVA_HOME environmental variable, you will need to edit the PATH environmental variable. See Figure 1.9. Place your cursor at the end of the Variable Value field and type the following:
;%JAVA_HOME%\bin

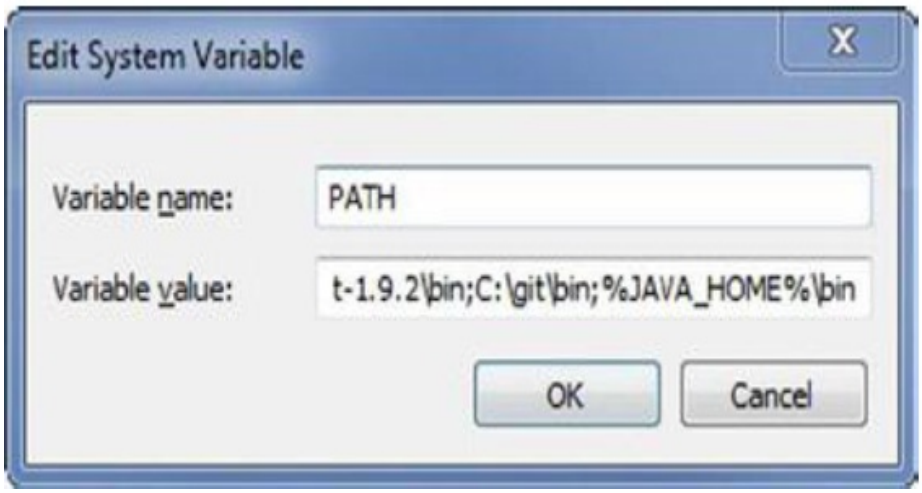


Figure 1.13: *Edit the PATH environmental variable*

Now click OK, OK, OK to accept these changes and back out of the system properties.

3.3 Installing the Java Development Kit on Mac

The first two steps in installing the JDK for Mac and Windows are the same. Click on the following link to visit the Oracle site: www.oracle.com/technetwork/java/javase/downloads/index.html

When the page opens, click the Java Download button as seen in the Figure below.

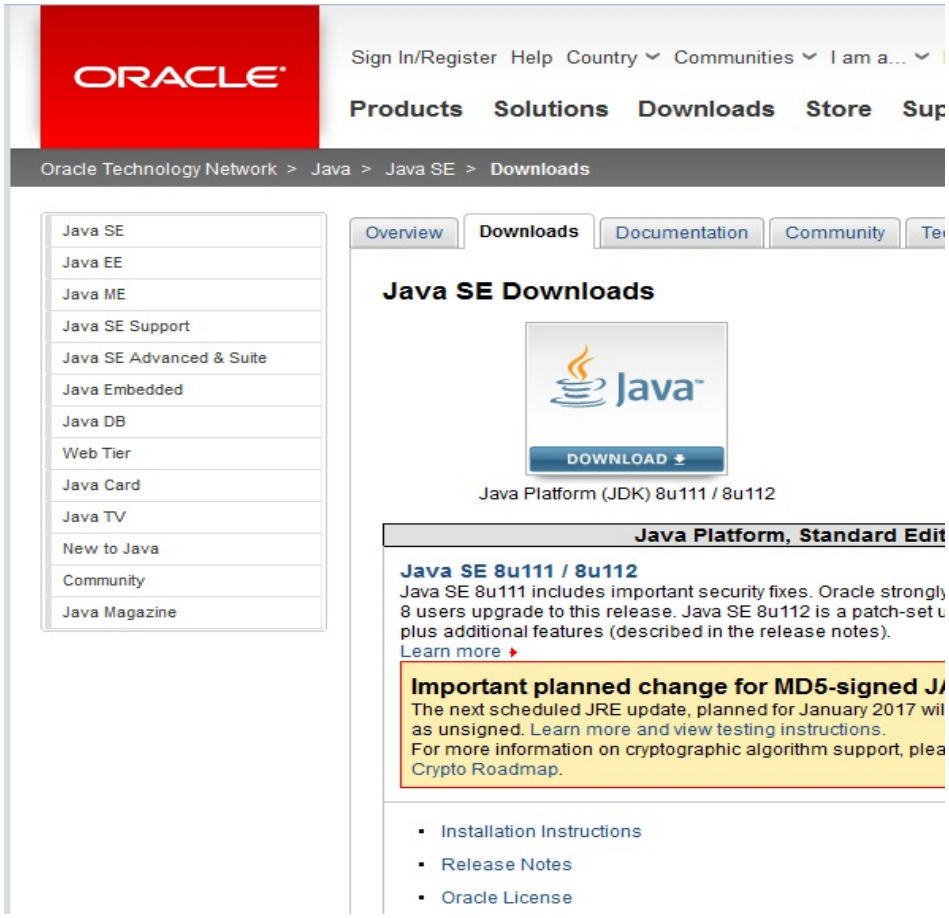


Figure 1.14: *The Java Download button on the Java Downloads page*

Downloading the JDK on Mac

After clicking on the download button, accept the license agreement for Mac just as shown in the figure below by clicking on the radio button. Ensure that you have chosen the correct JDK for the operating system. select Mac OSX64 for 64-bit version of OS X. Always select the latest version for your downloads, the moment the download is finished your can start your installation on your computer.

Looking for JDK 8 on ARM?
JDK 8 for ARM downloads have moved to the JDK 8 for ARM download page.

Java SE Development Kit 8u25

You must accept the Oracle Binary Code License Agreement for Java SE to download this software.

Accept License Agreement Decline License Agreement

Product / File Description	File Size	Download
Linux x86	135.24 MB	jdk-8u25-linux-i586.rpm
Linux x86	154.88 MB	jdk-8u25-linux-i586.tar.gz
Linux x64	135.6 MB	jdk-8u25-linux-x64.rpm
Linux x64	153.42 MB	jdk-8u25-linux-x64.tar.gz
Mac OS X x64	209.13 MB	jdk-8u25-macosx-x64.dmg
Solaris SPARC 64-bit (SVR4 package)	137.01 MB	jdk-8u25-solaris-sparcv9.tar.Z
Solaris SPARC 64-bit	97.14 MB	jdk-8u25-solaris-sparcv9.tar.gz
Solaris x64 (SVR4 package)	137.11 MB	jdk-8u25-solaris-x64.tar.Z
Solaris x64	94.24 MB	jdk-8u25-solaris-x64.tar.gz
Windows x86	157.26 MB	jdk-8u25-windows-i586.exe
Windows x64	169.62 MB	jdk-8u25-windows-x64.exe

Java SE Development Kit 8u25 Demos and Samples Downloads

Java SE Development Kit 8u25 Demos and Samples Downloads are released under the Oracle BSD License.

Product / File Description	File Size	Download
Linux x86	58.63 MB	jdk-8u25-linux-i586-demos.rpm
Linux x86	58.52 MB	jdk-8u25-linux-i586-demos.tar.gz

***Figure 1.15**: Accept the license agreement and click the appropriate link for Mac*

Running the JDK Wizard on Mac

Double-click the .dmg file to run it, click the .pkg file to begin the wizard and click Continue as prompted, see illustration below.

Figure 1.16: *JDK 8 Update 25.pkg*

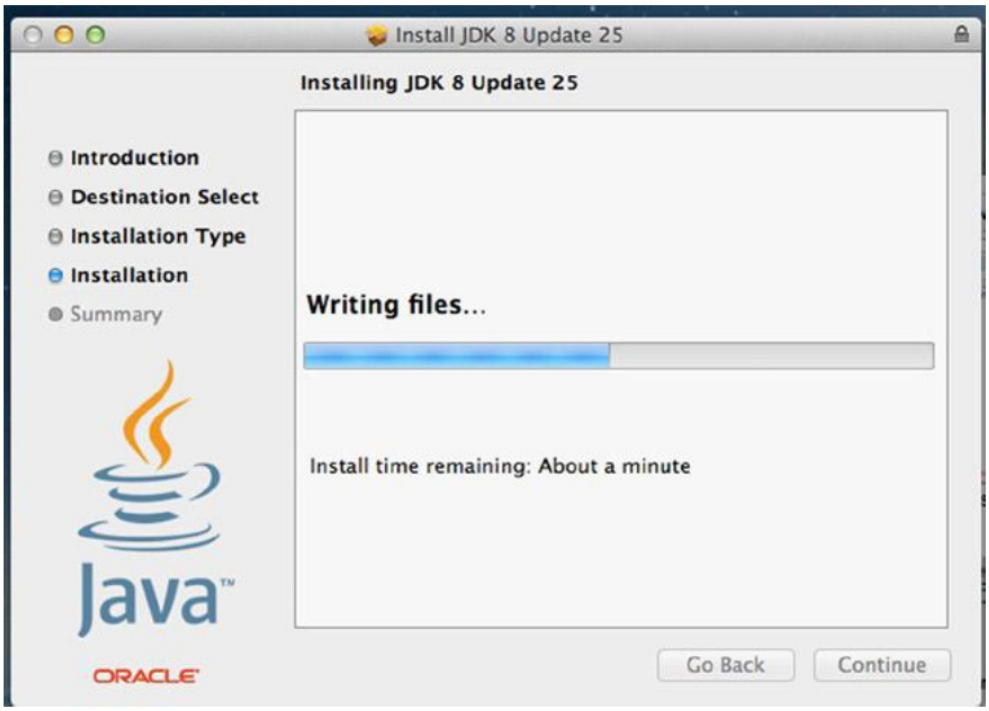


Figure 1.17: *Installation Wizard*

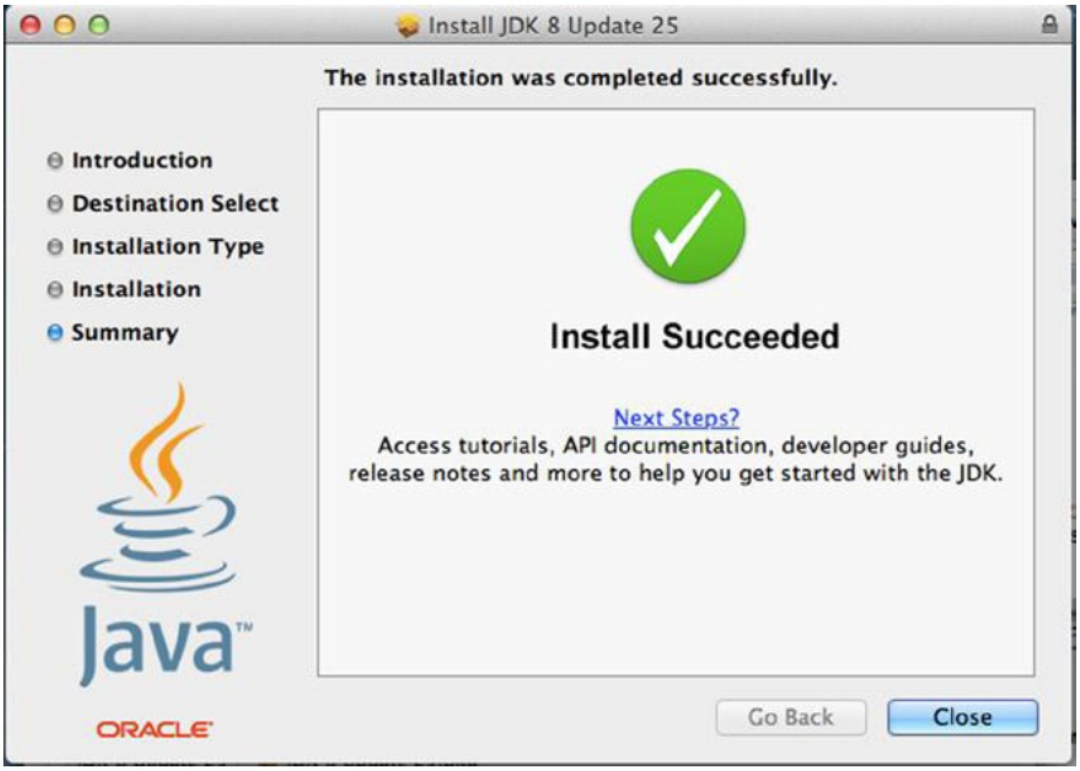


Figure 1.18: *Installation success*

Conclusion

We have discussed about setting up your environment for Java programming language. You will need a Microsoft Windows 7 or 8 with a Pentium 2 200-MHz process; computer with a minimum of 128 MB of RAM (256 MB of RAM recommended) with a disk space: 124 MB for JRE; 2 MB for Java Update or Intel-based Mac running Mac OS X 10.8.3+, 10.9+ for Mac.

Unit summary

In this unit, you learned that:

- There are system requirement for different operating systems when setting up your environment for Java programming language.
- Installing the Java Development Kit on Windows or Mac operating system, you will need to download, install, and configure the latest JDK.

Assessment

1. Outline the steps required to configuring Java Development Kit on Windows

VIDEO

http://tinyurl.com/y9zu2ogs

http://tinyurl.com/yapy3w9f

http://tinyurl.com/ybucovvk

Unit 3

Basic Syntax

Java supports some basic programming elements, such as data types, keywords, literals, and variables. Keywords are the reserved words for Java programming language, which cannot be used as names for variables, class, or method.

Before we begin with writing Java programs, it is very important to keep in mind the following points.

- Case Sensitivity - Java is case sensitive, which means identifier Hello and hello would have different meaning in Java.
- Class Names - For all class names the first letter should be in Upper Case. If several words are used to form a name of the class, each inner word's first letter should be in Upper Case.

Example: *class HelloWorldClass*

- Method Names - All method names should start with a Lower Case letter. If several words are used to form the name of the method, then each inner word's first letter should be in Upper Case.

