JAVA Programming

FOR BEGINNERS

by DR DANNY POO

www.DrDannyPoo.com

JAVA Programming

Community Edition

Danny Poo

PREFACE

Who Should Read This Book
This book is designed for beginners to Java programming. It teaches the fundamentals of Java.

Topics covered include: the Java programming environment; the Java basic building blocks that include variables; data structures; data types and their declaration; expressions, statements, and operators; program flow control mechanisms; arrays; methods; inputs and outputs; class and objects; file handling.

How This Book Is Organized
This book is organized into nine chapters with each chapter building on what have been covered in the previous chapters.

Chapter 1: Getting Started
This chapter prepares the readers for Java programming with instructions on how to set up the Java programming environment.

Chapter 2: Basic Building Blocks
This chapter defines the vocabulary of the Java language and explains how the words and symbols can be put together to form the basic program structure in Java.

Chapter 3: Statements and Operators
This chapter explains the structure of a Java expression and how it can be used in conjunction with a semi-colon to form statements, the basic executable component of Java.

Chapter 4: Program Flow Controls
This chapter explains how to add program flow control mechanisms in a Java program. Statements can be put together and executed sequentially using the Sequence construct, selectively using the Selection construct, and repetitively using the Iteration construct.

Chapter 5: Arrays
This chapter explains the concept of an array as a data structure for storing and manipulating data of the same type as a collection. Details on how to manipulate a one-dimensional and two-dimensional array are included in this chapter.

Chapter 6: Methods
This chapter explains how a Java program can be more structured with the use of method – a collection of data and statements for performing a task.

Chapter 7: Class and Objects
Java is an object-oriented programming language. This chapter defines what a class and object is and provides a foundation for further discussion on object-oriented programming.

Chapter 8: Inputs and Outputs
This chapter explains how data input and output is carried out in Java.

Chapter 9: File Handling
This chapter discusses how to read and write data from text and binary files.

Enjoy!
Dr Danny Poo
www.DrDannyPoo.com

ABOUT THE AUTHOR

Dr. Danny Poo brings with him 40 years of Software Engineering and Information Technology and Management experience. A graduate from the University of Manchester Institute of Science and Technology (UMIST), England, Dr. Poo is currently an Associate Professor at the Department of Information Systems and Analytics, National University of Singapore.

A well-known speaker in seminars, Dr. Poo has conducted numerous in-house training and consultancy for organizations, both locally and regionally. His notable teaching credentials include • Data Strategy • Data StoryTelling • Data Visualisation • Big Data Analytics • Machine Learning • Data Management • Data Governance • Data Architecture • Capstone Projects for Business Analytics • Software Engineering • Server-side Systems Design and Development • Information Technology Project Management • Health Informatics • Healthcare Analytics • Health Informatics Leadership.

Dr. Poo has also published extensively in conferences and journals on Software Engineering and Information Management.

Dr. Poo was the founding Director of the Centre for Health Informatics. This Centre provides courses to train healthcare professionals in Health Informatics. Dr. Poo is instrumental in developing curriculum and courses for this Centre. In particular, he has delivered numerous rounds of Health Informatics course since 2012 and has trained as many as 1000 healthcare and IT professionals on this subject. Besides, he teaches a course on Healthcare Analytics to healthcare professionals since it started in May 2015. To date, he has run fourteen 3-full-days-sessions of this course since May 2015. This course continues to receive great interest from participants.

Dr. Poo has authored a number of books including "Java Programming", "Object-Oriented Programming", "Graphical User Interface Programming in Java", "Python Programming", and "Learn to Program Enterprise JavaBeans 3.0".

Dr. Poo has consulted for these companies • Deutsche Bank • Gemplus • Micron • NCR • PIL • PSA • Rhode-Schwarz • Standard Chartered Bank • Singapore Technologies Electronic • Monetary Authority of Singapore (MAS) • Infocomm Development Authority (IDA) • National Library Board (NLB) • Ministry of Manpower (MOM) • Nanyang Technological University (NTU) • Nanyang Polytechnic (NYP) • National University Hospital.

CHAPTER 1: GETTING STARTED

Java is a programming language developed by Sun Microsystems in 1995. It was created in response to the need to have a programming language that can be operated on multiple operating system platforms. The desire is to have a program written for the Microsoft Windows operating system to be executable also in any other operating system such as the Apple IOS, Unix, or Linux operating system.

Sun Microsystems' answer to this requirement is to generate intermediate byte code that is not tied to any operating system. Byte codes are understandable by JVMs (Java Virtual Machines) and there is a JVM type for each operating system available. A Java program written for and executable on a Microsoft Windows operating system needs not be re-compiled for execution on a Linux platform, neither does it need to be re-compiled for one on a Unix system, so long as an appropriate JVM is available.

Java bears some similarities to the C programming language (such as the language syntax and identifiers), since Java was created based on C. Java is also an object-oriented programming language. What this means is that a Java program is structured based on the concept of objects and classes. Detailed discussion of object-oriented programming will be deferred to a separate text.

1.1 Structure of a Java Program

How does a Java program look like? Code 1.1 shows a simple Java program. When this program is run, the line "How do you do?" is printed on the Windows DOS prompt.

Code 1.1: A Simple Java Program
```
class HowDoYouDo {

  public static void main (String[] args) {
    System.out.println("How do you do?");
  }
}
```

The statement that actually prints the line is:

```
System.out.println("How do you do?");
```

The System.out.println statement is enclosed within a procedural block called a method in Java. In general, a method is a grouping of data variables and statements. The method referred here is the main() method. When a method is called, all the statements in the method will be executed. The execution is carried out statement by statement until the end of the method is reached. The beginning and ending of a method is denoted by an opening brace ("{") and a closing brace ("}") respectively.

The main() method is enclosed within a class HowDoYouDo. A class is defined as a grouping of data variables and methods and has a general structure as shown in Code 1.2.

<classname> refers to the name of the class. Replace this with a class name of your choice. Typically, a class name should reflect the purpose of the class. One convention for naming a class is to use upper case for the first letter of the class name. For example, to name the HowDoYouDo class, use HowDoYouDo instead of howDoYouDo.

The keywords "public", "static", and "void" have their significance. "public" indicates the method main() can be called by another program outside of this class. "static" indicates that the method main() belongs to a class as opposed to belonging to an object. "void" indicates that the