Example: *public void myMethodName()*

- public static void main(String args[]) - Java program processing starts from the main() method which is a mandatory part of every Java program.

Outcomes

Upon completion of this unit you will be able to use the following in Java programming:

- Data Types in Java
- Keywords
- Variables and Literals
- Operators

Terminology

Data Type:	In programming, classification of a particular type of information.
Keyword:	An index entry that identifies a specific record or document
Variables:	A symbol or name that stands for a value. For example, in the expression (x+y), x and y are variables.
Literals:	In programming, a value written exactly as it's meant to be interpreted. In contrast, a *variable* is a name that can represent different values during the execution of the program.
Operators:	A symbol that represents a specific action. For

example, a plus sign (+) is an operator that represents addition.

First Java Program

Let us look at a simple code that will print the words ***Hello World***.

```
1 public class HelloWorldClass {
2 /* This is my first java program.
3 * This will print 'Hello World' as the output
4 */
5 public static void main(String []args) {
6 System.out.println("Hello World"); // prints Hello World
7 }
8 }
```

Let's look at how to save the file, compile, and run the program. Please follow the subsequent steps:

- Open notepad and add the code as above.
- Save the file as HelloWorldClass.java.
- Open a command prompt window and go to the directory where you saved the class. Assume it's C:\.
- Type 'javacHelloWorldClass.java' and press enter to compile your code. If there are no errors in your code, the command prompt will take you to the next line (Assumption : The path variable is set).
- Now, type ' java HelloWorldClass ' to run your program
- You will be able to see ' Hello World ' printed on the window.

```
C:\>javacHelloWorldClass.java
C:\> java HelloWorldClass
Hello World
```

Data Types

Data types are used to define the operations possible on variables and the storage method. The data stored in memory of the computer can be of many types. For example, a person's age is stored as a numeric value and an address is stored as alphanumeric characters. Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory.
There are two data types available in Java:

- Primitive Data types

- Abstract Data types

Primitive Data Types

The built-in data types in Java are known as the *primitive* or the *simple* data types.

There are eight primitive data types in Java, which are further grouped in the following categories:

- Integer type: Can store whole number values. The size of the values of the variables depends upon the chosen integer data type. The four integer data types are:
    - byte
    - short
    - int
    - long

- Floating point type: Can store fractional numbers. The two types of floating point type are:
    - float
    - double
- Boolean type: Can store only the true and false values. Boolean data type is required when a condition has to be checked. The true or false value of the expression or the condition determines further execution of the Java program.
- Character type: Can store symbols, such as letters and numbers. In character type, there is one data type, char.

Abstract data types

The abstract data types include the data types derived from the primitive data types and have more functions than primitive data types. For example, String is an abstract data type that can store letters, digits, and other characters, such as /, (), :, :, $, and #. You cannot perform calculations on a variable of the String data type even if the data stored in it has digits. However, String provides methods for concatenating two strings, searching for one string within another, and extracting a part of a string.

Keywords

The keywords are the reserved words for a language, which express the language features. Keywords cannot be used for naming purpose of variables, constants, or classes. Java is a case sensitive language and the keywords should be written in lowercase only. The keywords with allor some letters in uppercase can be treated as variable name but that should be avoided. The following table lists the Java keywords:

Table 3.1: Java Keywords

Abstract	boolean	break	byte
Case	catch	char	class
Const	continue	default	do
Double	else	extends	finally
Finally	float	for	goto
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

Variables

A variable is used to store and manipulate data or values in programs. A *variable* is the name that refers to a memory location where some data value is stored. You can assign different values to a variable during program execution. Java allocates memory to each variable that you use in your program. If the name, number, is used to refer to an area in memory in which a value is stored, number is a variable.

You must declare all variables before they can be used. Following is the basic form of a variable declaration:

data type variable [= value][, variable [= value] ...] ;

Here *data type* is one of Java's datatypes and *variable* is the name of the variable. To declare more than one variable of the specified type, you can use a comma-separated list.

Following are valid examples of variable declaration and initialization in Java:

```
int a, b, c; // Declares three ints, a, b, and c.
int a = 10, b = 10; // Example of initialization
byte B = 22; // initializes a byte type variable B.
double pi = 3.14159; // declares and assigns a value of PI.
char a = 'a'; // the char variable a is initialized with value 'a'
```

Types of Variables

The area or the region of a program where a variable can be accessed is known as variable scope. The various types of variables on the basis of the variable scope in Java are:

- Class variables: Are accessible within a class and its objects. The class variables are declared inside the class before their use.

- Local variables: Are declared inside a method. Their scope is within the block of code in which they are defined. They are local to the block of code and are not accessible outside the method.
- Instance variables: Are declared inside a class and are created when the class is instantiated. Objects give different values to instance variables as per the specific requirements of the object of that class type.
- Static variables: Are allocated memory only once but are globally accessible to all instances of a class. Therefore, when an instance of a class is destroyed, the static variable is not destroyed and is available to other instances of that class.

Literals in Java

Literalsare the values to be stored in variables and constants. A literal contains a sequence of characters, such as digits, alphabets, or any other symbol that represents the value to be stored.

The various types of literals in Java are:

- Integer literals: Are numeric type values. The numerical values can be represented in octal and hexadecimal notation. The octal notation of a number is prefixed by zero and hexadecimal numbers are prefixed by 0x. For example, n=0123 is integer literal in octal notation, n=0x456 is integer literal in hexadecimal notation, and n=2 is decimal notation for an integer literal.
- Floating point literals: Are numeric values with fractional part. For example, x=7.9 is a floating point literal.
- Character literals: Are represented in single quotation marks. For example, x='k' is a character literal.
- String literals: Are enclosed in double quotation marks. For example, x="James" is a string literal.
- Boolean literals: Are the literals having value, true or false. For example, x= false is a Boolean literal.

Operators

To manipulate data and variables in Java, you use operators, which accept one or
more operands or arguments and produce an output. We can divide all the Java operators into the following groups:

- Arithmetic Operators
- Assignment Operators
- Relational Operators
- Logical Operators

Arithmetic Operators

Arithmetic operators are used to compute mathematical expressions. The following table lists the various arithmetic operators:

Table 3.2: Arithmetic Operators

Operator	**Operation**
+	Adds two operands **Example:** A + B will give 30
-	Subtracts one operand from another **Example:** A - B will give -10
*	Multiplies two operands **Example:** A * B will give 200
/	Divides two operands **Example:** B / A will give 2
%	Divides left-hand operand by right-hand operand and returns remainder **Example:** B % A will give 0
++	Increments a variable Increases the value of operand by 1 **Example:** B++ gives 21
--	Decrements a variable Decreases the value of operand by 1 **Example:** B-- gives 19

Example 1

The following program is a simple example which demonstrates the arithmetic operators. Copy and paste the following Java program in Demo.java file, and compile and run this program:

```
1 public class Demo {
2 public static void main(String args[]) {
3 int a = 10;
4 int b = 20;
5 int c = 25;
6 int d = 25;
7 System.out.println("a + b = " + (a + b) );
8 System.out.println("a - b = " + (a - b) );
9 System.out.println("a * b = " + (a * b) );
10 System.out.println("b / a = " + (b / a) );
11 System.out.println("b % a = " + (b % a) );
12 System.out.println("c % a = " + (c % a) );
13 System.out.println("a++ = " + (a++) );
14 System.out.println("b-- = " + (a--) );
15 // Check the difference in d++ and ++d
16 System.out.println("d++ = " + (d++) );
17 System.out.println("++d = " + (++d) );
18 } }
```

This will produce the following result:

a + b = 30
a - b = -10
a * b = 200
b / a = 2
b % a = 0
c % a = 5
a++ = 10
b-- = 11
d++ = 25
++d = 27

Assignment Operator

You use the *assignment operator* (=) to assign a value to a variable. The following syntax is used for the assignment operator:

Table 3.3: Assignment Operators

Operator	**Operation**
=	Simple assignment operator. Assigns values from right side operands to left side operand. **Example:** C = A + B will assign value of A + B into C
+=	Add AND assignment operator. It adds right operand to the left operand and assign the result to left operand. **Example:** C += A is equivalent to C = C + A
-=	Subtract AND assignment operator. It subtracts right operand from the left operand and assign the result to left operand. **Example:** C -= A is equivalent to C = C – A
*=	Multiply AND assignment operator. It multiplies right operand with the left operand and assign the result to left operand. **Example:** C *= A is equivalent to C = C * A
/=	Divide AND assignment operator. It divides left operand with the right operand and assign the result to left operand. **Example:** C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator. It takes modulus using two operands and assign the result to left operand. **Example:** C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator. **Example:** C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator **Example:** C >>= 2 is same as C = C >> 2
&-	Bitwise AND assignment operator.

	Example: C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator. **Example:** C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator. **Example:** C |= 2 is same as C = C | 2

Example 2

The following program is a simple example that demonstrates the assignment operators. Copy and paste the following Java program in Demo.java file. Compile and run this program:

```
1 public class Dest {
2 public static void main(String args[]) {
3 int a = 10;
4 int b = 20;
5 int c = 0;
6 c = a + b;
7 System.out.println("c = a + b = " + c );
8 c += a ;
9 System.out.println("c += a = " + c );
10 c -= a ;
11 System.out.println("c -= a = " + c );
12 c *= a ;
13 System.out.println("c *= a = " + c );
14a = 10;
15 c = 15;
16 c /= a ;
17 System.out.println("c /= a = " + c );
18 a = 10;
19 c = 15;
20 c %= a ;
21 System.out.println("c %= a = " + c );
22 c <<= 2 ;
23 System.out.println("c <<= 2 = " + c );
24 c >>= 2 ;
25 System.out.println("c >>= 2 = " + c );
26 c >>= 2 ;
27 System.out.println("c >>= a = " + c );
28 c &= a ;
29 System.out.println("c &= 2 = " + c );
30 c ^= a ;
31 System.out.println("c ^= a = " + c );
32 c |= a ;
33 System.out.println("c |= a = " + c );
34 }
35 }
```

This will produce the following result:

```
c = a + b = 30
c += a = 40
c -= a = 30
c *= a = 300
c /= a = 1
c %= a = 5
c <<= 2 = 20
c >>= 2 = 5
c >>= 2 = 1
c&= a = 0
c ^= a = 10
c |= a = 10
```

Relational Operators

Relational operators are used to compare the values of two variables or operands and find the relationship between the two. The relational operators are therefore called comparison operators also. The following table lists the various relational operators in Java:

Table 3.4: Relational Operators

Operator	**Operation**
== (equal to)	Checks if the values of two operands are equal or not, if yes then condition becomes true. **Example:** (A == B) is not true.
!= (not equal to)	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true. **Example:** (A != B) is true.
> (greater than)	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. **Example:** (A > B) is not true.
< (less than)	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. **Example:** (A < B) is true.
>= (greater than or equal to)	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. **Example:** (A >= B) is not true.
<= (less than or equal to)	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. **Example:** (A <= B) is true.

Example 3

The following program is a simple example that demonstrates the relational operators. Copy and paste the following Java program in Demo.java file and compile and run this program.

```
1 public class Demo {
2 public static void main(String args[]) {
3 int a = 10;
4 int b = 20;
5 System.out.println("a == b = " + (a == b) );
6 System.out.println("a != b = " + (a != b) );
7 System.out.println("a > b = " + (a > b) );
8 System.out.println("a < b = " + (a < b) );
9 System.out.println("b >= a = " + (b >= a) );
10 System.out.println("b <= a = " + (b <= a) );
11 }
12 }
```

This will produce the following result:

```
a == b = false
a != b = true
a > b = false
a < b = true
b >= a = true
b <= a = false
```

The Logical Operators

The *logical operators* are used to combine multiple conditions in one Boolean expression. The following table lists the logical operators:

Assume Boolean variables A holds true and variable B holds false, then:

Table 3.5: Logical Operators

Operator	**Operation**
&& (logical and)	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true. **Example:** (A && B) is false.
|| (logical or)	Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true. **Example:** (A || B) is true.
! (logical not)	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a

	condition is true then Logical NOT operator will make false. **Example:** !(A && B) is true.
< (less than)	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. **Example:** (A < B) is true.
>= (greater than or equal to)	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. **Example:** (A >= B) is not true.
<= (less than or equal to)	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. **Example:** (A <= B) is true.

Example 4

The following simple example program demonstrates the logical operators. Copy and paste the following Java program in Test.java file and compile and run this program:

```
1 public class Demo {
2 public static void main(String args[]) {
3 boolean a = true;
4 boolean b = false;
5 System.out.println("a && b = " + (a&&b));
6 System.out.println("a || b = " + (a||b) );
7 System.out.println("!(a && b) = " + !(a && b));
8 }
9 }
```

This will produce the following result:

```
a&& b = false
a || b = true
!(a && b) = true
```

All Java components require names. Names used for classes, variables, and methods.

Unit summary

In this unit, you learned that:

- The various data types defined in Java are:
  - Integers: Include byte, short, int, and long data types.
  - Floating-point numbers: Include double and float data types.
  - Characters: Include char data type.
  - Boolean: Include boolean data type.
- The built-in or the intrinsic data types in Java are known as

the primitive or the simple data types.

- The abstract data types include the data types derived from the primitive data types.
- The keywords are the reserved words for a language, which express the language features.
- A variable is the basic storage unit in Java. It is the name that refers to a memory location where some data value is stored.
- The various types of variables are:
  - Class variables
  - Instance variable
  - Local variables
  - Static variables
- You use operators in Java to manipulate data and variables. The various operators are assignment, arithmetic operators, relational and logical operators.

Assessment

Fill in the blanks in each of the following statements:

a) The output of the expression, 16 % 3 is ----------?

b) What is the default value of the float data type? ----------

c) Consider the statements:

Statement A: The name of a variable can begin with a digit? True or False

Statement B: The name of a variable can contain white spaces? True or False

d) ----------variables are the local variables that are accessed by the function in
which the variables are declared.

e) ----------literals are enclosed in single quotes

Unit 4

Selection, Decision & Repetition

Is there anyone who has never used an ATM machine? Typically, a bank offers ATM customers several options: withdraw cash, make a deposit, check a balance, and so on. A customer chooses a transaction and the ATM software responds accordingly. Indeed, the ATM machine (or more precisely, the software controlling the machine) accepts the user's decision and implements it.

When ordering a CD from an online vendor, a buyer supplies his credit card number. If the number is valid, the vendor's software processes the order; if the entry is invalid, the program prompts the customer to re-enter the number. The program selects its response or subsequent action based on the validity of the credit card number that a customer submits.
In each scenario, a computer program selects the next action based upon predetermined criteria or conditions. In this unit, you will learn how to add selection to your programs using Java's two selection (or conditional) statements while the next unit will conclude this by treating the switch statement:

1. The if statement,
2. The if-else statement, and

Each option adds the capability of choice and decision-making to a program. In fact, just about every program that you write from now on will utilize at least one of these statements.
This unit discusses the conditional statements which emphasize on the switch statement, repetitive statement and while the statements. The section/unit promises to be simple attractive and useful.

Outcomes

Upon completion of this unit you will be able to execute:

- Selection as a mechanism for controlling the flow of a program,
- The if statement, the if-else statement, and the switch statement,
- Nested selection statements,
- The else-if construction.
- The switch statement
- Repetitive statement
- and the while statement

Terminology

Selection:	Also called a decision, one of the three basic logic structures in computer programming. The other two logic structures are sequence and loop.

Syntax:	Refers to the spelling and grammar of a programming language
Block:	A block is a group of statements enclosed by matching curly braces.
Boolean expression:	An expression that results in a value of either TRUE or FALSE.

The If Statement & If-Else Statement

The If Statement

We begin with a very simple situation where selection is absolutely necessary to accomplish the required task.

The syntax for an if statement is:

```
if( boolean-expression )
{
statement-1;
statement-2;
...
statement-n;
}
```

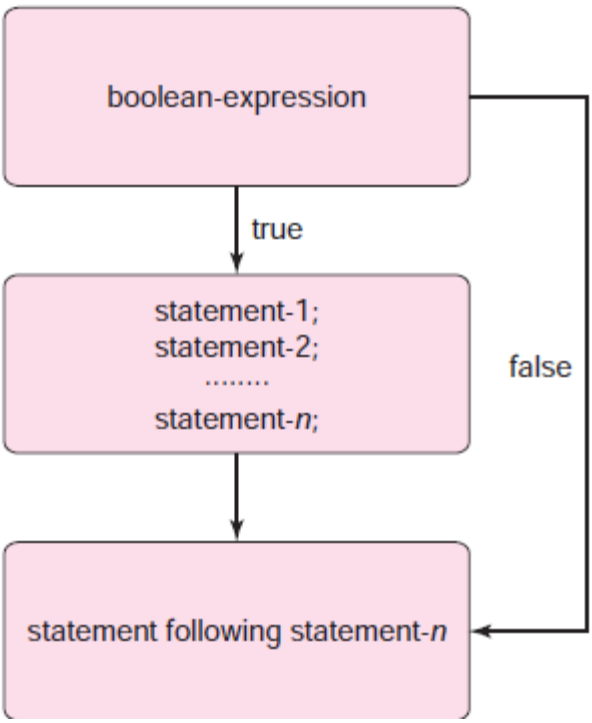


Figure 1.19 the if statement

An if statement is also termed a conditional or selection statement.

- The phrase if (boolean-expression) is called the if clause

- The boolean expression is also called a boolean condition (or simply a condition).
- The statement-list enclosed by curly braces comprises a block or compound statement
- If the statement-list consists of a single statement the braces may be omitted. A single statement without the braces is not considered a block.

Example 1
When you buy an item from an online vendor, a $5.00 shipping fee is waived for purchases of $25.00 or more.

Write a program that calculates the final cost of an item, including sales tax and shipping, if applicable. Sales tax is 8% of the purchase price.

Solution A decision statement appears in bold on lines 18–22.

```
1. // Given the price of an item, this program calculates the 8% sales tax, adds a $5.00 shipping fee
2. // for items costing less than $25.00 and prints the total cost of the item.
3. importjava.util.*;
4. public class BillCalculator
5. {
6. public static void main(String[] args)
7. {
8. Scanner input = new Scanner (System.in);
9. double sale, taxes, total;
10. final double TAX_RATE = 0.08; // notice TAX_RATE is a constant
11. final double SHIPPING_FEE - 5.00; // another constant
12. System.out.print("Enter the item price: ");
13. sale = input.nextDouble();
14. taxes =sale* TAX_RATE;
15. total = sale+ taxes;
16. System.out.println("Sale: $" + sale);
17. System.out.println("Tax: $" + taxes);
18. if ( sale <25.00)
19. {
20. total +=SHIPPING_FEE;
21. System.out.println("Shipping is $5.00");
22. }
23. System.out.println("Final cost: $" + total);
24. }
25. }
```

Running the program twice produces the following output:

Output 1
Enter the item price: $ **34.00,** Tax: $2.72, Final cost: $36.72

Output 2
Enter the item price: $ **16.00,** Tax: $1.28, Shipping is $5.00, Final cost: $22.28

Discussion The first display shows the total cost without a shipping fee. The sale is more than $25.00, so shipping is free. However, when the program runs a second time,because the sale is just $16.00, a $5.00 shipping fee is added to the order. Most of the code in the preceding program is straightforward and requires no elaboration.

Example 2
The following code fragment determines the largest of three integers (a, b, and c) is an example of an if statement that does not contain curly braces.

```
1. int max = a; //a is biggest so far
2. if (b >max) // is b bigger than the current maximum?
3. max =b; // if so, set max to b
4. if (c >max) // is c bigger than the current maximum?
5. max =c; // if so set max to c
6. System.out.println ("The maximum value is "+max);
```

Suppose that a, b , and c have the values 3, 5, and 4, respectively. Let's step through the fragment:
Table 4.1 the use of the if statement

	a	b	c	max
Line 1: Variable max is set to 3. So, the "current" maximum value is 3.	3	5	4	3
Line 2: The boolean condition b > max is true (since 5 > 3) so the statement on line 3 executes	3	5	4	3
Line 3: Variable max is set to 5. Thus, the current maximum is 5.	3	5	4	5
Line 4: The boolean condition c > max is false so the statement on line 5 is skipped.	3	5	4	5
Line 6: The string "the maximum value is 5" is displayed	3	5	4	5

Alternatively, the same fragment can be written using curly braces:

```
int max = a;
if (b >max)
{
max =b;
}
if (c >max)
{
```

```
max =c;
}
System.out.println("The maximum value is "+max);
```

The If-Else Statement

As you have seen, an if statement allows a program to decide whether to execute or ignore a particular group of statements. The if-else statement provides an alternative: if the boolean condition is true, one group of statements executes, but if the condition evaluates to false, a different group is selected.

The syntax for an if statement is:
if(boolean-expression)
statement-list-1
else
statement-list-2
where statement-list-1 and/or statement-list-2 can comprise single statements or a block. If boolean-expression is true then statement-list-1 is executed and statement-list-2 is skipped; otherwise, statement-list-1 is skipped and statement-list-2 is executed. Every time an if-else statement is encountered, one of the two statement-lists always executes.
See Figure 1.21 .

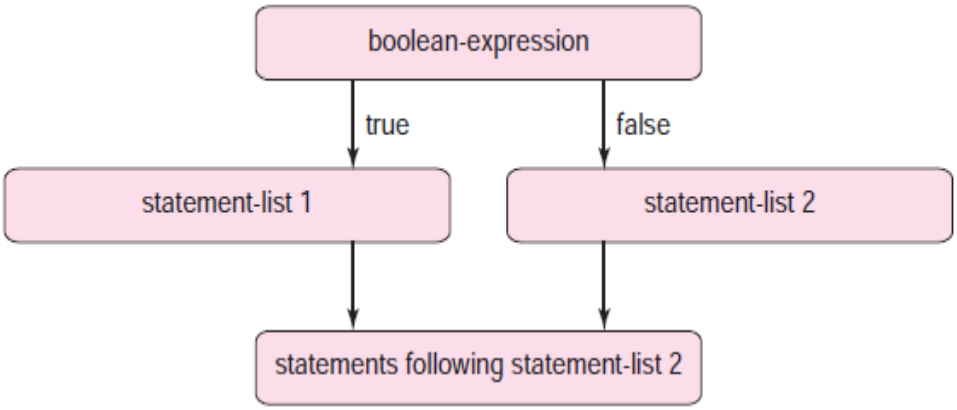


Figure 1.20. The if else statement

The following example uses an if-else statement in a program that converts U.S. dollars to euros, and euros to dollars based upon user input.

Example 3
Problem Statement Assume that one euro costs $1.31. Write a program that converts dollars to euros or euros to dollars based upon user input.

Java Solution The application prompts the user for an integer: 1 or 2. If the user enters “1,” a dollar amount is requested and the application displays the equivalent number of euros. If the user enters “2” or any other integer, euros are converted to dollars.

```
1. importjava.util.*;
2. public class CurrencyConverter
3. {
4. public static void main (String[] args)
5. {
6. Scanner input= new Scanner(System.in);
7. final double DOLLARS_PER_EURO = 1.31; // exchange rate
8. inttransactionType;
9. double euros, dollars;
10. System.out.print("Enter 1 to convert from dollars to euros and 2 from euros to dollars: " );
11. transactionType = input.nextInt();
12. if (transactionType==1) // dollars to euros
13. {
14. System.out.print("Number of dollars: ");
15. dollars =input.nextDouble();
16. euros =dollars/DOLLARS_PER_EURO;
17. System.out.println("Number of euros: "+euros);
18. }
19. else // otherwise euros to dollars
20. {
21. System.out.print("Number of euros: ");
22. euros =input.nextDouble();
23. dollars =euros* DOLLARS_PER_EURO;
24. System.out.println("Number of dollars: " +dollars);
25. }
26. }
27. }
```

Two sample executions of the program produce Output1 and Output 2.

Output 1

```
Enter 1 to convert from dollars to euros and 2 from euros to dollars:
1
Number of dollars: 335.36
Number of euros: 256.0
```

Output 2

```
Enter 1 to convert from dollars to euros and 2 from euros to dollars:
2
Number of euros: 6908
Number of dollars: 9049.48
```

Nested If-Else Statements

An if-else statement can be nested inside another if-else statement, which can be nested inside another if-else statement, and so on. For example, consider the following fragment:

```
1. int grade = input.nextInt(); //user supplies a grade
2. if ( grade >= 70 )
3. {
4. if ( grade >=90)
5. System.out.println( "High pass");
6. else
7. System.out.println("Pass");
8. }
9. else
10. System.out.println("Fail");
```

Here, an if-else statement (lines 4–7) is nested within an if-else statement so that several paths of execution are possible, depending on the value of grade.

• If, for example, the value of grade is 65, the condition on line 2 is false and the corresponding else clause of line 10 executes. The output is “Fail.” Notice that the if-else statement on lines 4–7 is skipped.

• If grade is 75, the boolean condition on line 2 is true . As a result, the if-else statement on lines 4–7 executes and the else clause on line 9 is skipped. Because grade is not greater than or equal to 90, the boolean condition of line 4 is false and the else clause of line 7 executes. The output is “Pass.”

• If grade has the value 95, the condition of line 2 is true, so the if-else statement of lines 4–7 executes and the else clause on line 9 is skipped. This time grade is greater than or equal to 90, so the condition on line 4 is true and the println(…) statement on line 5 executes. The output is “High pass .”

It is good programming practice to test every path through a nested if-else statement.

3.2 The Switch Statement, Repetition & The While Statement

The Switch Statement

Java’s switch statement sometimes offers a more compact alternative to the else-if construction.

The syntax of the switch statement is:

```
switch( switch-expression )
{
casecasevalue- 1: statement;
statement;
```

```
...
statement;
break;
casecasevalue- 2: statement;
statement;
...
statement;
break;
...
casecasevalue- n: statement;
statement;
statement;
break;
default: statement;
statement;
...
statement;
}
```

The following else-if segment displays a one-word description for each letter grade A through F.

```
if( grade == 'A')
System.out.println("Excellent");
else if (grade == 'B')
System.out.println("Good");
else if (grade== 'C')
System.out.println("Average");
else if (grade == 'D')
System.out.println("Passing");
else
System.out.println("Failure");
```

As you know, each boolean condition is evaluated in turn. When a condition evaluates to true, the corresponding println (…) statement executes and the else-if construction terminates.

The following switch statement accomplishes the same task.

```
switch(grade )
{
case 'A': System.out.println("Excellent"); break;
case 'B': System.out.println("Good"); break;
case 'C': System.out.println("Average"); break;
case 'D': System.out.println("Passing"); break;
default :System.out.println(Failure");
}
```

The switch statement works as follows:

• The value of grade is compared to each " case value" ('A', 'B', 'C', and 'D') until a match is found.

• If one of the case values matches the value of grade , the corresponding println(…)statement executes and the break statement terminates the switch statement.

• If no case value matches the value of grade, then the statement of the default case executes.

The switch statement behaves in a manner similar to the else-if construction.

Example 4
Problem Statement
Write a program that simulates an ATM machine. Use a switch statement rather than an else-if construction.
Solution

```
1. importjava.util.*;
2. public class ATMMachine
3. {
4. public static void main (String[] args)
5. {
6. Scanner input = new Scanner(System.in);
7. double balance = 5423.00, deposit, withdrawal;
8. int transaction;
9. System.out.println("Welcome! Enter your the number for your transaction");
10. System.out.println("Withdraw cash: 1");
11. System.out.println("Make a deposit: 2");
12. System.out.println("Check your balance: 3");
13. System.out.println("Exit: 4");
14. System.out.print("Transaction number: ");
15. transaction = input.nextInt();
16. switch (transaction)
17. {
18. case 1: System.out.println("Enter amount");
19. withdrawal =input.nextDouble();
20. if ( withdrawal >balance)
21. System.out.println("Invalid amount");
22. else
23. {
24. balance -= withdrawal;
25. System.out.println("Your new balance is $" +balance);
26. }
27. break;
28. case 2: System.out .println("Enter amount of deposit: ");
29. deposit =input.nextDouble();
30. balance +=deposit;
31. System.out.println("Your new balance is $"+balance);
32. break;
33. case 3: System.out.println("Your balance is $"+balance);
34. break;
35. case 4: System.out.println("Thank you.");
36. break;
37. default: System.out.println("Invalid transaction");
```

38. }
39. }
40. }

Discussion The preceding application produces output identical to the output of the earlier unit. However, this program accomplishes its task using a switch statement(lines 16–38) rather than the else-if construction.

We begin with line 16: switch (transaction)

The variable transaction, enclosed by parentheses and following the keyword switch, is called the switch expression. Following line 16, and enclosed in curly braces, you will notice a number of cases. Each case includes a possible value for this switch expression followed by a colon. In this example, these values are 1, 2, 3, or 4. (See lines 18, 28, 33, and 35.)

When the switch statement executes.
• each case value is examined in turn;
• if the value of transaction matches one of the case values, the code associated with that case is executed and the break statement terminates the switch statement;
• if the value of transaction does not match any of the case values, then the code associated with the default case (line 37) executes.

So, for example, if an ATM customer chooses transaction number 3 (line 15), then the value of transaction is 3. That's the value of the switch expression. This value 3 is compared to the case value on line 18, which is 1. There is no match. Next, the value is tested against the second case value (line 28); again no match. Finally the third case is tried. This time the value of the switch expression and the case value are both 3 and do, in fact, match. Consequently, the code associated with this case value (line 33) is executed, and the output is:

Your balance is $5423.0 No further testing is attempted. The break statement on line 34 terminates the switch statement.

The Else-If Version

```
if(score >= 35)
System.out.println("Score: "+ score+ ". Your personality is Type A");
else if (score >= 21)
System.out.println("Score: " + score+ ". You are between A and B tending towards A");
else if (score >= 12)
```

System.out.println("Score: " +score+ ". You are between A and B tending towards B");
else
System.out.println("Score: " + score+ ". Your personality is Type B");

The Switch Version
switch (score) // every value must be enumerated!
{
case 40:
case 39:
case 38:
case 37:
case 36:
case 35: System.out.println("Score: " + score+ ". Your personality is Type A");
break;
case 34:
case 33:
case 32:
case 31:
case 21: System.out.println("Score: " + score+ ". You are between A and B tending towards A"); break;
//etc.
}
Although the choice between switch and else-if is often a matter of preference, convenience,
or style, there are situations when the else-if construction is the only reasonable option. Example 2 presents such a case.

Problem Statement 5
Write a program that accepts the decisions of prisoners Bozo and Bongo and reports the result.

Solution

1. importjava.util.*;
2. public class PrisonersDilemma
3. {
4. public static void main (String[] args)
5. {
6. Scanner input = new Scanner(System.in);
7. boolean prisoner1Confesses = true;
8. boolean prisoner2Confesses = true;
9. int response;
10. // Enter data for Prisoner 1
11. System.out.println("For each prisoner enter 1 for a confession and 0 otherwise");
12. System.out.print("Prisoner1: ");

```
13. response = input.nextInt();
14. if (response== 0) // Prisoner 1 does not confess
15. prisoner1Confesses = false;
16. // Enter data for Prisoner 2
17. System.out.print("Prisoner2: ");
18. response = input.nextInt();
19. if (response == 0) // Prisoner 2 does not confess
20. prisoner2Confesses = false;
21. if (prisoner1Confesses && prisoner2Confesses) //both confess
22. System.out.println("Both confessed. Each gets 5 years!");
23. else if (prisoner1Confesses && !prisoner2Confesses) // 1 confesses; 2 does not
24. System.out.println("Prisoner 1 goes free; Prisoner 2 gets 10 years.");
25. else if (!prisoner1Confesses && prisoner2Confesses) // 2 confesses; 1 does not
26. System.out.println("Prisoner 2 goes free; Prisoner 1 gets 10 years.");
27. else // neither confess
28. System.out.println("Neither confessed. Each gets one year.");
29. }
30. }
```

Output

```
For each prisoner enter 1 for a confession and 0 otherwise
Prisoner1: 1
Prisoner2: 0
Prisoner 1 goes free; Prisoner 2 gets 10 years.
```

Loops, The Do-While Statement, For Statement, Nested Loop & Break Statement

There may be a situation when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.
Programming languages provide various control structures that allow for more complicated execution paths.

A **loop** statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages:

The While Statement

A **while** loop statement in Java programming language repeatedly executes a target statement as long as a given condition is true.

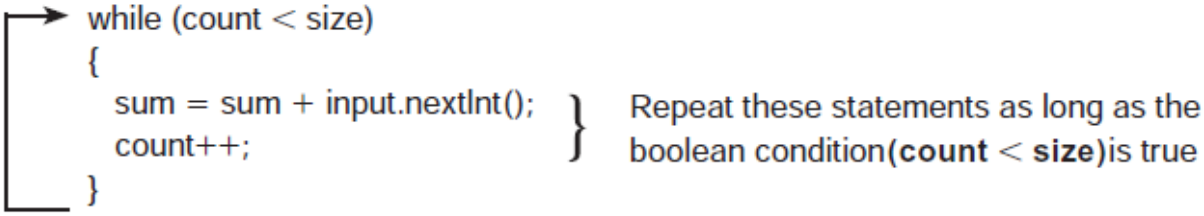


Figure 1.22 shows the action of the loop.

The syntax of the while statement is:
while (condition)
{
statement-1;
statement-2;
...
statement-n;
}

In general, the while statement executes as follows:
1. condition, a boolean expression, is evaluated.
2. If condition evaluates to true,
a. statement-1, statement-2 , . . . , statement-n execute.
b. Program control returns to the top of the loop.
c. The process repeats (go to step 1).
3. If condition evaluates to false ,
a. statement-1, statement-2 , . . . , statement-n are skipped .
b. Program control passes to the fi rst statement after the loop.

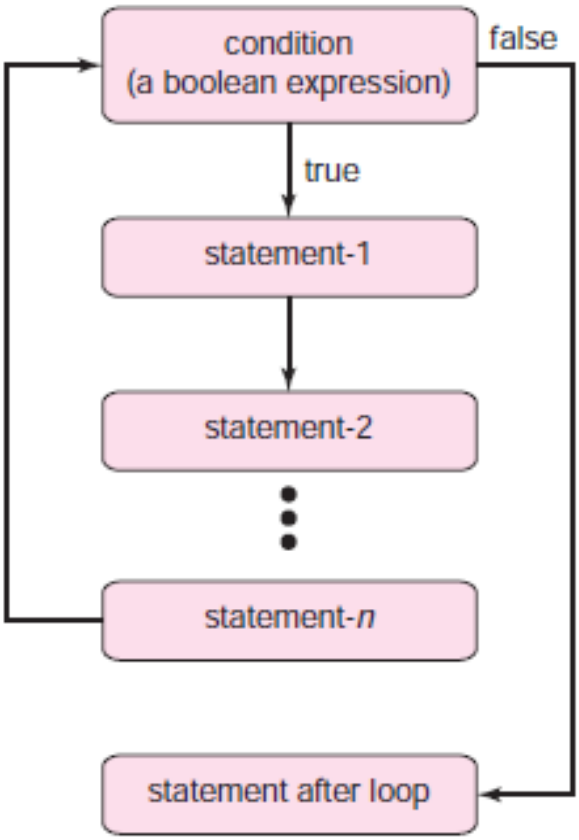


Figure 1.23 The semantics of the while statement

We can write a program that adds exactly5 integers and a different application that sums exactly 50 integers. But can we write a program flexible enough to add 5 integers, 50 integers, 50,000 integers, or even 50,000,000 integers?

With a while loop, the addition of 50 numbers can be achieved as easily and compactly as the addition of 5 or 50,000 numbers. The following segment adds 50 numbers with just a few lines of code. There is nothing special about 50, and we can just as easily add 500,000numbers.

```
1. int sum = 0;
2. int count = 0;
3. while(count< 50)
4. {
5. sum = sum + input.nextInt();
6. count++;
7. }
8. System.out.print(“Sum is “ + sum);
```

The statements on lines 3–8 execute as follows:

1. The condition on line 3 (the boolean expression, count <50) is evaluated.
2. If the condition, count < 50, is true , continue to line 5:
a. A number is accepted from the keyboard and added to sum (line 5).
b. Variable count is increased by 1 (line 6).
c. Program control returns to the “top of the loop” (line 3), and the process repeats.
3. However, if the condition on line 3 is false,
a. The statements on lines 5 and 6 are skipped.
b. Program control passes to line 8 and the sum is displayed.

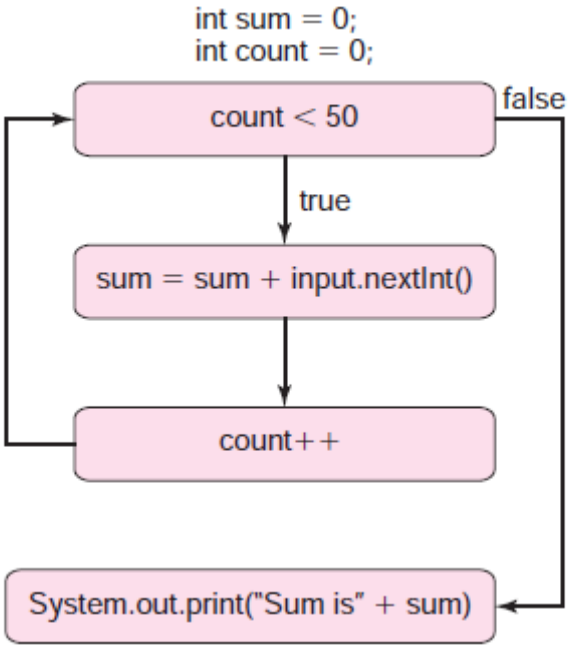


Figure 1.24 A loop that adds 50 integers

Example 6

Problem Statement Write a program that sums a list of integers supplied by a user. The list can be of any size. The program should prompt the user for the number of data.

Solution The following application utilizes three variables: size, sum, and count.
The addition is accomplished using a while loop similar to the loop in the segment that
precedes this example.

```
1. importjava.util.*;
2. public class AddEmUp
3. {
4. // adds an arbitrarily long list of integers
5. // the user first supplies the size of the list
6. public static void main (String[] args)
```

```
7. {
8. Scanner input = new Scanner(System.in);
9. int sum = 0; // Running sum
10. int count = 0; // Keeps track of the number of integers
11. int size ; // Size of the list
12. System.out.print("How many numbers would you like to add? ");
13. size = input.nextInt();
14. System.out.println("Enter the "+ size+ " numbers");
15. while (count<size) // while the number of data is less than size repeat:
16. {
17. sum =sum +input.nextInt(); // read an integer, add it to sum
18. count++; // keep track of the number of data
19. }
20. System.out.println("Sum: " + sum);
21. }
22. }
```

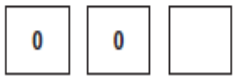
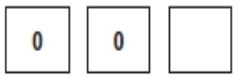
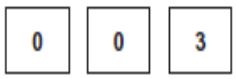
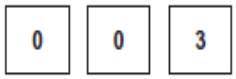
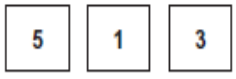
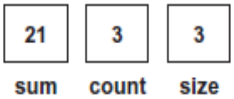
Output 1

How many numbers would you like to add? 3
Enter the 3 numbers
5, 7, 9
Sum: 21

Output 2

How many numbers would you like to add? 12
Enter the 12 numbers
23, 45, 65, 23, 43, 12, 87, 56, 34, 31, 84, 90
Sum: 593

Table 4.2 Showing the use of count, sum and size

	The statements on lines 9–11 declare three variables and initialize two of them to 0.
	The print statement on line 12 displays a prompt for the user. How many numbers would you like to add?
	Line 13 is an assignment. The value 3 (entered by the user) is assigned to variable size.
	The statement on line 14 prompts the user to enter the data: Enter the 3 numbers
	The program reaches the while loop. The first. action of the loop is the evaluation of the expression on line 15. In this case, the expression (count < size) is true. Consequently, the block on lines 16 through 19 executes: The user enters the number 5, 5 is added to sum, (sum is 5), and count increases to 1.
	Following line 19, control returns to line 15, i.e., the program loops back to line 15. Since the condition on line 15 (count < size) again evaluates to true, the statements of lines 16 through 19 execute again: The user enters 7, 7 is added to sum (sum is 12), and count increases to 2.
	For a third time, control returns to line 15 and again the expression count < size is true. So one more time, the block on lines 16 through 19 executes: The user enters 9, 9 is added to sum (sum is 21), and count increases to 3.
	Finally, control returns one last time to line 15. This time, however, because count and size are both equal to 3, the expression is false, so the block is skipped. Control passes to line 20, a println statement, which displays the value of sum: Sum: 21

The Do-While Statement

Although the while loop is sufficient for any task requiring repetition, Java provides two alternative statements: the do-while loop and the for loop.

If the condition of a while loop is initially false, the body of a while loop never executes.

In contrast, a do-while loop always executes the body of the loop at least once before checking the terminating condition.

A do-while loop checks the condition at the end of the loop body.

The syntax of the do-while statement is:

```
do
{
statement-1;
statement-2;
. . . ;
statement-n;
} while (condition);
```

As always, condition is a boolean expression and, if the block consists of single executable statement, the curly braces may be omitted.

Execution of the do-while statement proceeds as follows:

1. statement-1, statement-2, . . . , statement-n execute.
2. condition is evaluated.
3. If the condition is true , the process repeats (go back to statement-1).
4. If condition is false , the loop terminates and program control passes to the first statement following the loop.

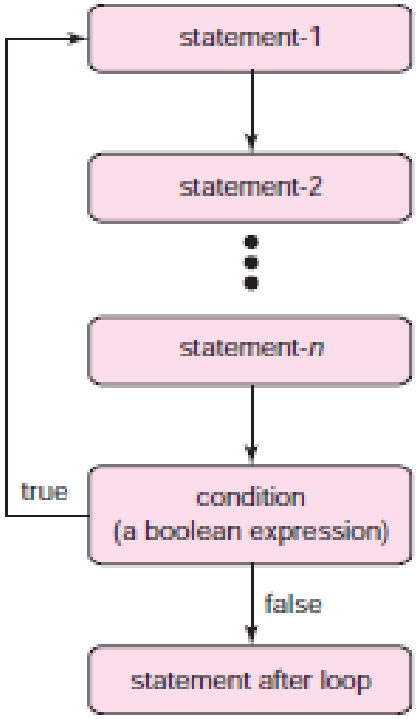


Figure 1.25 The semantics of the do while statement

For example, the following segment, which screens for bad input, is a natural application of a do-while statement.

```
1. int x; // must be positive
2. do
3. {
4. System.out.println("Enter a number > 0");
5. x = input.nextInt();
6. }while (x<= 0); // if negative, repeat
```

The loop executes as follows:
• The statement on line 4 prompts the user for a positive number.
• The statement on line 5 reads a value and assigns that value to variable x.
• The condition (x<= 0) on line 6 is evaluated. If the condition is true, the loop repeats the actions of lines 4 and 5; if the condition is false, the loop terminates.
A do-while loop is guaranteed to execute at least once. This is not the case with a while loop.

Example 7

Problem Statement Write a program that calculates the sum of a list of integers that is interactively supplied by a user. The program should prompt the user for the number of data. The program should ensure that each number supplied by the user is positive.

Solution

```
1. importjava.util.*;
2. public class DoWhileAdd
3. {
4. public static void main (String[] args)
5. {
6. Scanner input = new Scanner(System.in);
7. int size; // the number of integers to add
8. do // repeat until size is positive
9. {
10. System.out.print("How many numbers would you like to add? ");
11. size =input.nextInt();
12. } while (size <=0);
13. System.out.println("Enter the " + size+ " numbers");
14. int sum = 0; // the running sum
15. int count = 0; // keeps track of the number of data
16. while (count < size)
17. {
18. sum= sum+ input.nextInt(); // read the next integer, add to sum
19. count++; // increment counter
20. }
21. System.out.println("Sum: "+ sum);
22. }
23. }
```

Output

How many numbers would you like to add? **0**
How many numbers would you like to add? **-3**
How many numbers would you like to add? **3**
Enter the 3 numbers
5, 7, 9
Sum: 21

How does the do-while construction differ from that of the while loop?

The while loop is top-tested, that is, the condition is evaluated before any of the loop statements executes. If the condition of a while loop is initially false, the loop never executes. The do-while loop, on the other hand, is bottom-tested, that is, the condition is tested after the first iteration of the loop. A do-while loop always executes at least once.

Let’s take a second look at another Example, this time using a do-while loop.

Problem Statement Rewrite Example 1 using a d o-while loop rather than a while loop.

Example 8

Solution

```
1. importjava.util.*;
2. public class DoWhileAddEmUpAgain
3. {
4. public static void main (String[] args)
5. {
6. Scanner input= new Scanner(System.in);
7. finalint FLAG =-999;
8. int sum=_ 0; // Running sum
9. int number; // holds the next integer to be added
10. System.out.println("Enter the numbers end with " + FLAG);
11. number = input.nextInt();
12. do
13. {
14. sum +=number; // add the current integer to sum
15. number =input.nextInt();
16. } while (number !=FLAG);
17. System.out.println("Sum: " + sum);
19. }
20. }
```

The For Statement

Java provides a third alternative for repetition: the for statement. Use a for statement when you can count the number of times that a loop executes.

The syntax of the for statement is:

for(initialization; loop condition; update statement(s))
{
statement-1:
statement-2;
. . .
statement-n:
}

As usual, the braces may be omitted if the statement block consists of a single statement.

The semantics of the for statement are:

1. The initialization statement executes.
2. The loop condition (a boolean expression) is evaluated.
3. If the loop condition is true, then:

a. statement-1, statement-2, . . . , statement-n execute,
b. The update-statement(s) executes,
c. Go to step 2

4. If the loop condition is false, then program control passes to the first statement following
the block consisting of statement-1, statement-2, . . . , statement-n .
You should note that:

- The initialization is performed exactly once.
- The loop condition is always tested before the statement block executes.
- The update statement always executes after the actions of the statement block.
- The declared, initialized variables disappear after the for loop completes execution.

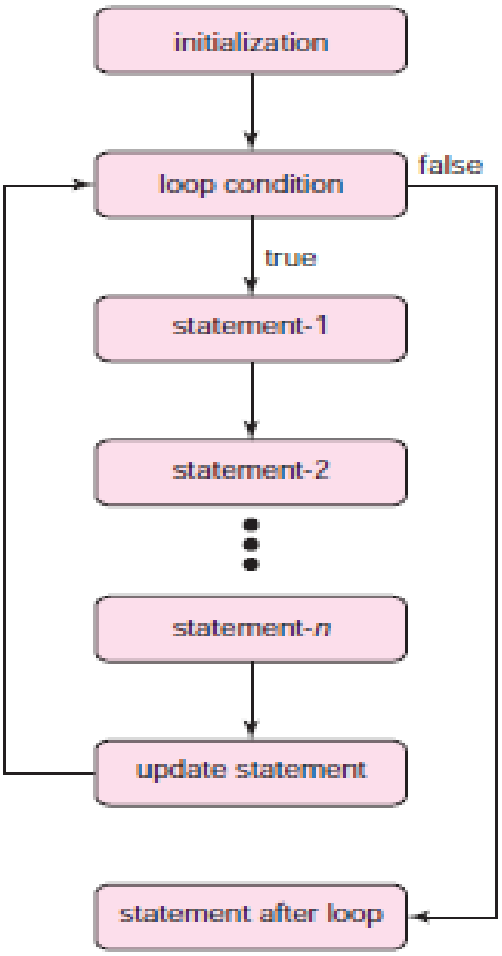
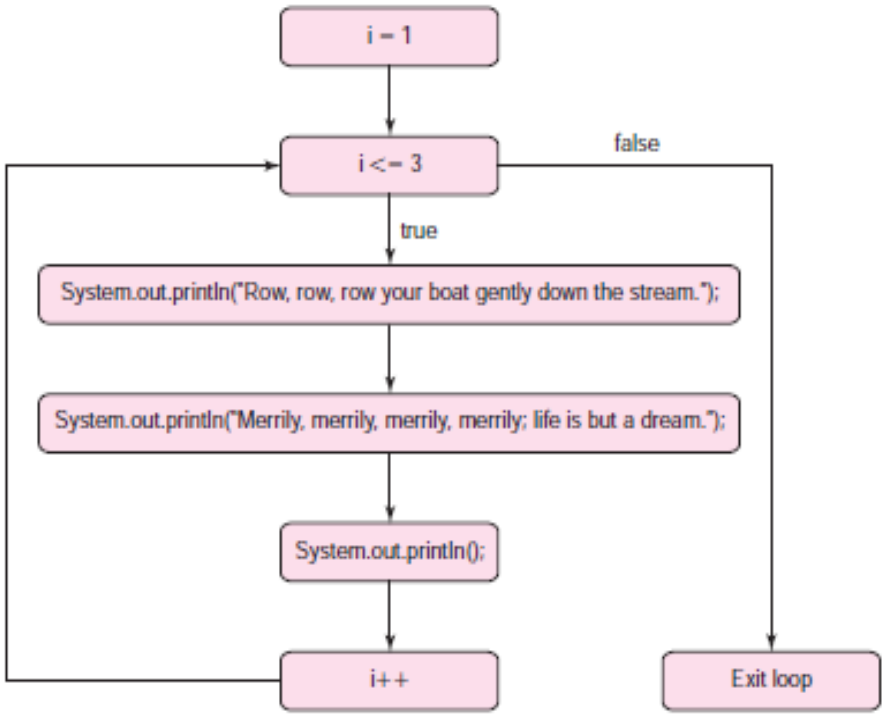


Figure 1.26 The semantics of the for statement

The following program segment uses a for loop to print the verse of a familiar, if boring, song exactly three times:

```
1. for (int i = 1; i <= 3; i++)
2. {
3. System.out.println (“Row, row, row your boat, gently down the stream,”);
4. System.out.println (“Merrily, merrily, merrily, merrily; life is but a dream”);
```

5. System.out.println();
6. }
The loop executes as follows:
1. The variable i is declared and initialized to 1 (int i = 1); i keeps track of the number of iterations; i counts.
2. The condition i <= 3 on line 1 is evaluated.
3. If the condition i <= 3 is true :
Lines 3, 4, and 5 execute. // Sing along if you wish!
The statement i++ on line 1 executes. Go to step 2 (check whether or not i <= 3).
4. If the condition i <= 3 is false, the loop terminates.
The variable i keeps track of the number of iterations. Before the body of the loop executes, the terminating condition (i<= 3) is checked. Once the body of the loop completes execution, the value of i is increased by 1.
Conveniently,
• the initial value of i , (i = 1),
• the loop condition, (i <= 3), and
• the update statement for i, (i++)
all appear together on line 1.



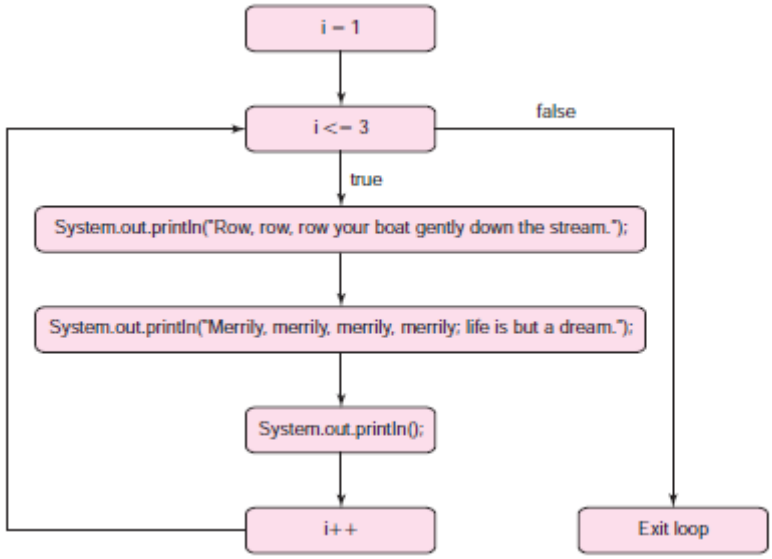


Figure 1.27 shows the program flow of this segment.

Example 8
Problem Statement
Using a for statement, write a program that sums a list of integers. The program should prompt the user for the size of the list.

Solution

```
1. importjava.util.*;
2. public class ForAddEmUp
3. {
4. public static void main (String[] args)
5. {
6. Scanner input = new Scanner(System.in);
7. int sum = 0; // Cumulative sum
8. int size; // Number of integers to add
9. int number; // holds the next integer to be added
10. System.out.print("How many numbers would you like to add? ");
11. size = input.nextInt();
12. System.out.println("Enter the " + size + " numbers");
13. for (int count =1; count<=size; count++) // for i=1 to count
14. {
15. number =input.nextInt(); // read the next integer
16. sum +=number; // add the current integer to sum
17. }
18. System.out.println("Sum: " + sum);
19. }
20. }
```

Output
How many numbers would you like to add? 4

Enter the 4 numbers 3, 5, 7, 9
Sum: 24

Discussion The “header” of the for statement, displayed on line 13, may appear a bitdaunting at first glance. Notice that the header consists of three parts:
1. the initialization statement, int count= 1 ,
2. the loop condition (a boolean expression), c ount<= size , and
3. the update statement , count++.

Without examining the body of a for loop, you can understand its termination structure.
A for loop gathers this information in one place:
for (initialization; loop condition; update statement)
The while loop and do-while loop scatter this information throughout the body of the loop.

Nested Loops

It should come as no surprise that loops may be nested within loops.

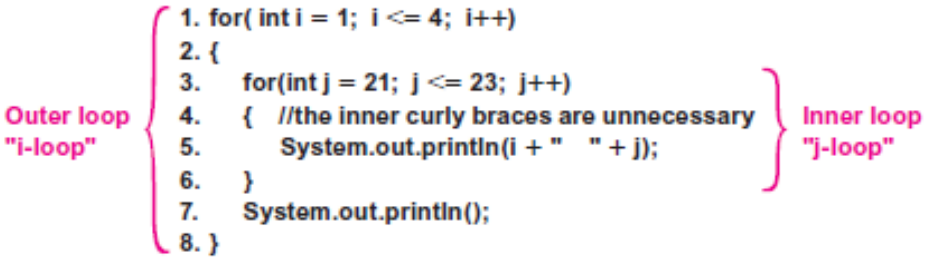


Figure 1.28 The nested loops

Notice that the inner “ j -loop” (lines 3 through 6) is nested within the outer “ i- loop” (lines 1 through 8). For each value of i (1, 2, 3, and 4), the j-loop executes once. Consequently, the println statement on line 5 executes 4 * 3 =12 times. The empty println statement (line 7) is not part of the inner loop, so this statement, which prints a blank line, executes just four times, once for each value of i.

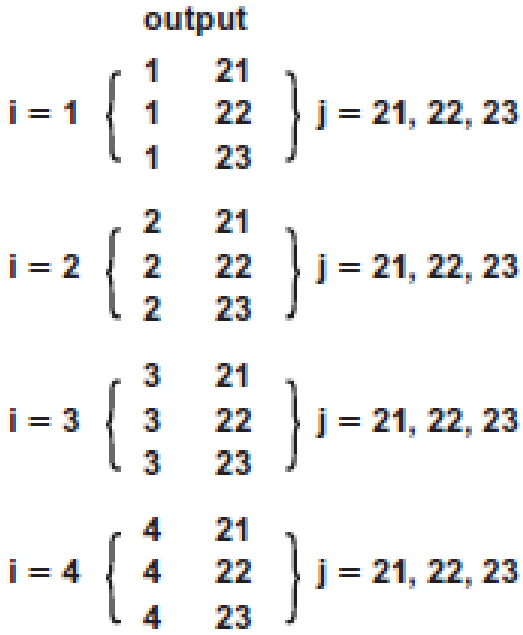


Figure 1.29 Nested while loop

Conclusion

The unit has however examined the if statement which are conditioner statement as well as if else statement which makes us have alternative decision to our inte intended actions.
The next units would open our minds to a broader view of alternative decisions.
Your programs are now capable of making decisions, and Java provides you with several decision-making options: the if statement, the if-else statement, and the switch statement.
By nesting these selection statements, your programs can implement some rather complex logic, as you have seen in the program above.

Java provides three statements that effect repetition: the while statement, the do-while statement, and the for statement. All three statements are equally powerful, but each is best suited for specific kinds of applications. A loop that always executes at least once is usually implemented with a do-while statement, and one that may never execute with a while statement. A loop that counts iterations is usually constructed with a for statement. The choice is a matter of style, technique, and convenience. Repetition, however, is not a convenience but a programming necessity. Repetition allows programs to perform any task multiple times. With repetition and selection, your programs can implement most any complex algorithm. No other control structures are necessary. But as your programming tasks become more complex, so do your programs.

Unit summary

In this unit, you learned that:

- Conditional statements are used to allow selective execution of statements. The conditional statements in Java are:
    - if-else
    - switch-case
- Looping statements are used when you want a section of a program to be repeated a certain number of times. Java offers the following looping statements:
    - for
    - while
    - do-while
- The for loop checks the validity of a condition first and then executes the body of the loop.
- The while loop checks for a condition before executing the body of the loop.
- In the do-while loop, first the body of the loop is executed and then the conditional expression is evaluated.
- The break and continue statements are used to control the program flow within a loop.

Assessment

Exercise 1

1. **True or False**

If false, give an explanation.

a. Every if clause has a matching else clause.

b. By default, an else clause is paired with the closest if clause.

c. switch (x> 5) causes a syntax error.

d. The case values of a switch statement cannot be of type double .

e. Every if clause is followed by a block.

f. A semicolon placed after an if clause causes a syntax error.

g. Omitting the curly braces that enclose the block of an if clause causes a syntax error.

h. Omitting parentheses that enclose the boolean expression of an if clause causes a syntax error.

i. Every switch statement can be directly converted to an else-if construction.

j. Every else-if construction can be directly converted to a switch statement.

k. Every case of a switch statement must include a break statement.

l. The default case of a switch statement is optional.

m. if statements may be nested within other if statements.

1. What's the Output?

Determine the output of the following three code segments:

(a) int a= 3;

```
if (a++== 3 )
System.out.println("Three");
else
System.out.println("Four");
(b) int a = 3;
if (++a == 3 )
System.out.println("Three");
else
System.out.println("Four");
(c) int a = 3;
a = a++;
if (a== 3 )
System.out.println("Three");
else
System.out.println("Four");
```

1. What's the Output?

Determine the output of the following poetic switch statement or point out the errors.

```
int a= 3;
switch (a)
{
case 1: System.out.println(" Once upon a midnight dreary, while I pondered weak and weary, );
case 2: System.out.println(" Over many a quaint and curious volume of forgotten lore, ");
case 3: System.out.println(" While I nodded, nearly napping, suddenly there came a tapping, );
case 4: System.out.println(" As of someone gently rapping, rapping at my chamber door ");
case 5: System.out.println(" Tis some visitor, I muttered, tapping at my chamber door, ");
default: System.out.println(" Only this, and nothing more. ");
```

Programming Exercises 2

1. **Sort Three**

Write a program that accepts three integers and displays the numbers in order from lowest to highest.

2. **Taxes**

Write a program that calculates the Minnesota state income tax according to the following rules:

Income Tax Rate

$0–$19,440 5.35%
$19,441–$63,860 7.05%
Over $63,860 7.85%
All data are type double.

Exercises 3

1. **True or False**

If false, give an explanation.

a. To implement a loop that always repeats 100 times, it is easier to use a for statement than a while statement.

b. Any operation that you can perform with a for statement you can also implement with a while statement.

c. Any operation that you can perform with a while statement you can also accomplish with a for statement.

d. A while statement always executes the loop body at least once.

e. You cannot nest a for loop within a while loop.

f. The data type of condition in while condition must be boolean .

g. Using the number 0 as a sentinel value is one way to signal the end of a list of integers.

h. The nesting depth of for loops is limited to at most three.

i. The statement

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

results in an infinite loop.

j. The statement

```
for (int i= 1; i<= 0; i++)
{i= i-1;}
```

results in an infinite loop.

Programming Exercises 4

1. **Credit Card Revisited**

Rewrite Example 5.8, using a for loop index that increases the loop counter by two with each iteration, that is, use a loop such as the following

for (int i = 1; i < MAX_DIGITS; i += 2) {...}.Why might this improve the performance of the

program?

2. **A Bank Account Record**

Write a program that reads a list of numbers representing deposits to and withdrawals from a savings account. Positive entries represent deposits and the negative entries withdrawals. Your program should calculate the sum of all deposits and the sum of all withdrawals. Use the sentinel zero to signal the end of the data.

3. **Prime Numbers**

Write a program that accepts an integer n and displays all the prime numbers between 2 and n. A prime number is a positive integer divisible only by itself and 1.

4. Perfect Numbers

A perfect number, p, is a positive integer that equals the sum of its divisors, excluding p itself. For example, 6 is a perfect number because the divisors of 6(1, 2, and 3) sum to 6. Write a program that prints all perfect numbers less than 1000. There are not many!

5. General Average

Write a program that calculates the average of n test scores, such that each score is an integer in the range 0 through 100. Your program should first prompt for an integer n and then request n scores. Your program should also check for invalid data. If a user enters a number outside the correct range, the program should prompt for another value. Round the average to the closest integer.

VIDEO

http://tinyurl.com/ycmogrz8

http://tinyurl.com/ycjt5tan

Unit 5

Objects and Classes

We will begin with taking a deeper look at building classes, controlling access to members of a class and creating constructors. We discuss composition—a capability that allows a class to have references to objects of other classes as members. We re-examine the use of *set* and *get* methods. The unit also discusses static class members and final instance variables in detail. Finally, we explain how to organize classes in packages to help manage large applications and promote reuse, and then show a special relationship between classes in the same package.

This unit continues our discussion of object-oriented programming (OOP) by introducing one of its primary capabilities inheritances, which is a form of software reuse in which a new class is created by absorbing an existing class's members and embellishing them with new or modified capabilities. With inheritance, you can save time during program development by basing new classes on existing proven and debugged high-quality software. This also increases the likelihood that a system will be implemented and maintained effectively.

When creating a class, rather than declaring completely new members, you can designate that the new class should inherit the members of an existing class. The existing class is called the superclass, and the new class is the subclass.(The C++ programming language refers to the superclass as the base class and the subclass as the derived class.) Each subclass can become a superclass for future subclasses.

A subclass can add its own fields and methods. Therefore, a subclass is more specific than its superclass and represents a more specialized group of objects. The subclass exhibits the behaviors of its superclass and can modify those behaviours so that they operate appropriately for the subclass. This is why inheritance is sometimes referred to as specialization.

The direct superclass is the superclass from which the subclass explicitly inherits. An indirect superclass is any class above the direct superclass in the class hierarchy, which defines the inheritance relationships between classes. In Java, the class hierarchy begins with Class Object (in package java.lang), which *every* class in Java directly or indirectly extends(or "inherits from"). Java supports only single inheritance, in which each class is derived from exactly *one* direct superclass. Unlike C++, Java does *not* support multiple inheritance (which occurs when a class is derived from more than one direct superclass). Object-Oriented Programming: Polymorphism, explains how to use Java interfaces to realize many of the benefits of multiple inheritance while avoiding the associated problems.

We distinguish between the is-a relationship and the has-a relationship. Is-*a* represents inheritance. In an is-a relationship, an object of a subclass can also be treated as an object of its superclass—e.g., a car *is a* vehicle.

By contrast, *has-a* represents composition. In a *has-a* relationship, an object contains as members references to other objects e.g., a car has a steering wheel (and a car object has a reference to a steering-wheel object).

New classes can inherit from classes in class libraries. Organizations develop their own class libraries and can take advantage of others available worldwide. Someday, most new software likely will be constructed from standardized reusable components, Just as automobiles and most computer hardware are constructed today. This will facilitate the development of more powerful, abundant and economical software.

Outcomes

Upon completion of this unit you will be able to:

- Encapsulation and data hiding.
- To use keyword this.
- To use static variables and methods.
- To import static members of a class.
- To use the enum type to create sets of constants with unique identifiers.
- To declare enum constants with parameters.
- To organize classes in packages to promote reuse.
- In this unit you will also learn:
- How inheritance promotes software reusability.
- The notions of superclasses and subclasses and the relationship between them.
- To use keyword extends to create a class that inherits attributes and behaviors from another class.
- To use access modifier protected to give subclass methods access to superclass members.
- To access superclass members with super.
- How constructors are used in inheritance hierarchies.
- The methods of class Object, the direct or indirect superclass of all classes

Terminology

Method:	In object-oriented programming, a procedure that is executed when an object receives a message. A method is really the same as a *procedure*, *function*, or *routine* in procedural programming languages.
Classes:	In object-oriented programming, a category of objects. For example, there might be a class called shape that contains objects which are circles, rectangles, and triangles.
Inheritance**:**	In object-oriented programing (OOP) inheritance is a feature that represents the "is a" relationship between different classes. Inheritance allows a class to have the same behavior as another class and extend or tailor that behavior to provide special action for specific needs

Polymorphism**:** In object-oriented programming, polymorphism refers to a programming language's ability to process objects depending on their class.

Controlling Access to Members

Java provides access specifiers and modifiers to decide which part of the class, such as data members and methods will be accessible to other classes or objects and how the data members are used in other classes and objects. The access specifiers public and private control access to a class's variables and methods.

As we stated earlier, the primary purpose of public methods is to present to the class's clients a view of the services the class provides (the class's public interface). Clients need not be concerned with how the class accomplishes its tasks. For this reason, the class's private variables and private methods (i.e., its implementation details) are not accessible to its clients.

The program in example 1 below demonstrates that private class members are not accessible outside the class.

Example 1

```
1 // MemberAccessTest.java
2 // Private members of class Time1 are not accessible.
3 public classMemberAccessTest
4 {
5 public static void main( String[] args )
6 {
7 Time1 time = new Time1(); // create and initialize Time1 object
8
9
10
11
12 } // end main
13 } // end class MemberAccessTest
```

```
time.hour = 7; // error: hour has private access in Time1
time.minute = 15; // error: minute has private access in Time1
time.second = 30; // error: second has private access in Time1
MemberAccessTest.java:9: hour has private access in Time1
time.hour = 7; // error: hour has private access in Time1
MemberAccessTest.java:10: minute has private access in Time1
time.minute = 15; // error: minute has private access in Time1
MemberAccessTest.java:11: second has private access in Time1
time.second = 30; // error: second has private access in Time1
```

Overloaded Constructors

Overloaded constructors enable objects of a class to be initialized in different ways. The compiler differentiates overloaded constructors by

their signatures. To call one constructor of a class from another of the same class, you can use the 'this' keyword followed by parentheses containing the constructor arguments. Such a constructor call must appear as the first statement in the constructor's body.

Default and No-Argument Constructors

If no constructors are provided in a class, the compiler creates a default constructor.
If a class declares constructors, the compiler will not create a default constructor. In this case, you must declare a no-argument constructor if default initialization is required.

Set and Get Methods

Setmethods are commonly called mutator methods because they typically change a value.
Getmethods are commonly called accessor methods or query methods. A predicate method tests whether a condition is true or false.

Composition

A class can have references to objects of other classes as members. This is called composition and is sometimes referred to as a *has-a* relationship.

Enumerations

All enum types are reference types. An enum type is declared with an enum declaration, which is a comma-separated list of enum constants. The declaration may optionally include other components of traditional classes, such as constructors, fields and methods. enum constants are implicitly final, because they declare constants that should not be modified. enum constants are implicitly static. Any attempt to create an object of an enum type with operator new results in a compilation error.

enum constants can be used anywhere constants can be used, such as in the case labels of switch statements and to control enhanced for statements. Each enum constant in an enum declaration is optionally followed by arguments which are passed to the enum constructor. For every enum, the compiler generates a static method called values that returns an array of the enum's constants in the order in which they were declared.

EnumSet static method range receives the first and last enum constants in a range and returns an EnumSet that contains all the constants between these two constants, inclusive.

Static Class Members

A static variable represents class wide information that's shared among the class's objects.
Static variables have class scope. A class's public static members can be accessed through a reference to any object of the class, or they can be accessed by qualifying the member name with the class name and a dot (.). Client code can access a class's private static class members only through methods of the class. Static class members exist as soon as the class is loaded into memory.

A method declared static cannot access non-static class members, because a static method can be called even when no objects of the class have been instantiated. The this reference cannot be used in a static method.

Static Import

A static import declaration enables you to refer to imported static members without the class name and a dot (.). A single static import declaration imports one static member, and a static import on demand imports all static members of a class.

Final Instance Variables

In the context of an application, the principle of least privilege states that code should be granted only the amount of privilege and access that it needs to accomplish its designated task.

Keyword final specifies that a variable is not modifiable. Such variables must be initialized when they're declared or by each of a class's constructors.

Super Classes and Sub Classes

Java supports inheritance that enables a class to inherit data members and methods from another class. Inheritance enables you to reuse the functionalities and capabilities of the existing class by extending a new class from the existing class and adding new features to it. In inheritance, the class that inherits the data members and methods from another class is known as the *subclass*. The class from which the subclass inherits is known as the *superclass*.

An object of one class *is an* object of another class as well." For example, a Car Loan *is a* Loan as are Home Improvement Loans and Mortgage Loans. Thus, in Java, class Car Loan can be said to inherit from class Loan. In this context, class Loan is a superclass and class CarLoan is a subclass. A CarLoan*is a* specific type of Loan, but it's incorrect to claim that every Loan *is a* CarLoan the Loan could be any type of loan.

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

Figure 1.30 Inheritance Example

Because every subclass object *is an* object of its superclass, and one superclass can have many subclasses, the set of objects represented by a superclass is often larger than the set of objects represented by any of its subclasses. For example, the superclass Vehicle represents all vehicles, including cars, trucks, boats, bicycles and so on. By contrast, subclass Car represents a smaller, more specific subset of vehicles.

University Community Member Hierarchy

Inheritance relationships form treelike hierarchical structures. A superclass exists in a hierarchical relationship with its subclasses. Let's develop a sample class hierarchy (Figure. 1.20), also called an **inheritance hierarchy**. A university community has thousands of members, including employees, students and alumni. Employees are either faculty or staff members. Faculty members are either administrators (e.g., deans and department chairpersons) or teachers. The hierarchy could contain many other classes. For example, students can be graduate or undergraduate students. Undergraduate students can be freshmen, sophomores,
juniors or seniors.

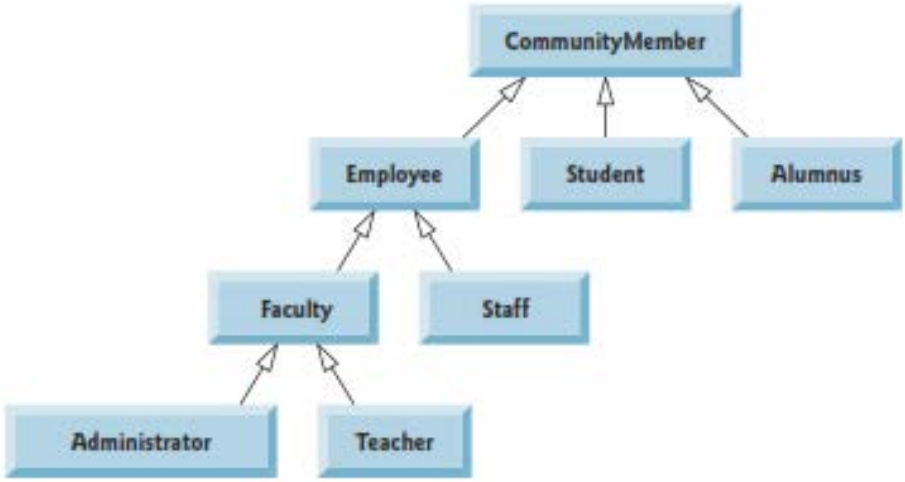


Figure 1.31: Inheritance hierarchy for university community members

Each arrow in the hierarchy represents an *is-a* relationship. As we follow the arrows upward in this class hierarchy, we can state, for instance, that "an Employee *is a* CommunityMember" and "a Teacher *is a* Faculty member." Community Member is the direct superclass of Employee, Student and Alumnus and is an indirect superclass of all the other classes in the diagram. Starting from the bottom, you can follow the arrows and apply the *is-a* relationship up to the topmost superclass. For example, an Administrator *is a* Faculty member, *is an* Employee, *is a* CommunityMember and, of course, *is an* Object.

Relationship between Super Classes and Sub Classes

We now use an inheritance hierarchy containing types of employees in a company's payroll application to discuss the relationship between a superclass and its subclass. In this company, commission employees (who will be represented as objects of a superclass) are paid a percentage of their sales, while base-salaried commission employees (who will be represented as objects of a subclass) receive a base salary *plus* a percentage of their sales.

We divide our discussion of the relationship between these classes into five examples. The first declares classCommissionEmployee, which directly inherits from class Object and declares as private instance variables a first name, last name, social security number, commission rate and gross (i.e., total) sales amount.

The second example declares classBasePlusCommissionEmployee, which also directly inherits from class Object and declares as private instance variables a first name, lastname, social security number, commission rate, gross sales amount *and* base salary. We create this class by *writing every line of code* the class requires—we'll soon see that it's much more efficient to create it by inheriting from class CommissionEmployee.

The third example declares a newBasePlusCommissionEmployee class that *extends* class CommissionEmployee (i.e., aBasePlusCommissionEmployee *is a* CommissionEmployeewho also has a base salary). In this example, classBasePlusCommissionEmployee attempts to accessclassCommissionEmployee's private members — this results in compilation errors, because the subclass cannot access the superclass's private instance variables.

The fourth example shows that if CommissionEmployee's instance variables are declared as protected, the BasePlusCommissionEmployee subclass can access that data directly. Both BasePlusCommissionEmployee classes contain identical functionality, but we show how the inherited version is easier to create and manage. After we discuss the convenience of using protected instance variables, we create the fifth example, which sets the CommissionEmployee instance variables back to private to enforce good software engineering.

Then we show how the BasePlusCommissionEmployeesubclass can use CommissionEmployee's public methods to manipulate (in a controlled manner) the private instance variables inherited from CommissionEmployee.

Overview of Class Commission Employee's Methods and Instance Variables

Class CommissionEmployee's public services include a constructor (lines 13–22) and Methods earnings (lines 93–96) and toString (lines 99–107). Lines 25–90 declare public *Get* and *set* methods for the class's instance variables (declared in lines 6–10) first-Name, lastName, socialSecurityNumber, grossSales and commissionRate. The class declares its instance variables as private, so objects of other classes cannot directly access these variables. Declaring instance variables as private and providing *get* and *set* method so manipulate and validate them helps enforce good software engineering.

Methods setGrossSales and setCommissionRate, for example, validate their arguments before assigning the values to instance variables grossSales and commissionRate. In a real-world, business critical application, we'd also perform validation in the class's other *set* methods.

```
1 // Program 1 CommissionEmployee.java
2 // CommissionEmployee class represents an employee paid a
3 // percentage of gross sales.
4
5 {
6-9//empty pscaes
public class CommissionEmployee extends Object
10----23//empty spaces
24 // set first name
25 public void setFirstName( String first )
26 {
27 firstName = first; // should validate
28 } // end method setFirstName
29
30 // return first name
31 public String getFirstName()
32 {
33 return firstName;
34 } // end method getFirstName
35
36 // set last name
37 public void setLastName( String last )
38 {
39 lastName = last; // should validate
40 } // end method setLastName
```

```
41
42 // return last name
43 public String getLastName()
44 {
private String firstName;
private String lastName;
private String socialSecurityNumber;
private double grossSales; // gross weekly sales
private double commissionRate; // commission percentage
// five-argument constructor
publicCommissionEmployee( String first, String last, String ssn,
double sales, double rate )
{
// implicit call to Object constructor occurs here
firstName = first;
lastName = last;
socialSecurityNumber = ssn;
setGrossSales( sales ); // validate and store gross sales
setCommissionRate( rate ); // validate and store commission rate
} // end five-argument CommissionEmployee constructor
45 return lastName;
46 } // end method getLastName
47
48 // set social security number
49 public void setSocialSecurityNumber( String ssn )
50 {
51 socialSecurityNumber = ssn; // should validate
52 } // end method setSocialSecurityNumber
53
54 // return social security number
55 public String getSocialSecurityNumber()
56 {
57 return socialSecurityNumber;
58 } // end method getSocialSecurityNumber
59
60 // set gross sales amount
61 public void setGrossSales( double sales )
62 {
63 if ( sales>= 0.0 )
64 grossSales = sales;
65 else
66 throw new IllegalArgumentException(
67 "Gross sales must be >= 0.0" );
68 } // end method setGrossSales
69
70 // return gross sales amount
71 public double getGrossSales()
72 {
73 return grossSales;
74 } // end method getGrossSales
75
```

```
76 // set commission rate
77 public void setCommissionRate( double rate )
78 {
79 if ( rate> 0.0 && rate < 1.0 )
80 commissionRate = rate;
81 else
82 throw new IllegalArgumentException(
83 "Commission rate must be > 0.0 and < 1.0" );
84 } // end method setCommissionRate
85
86 // return commission rate
87 public double getCommissionRate()
88 {
89 return commissionRate;
90 } // end method getCommissionRate
91
92
93
94
95
96
// calculate earnings
public double earnings()
{
returncommissionRate * grossSales;
} // end method earnings
97-107 //empty spaces
108 } // end class CommissionEmployee
// return String representation of CommissionEmployee object
@Override // indicates that this method overrides a superclass method
public String toString()
{
returnString.format( "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f",
"commission employee", firstName, lastName,
"social security number", socialSecurityNumber,
"gross sales", grossSales,
"commission rate", commissionRate );
} // end method toString
```

Constructors in Subclasses

As we explained in the preceding section, instantiating a subclass object begins a chain of constructor calls in which the subclass constructor, before performing its own tasks, invokes its direct superclass's constructor either explicitly via the super reference or implicitly calling the superclass's default constructor or no-argument constructor. Similarly, if the superclass is derived from another class—as is, of course, every class except Object the superclass constructor invokes the

constructor of the next class up the hierarchy, and so on. The last constructor called in the chain is *always* the constructor for class Object. The original subclass constructor's body finishes executing *last*. Each superclass's constructor manipulates the superclass instance variables that the subclass object inherit

Conclusion

In this unit, we presented additional class concepts. The Time class case study presented a complete class declaration consisting of private data, overloaded public constructors for initialization flexibility, set and get methods for manipulating the class's data, and methods that returned String representations of a Time object in two different formats. You also learned that every class can declare a to String method that returns a String representation of an object of the class and that method to String can be called implicitly whenever an object of a class appears in the code where a String is expected. You learned that the this reference is used implicitly in a class's non-static methods to access the class's instance variables and other non static methods. You also saw explicit uses of the this reference to access the class's members (including shadowed fields) and how to use keyword this in a constructor to call another constructor of the class.

This unit introduced inheritance—the ability to create classes by absorbing an existing class's members and embellishing them with new capabilities. You learned the notions of superclasses and subclasses and used keyword extends to create a subclass that inherits members from a superclass. We showed how to use the @ Override annotation to prevent unintended overloading by indicating that a method overrides a superclass method. We introduced the access modifier protected; subclass methods can directly access protected superclass members.

Unit summary

In this unit, you learned that:

- Inheritance is the concept of extending data members and methods of a superclass in a subclass.
- You can derive data members and methods from a single superclass that is a subclass of another superclass.
- Java does not support multiple inheritance directly.
- You can use the concept of method overriding to override the superclass method with the subclass method having same names.
- You can use super to access overridden superclass members.
- Constructors are used in inheritance hierarchies.
- Methods of class Object, the direct or indirect superclass of all Java classes.

Assessment

Fill in the blanks in each of the following statements:
a) When compiling a class in a package, the javac command-line option---------- specifies where to store the package and causes the compiler to create the package's directories if they do not exist.
b) String class static method---------- is similar to method System.out.printf, but returns a formatted String rather than displaying a String in a command window.
c) If a method contains a local variable---------- with the same name as one of its class's fields, the local variable the field in that method's scope.
d) The----------method is called by the garbage collector just before it reclaims an object's memory.
e) A(n)-------declaration specifies one class to import.
f) If a class declares constructors, the compiler will not create a(n)----------
g) An object's----------method is called implicitly when an object appears in code where a String is needed.
h) Get methods are commonly called--------or-------i) A(n)method tests whether a condition is true or false.
i) For every enum, the compiler generates a static method called---------- that returns an array of the enum's constants in the order in which they were declared.
j) Composition is sometimes referred to as a(n)----------relationship.

Fill in the blanks in each of the following statements:
a) ----------is a form of software reusability in which new classes acquire the members of existing classes and embellish those classes with new capabilities.
b) A superclass's---------members can be accessed in the superclass declaration *and in* subclass declarations.
c) In a(n)----------relationship, an object of a subclass can also be treated as an object of its superclass.
d) In a(n)----------relationship, a class object has references to objects of other classes as members.
e) In single inheritance, a class exists in a(n)---------relationship with its subclasses.
f) A superclass's--------members are accessible anywhere that the program has a reference to an object of that superclass or to an object of one of its subclasses.
g) When an object of a subclass is instantiated, a superclass-------is called implicitly or explicitly.
h) Subclass constructors can call superclass constructors via the---------- keyword.

VIDEO

http://tinyurl.com/y7n43vrx

shttp://tinyurl.com/y7t95x2b

http://tinyurl.com/yauoz3wa

Unit 6

Polymorphism

We continue our study of object-oriented programming by explaining and demonstrating **polymorphism** with inheritance hierarchies. Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism occurs when a parent class reference is used to refer to a child class object. Any Java object that can pass more than one IS-A test is considered to be polymorphic.

Consider the following example of polymorphism. Suppose we create a program that simulates the movement of several types of animals for a biological study. Classes Fish, Frog and Bird represent the types of animals under investigation. Imagine that each class extends superclass Animal, which contains a method move and maintains an animal's current location as *x*-*y* coordinates. Each subclass implements method move. Our program maintains an Animal array containing references to objects of the various Animal subclasses.

To simulate the animals' movements, the program sends each object the *same* message once per second—namely, move. Each specific type of Animal responds to a move message in its own way—a Fish might swim three feet, a Frog might jump five feet and a Bird might fly ten feet. Each object knows how to modify its *x*-*y* coordinates appropriately for its *specific* type of movement. Relying on each object to know how to "do the right thing" (i.e., do what is appropriate for that type of object) in response to the same method call is the key concept of polymorphism. The same message (in this case, move) sent to a variety of objects has "many forms" of results—hence the term polymorphism.

Outcomes

Upon completion of this unit you will be able to:

- explain the concept of polymorphism.
- use overridden methods to effect polymorphism.
- distinguish between abstract and concrete classes.
- declare abstract methods to create abstract classes.
- Understand how polymorphism makes systems extensible and maintainable.
- determine an object's type at execution time.
- declare and implement interfaces.

Terminology

Static: Generally refers to elements of the Internet or computer programming that are fixed and not capable of action or change. The opposite of static is *dynamic*.

Bind:	Bind means to assign a value to a symbolic placeholder. During compilation, for example, the compiler assigns symbolic addresses to some variables and instructions.

Polymorphism Examples

We now consider several additional examples of polymorphism.

Quadrilaterals
If class Rectangle is derived from class Quadrilateral, then a Rectangle object is a more specific version of a Quadrilateral. Any operation (e.g., calculating the perimeter or the area) that can be performed on a Quadrilateral can also be performed on a Rectangle. These operations can also be performed on other Quadrilaterals, such as Squares, Parallelograms and Trapezoids.

The polymorphism occurs when a program invokes a method through a superclass Quadrilateral variable—at execution time, the correct subclass version of the method is called, based on the type of the reference stored in the superclass variable.

Demonstrating Polymorphic Behaviour

BasePlusCommissionEmployee as discussed earlier present objects by using references to them to invoke their methods—we aimed superclass variables at superclass objects and subclass variables at subclass objects. These assignments are natural and straight forward superclass variables are *intended* to refer to superclass objects, and subclass variables are *intended* to refer to subclass objects. However, as you'll soon see, other assignments are possible. In the next example, we aim a *superclass* reference at *a subclass* object. We then show how invoking a method on a subclass object via a superclass reference invokes the *subclass* functionality—the type of the *referenced object*, not the type of the *variable*, determines which method is called. This example demonstrates that *an object of a subclass can be treated as an object of its superclass*, enabling various interesting manipulations. A program can create an array of superclass variables that refer to objects of many subclass types. This is allowed because each subclass object *is an* object of its superclass.

For instance, we can assign the reference of a BasePlusCommissionEmployee object to a superclass CommissionEmployee variable, because a BasePlusCommissionEmployee *is a* CommissionEmployee — we can treat a BasePlusCommissionEmployee as a CommissionEmployee. As you will learn later in the unit, you *cannot treat a superclass object as a subclassobject*, because a superclass object is *not* an object of any of its subclasses.

For example, we cannot assign the reference of a CommissionEmployee object to a subclass BasePlusCommissionEmployeevariable, because a CommissionEmployee is *not* a BasePlusCommissionEmployee, a CommissionEmployee does *not* have a baseSalary instance variable and does*not*have methods setBaseSalary and getBaseSalary.The*is-a* relationship applies only *up the hierarchy* from a subclass to its direct (and indirect) superclasses, and *not* vice versa(i.e., not down the hierarchy from a superclass to its subclasses).

The Java compiler *does* allow the assignment of a superclass reference to a subclass variable if we explicitly cast the superclass reference to the subclass type a technique we discuss earlier. Why would we ever want to perform such an assignment? A superclass reference can be used to invoke only the methods declared in the superclass attempting to invoke subclass-only methods through a superclass reference results in compilation errors. If a program needs to perform a subclass-specific operation on a subclass object referenced by a superclass variable, the program must first cast the superclass reference to a subclass reference through a technique known as **downcasting**. This enables the program to invoke subclass methods that are not in the superclass.

The example in program 10.1 demonstrates three ways to use superclass and subclass variables to store references to superclass and subclass objects. The first two are straight forward as discussed earlier; we assign a superclass reference to a superclass variable, and a subclass reference to a subclass variable. Then we demonstrate the relationship between subclasses and superclasses (i.e., the *is-a* relationship) by assigning a subclass reference to a superclass variable. This program uses classesCommissionEmployee and BasePlusCommissionEmployee from earlier presented code in unit 1of module 5

1 // Program 10.1: PolymorphismTest.java
2 // Assigning superclass and subclass references to superclass and
3 // subclass variables.
4
5 public classPolymorphismTest
6 {
7 public static void main(String[] args)
8 {
9-17 //empty spaces
18 // invoke toString on superclass object using superclass variable// assign superclass reference to superclass variable CommissionEmployeecommissionEmployee = new CommissionEmployee("Sue", "Jones", "222-22-2222", 10000, .06);
// assign subclass reference to subclass variableBasePlusCommissionEmployeebasePlusCommissionEmployee =new BasePlusCommissionEmployee("Bob", "Lewis", "333-33-3333", 5000, .04, 300);
19 System.out.printf("%s %s:\n\n%s\n\n",

```
20 "Call CommissionEmployee'stoString with superclass reference ",
21 "to superclass object",);
22
23 // invoke toString on subclass object using subclass variable
24 System.out.printf( "%s %s:\n\n%s\n\n",
25 "Call BasePlusCommissionEmployee'stoString with subclass",
26 "reference to subclass object",
27 );
28
29 // invoke toString on subclass object using superclass variable
30
31
32 System.out.printf( "%s %s:\n\n%s\n",
33 "Call BasePlusCommissionEmployee'stoString with superclass",
34 "reference to subclass object",);commissionEmployee2.toString()
35 } // end main
36 } // end class PolymorphismTest
```

```
Call CommissionEmployee'stoString with superclass reference to superclassobject:commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06
Call BasePlusCommissionEmployee'stoString with subclass reference tosubclassobject:base-salaried commission employee: Bob Lewissocialsecurity number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00
Call BasePlusCommissionEmployee'stoString with superclass reference tosubclass object:
base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00
```

Abstract Classes and Methods

When we think of a class, we assume that programs will create objects of that type. Some-times it’s useful to declare classes called **abstract classes** for which you *never* intend to create objects. Because they’re used only as superclasses in inheritance hierarchies, we refer to them as **abstract superclasses**. These classes cannot be used to instantiate objects, because, as we’ll soon see, abstract classes are *incomplete*. Subclasses must declare the “missing pieces” to become “concrete” classes, from which you can instantiate objects. Otherwise, these subclasses, too, will be abstract.

Purpose of Abstract Classes

An abstract class's purpose is to provide an appropriate superclass from which other classes can inherit and thus share a common design. In the Shape hierarchy discussed earlier, for example, subclasses inherit the notion of what it means to be a Shape—perhaps common attributes such as location, color and borderThickness, and behaviors such as draw, move, resize and changeColor. Classes that can be used to instantiate objects are called **concreteclasses**. Such classes provide implementations of *every* method they declare (some of the implementations can be inherited). For example, we could derive concrete classes Circle, Square and Triangle from abstract superclass TwoDimensionalShape. Similarly, we could derive concrete classes Sphere, Cube and Tetrahedron from abstract superclass **ThreeDimensionalShape.**

Abstract superclasses are *too general* to create real objects—they specify only what is common among subclasses. We need to be more *specific* before we can create objects. For example, if you send thedraw message to abstract class TwoDimensionalShape, the class knows that two-dimensional shapes should be drawable, but it does not know what specific shape to draw, so it cannot implement a realdraw method. Concrete classes provide the specifics that make it reasonable to instantiate objects.

Not all hierarchies contain abstract classes. However, you'll often write client code that uses only abstract superclass types to reduce the client code's dependencies on a range of subclass types. For example, you can write a method with a parameter of an abstract superclass type. When called, such a method can receive an object of any concrete class that directly or indirectly extends the superclass specified as the parameter's type. Abstract classes sometimes constitute several levels of a hierarchy. For example, the Shape hierarchy of above begins with abstract class Shape.
On the next level of the hierarchy are *abstract* classes TwoDimensionalShape and ThreeDimensionalShape. The next level of the hierarchy declares *concrete* classes for TwoDimensionalShapes(Circle, Square and Triangle) and for ThreeDimensionalShapes(Sphere, Cube and Tetrahedron).

Declaring an Abstract Class and Abstract Methods

You make a class abstract by declaring it with keyword abstract. An abstract class normally contains one or more **abstract methods**. An abstract method is one with keyword abstract in its declaration, as in public abstract void draw(); // abstract method Abstract methods do *not* provide implementations. A class that contains *any* abstract methods must be explicitly declared abstract even if that class contains some concrete(nonabstract) methods. Each concrete subclass of an abstract superclass also must provide concrete implementations of each of the

superclass's abstract methods. Constructors and static methods cannot be declared abstract. Constructors are not inherited, so an abstract constructor could never be implemented. Though non-private static methods are inherited, they cannot be overridden. Since abstract methods are meant tobe overridden so that they can process objects based on their types, it would not make sense to declare a static method as abstract.

Using Abstract Classes to Declare Variables

Although we cannot instantiate objects of abstract superclasses, you'll soon see that we *can* use abstract superclasses to declare variables that can hold references to objects of any concrete class derived from those abstract superclasses. Programs typically use such variables to manipulate subclass objects polymorphically. You also can use abstract superclass names to invoke static methods declared in those abstract superclasses. Consider another application of polymorphism. A drawing program needs to display many shapes, including types of new shapes that you will add to the system after writing the drawing program. The drawing program might need to display shapes, such as Circles, Triangles, Rectangles or others, that derive from abstract class Shape. The drawing programusesShape variables to manage the objects that are displayed. To draw any object inthis inheritance hierarchy, the drawing program uses a superclass Shape variable containing a reference to the subclass object to invoke the object's draw method. This method is declared abstract in superclass Shape, so each concrete subclass *must* implement method draw in a manner specific to that shape—each object in the Shape inheritance hierarchy *knows how to draw itself*. The drawing program does not have to worry about the type of each object or whether the program has ever encountered objects of that type.

Final Methods and Classes

We saw in earlier that variables can be declared final to indicate that theycannot be modified after they're initialized—such variables represent constant values. It'salso possible to declare methods, method parameters and classes with thefinal modifier.

Final Methods Cannot Be Overridden

A final **method** in a superclass *cannot* be overridden in a subclass—this guarantees that the final method implementation will be used by all direct and indirect subclasses in the hierarchy. Methods that are declared private are implicitly final, because it's not possible to override them in a subclass. Methods that are declared static are also implicitly final. A final method's declaration can never change, so all subclasses use the same method implementation, and calls to final methods are resolved at compile time—this is known as **static binding**.

Final Classes Cannot Be Superclasses

A final **class** that's declared final cannot be a superclass (i.e., a class cannot extend a final class).All methods in a final class are implicitly final. Class String is an example of a final class. If you were allowed to create a subclass of String, objects of that subclass could be used wherever Strings are expected. Since class String cannot be extended,

programs that use Strings can rely on the functionality of String objects as specified in the Java API. Making the class final also prevents programmers from creating subclasses that might bypass security restrictions.

Conclusion

This unit introduced polymorphism—the ability to process objects that share the same superclass in a class hierarchy as if they're all objects of the superclass. The unit discussed how polymorphism makes systems extensible and maintainable, then demonstrated how to use overridden methods to effect polymorphic behavior. We introduced abstract classes, which allow you to provide an appropriate superclass from which other classes can inherit. You learned that an abstract class can declare abstract methods that each subclass must implement to become a concrete class and that a program can use variables of an abstract class to invoke the subclasses' implementations of abstract methods in polymorphic manner.

Unit summary

In this unit, you learned that:

- To determine an object's type at execution time. We discussed the concepts of final methods and classes.
- Declaring and implementing an interface was another way to achieve polymorphic behavior.
- Classes, objects, encapsulation, inheritance, interfaces and polymorphism are the most essential aspects of object-oriented programming.

Assessment

Fill in the blanks in each of the following statements:
a) If a class contains at least one abstract method, it's a(n)----class.
b) Classes from which objects can be instantiated are called----classes.
c)-------involves using a superclass variable to invoke methods on superclass and subclass objects, enabling you to "program in the general."
d) Methods that are not interface methods and that do not provide implementations must be declared using keyword-------
e) Casting a reference stored in a superclass variable to a subclass type is called-------

Section II

1. How does polymorphism enable you to program "in the general"

rather than “in the specific”? Discuss the key advantages of programming “in the general.”
2. What are abstract methods? Describe the circumstances in which an abstract method would be appropriate.
3. How does polymorphism promote extensibility?
4. Discuss four ways in which you can assign superclass and subclass references tovariables of superclass and subclass types.
5. Compare and contrast abstract classes and interfaces. Why would you use an abstract class? Why would you use an interface?

http://tinyurl.com/yczs5ezx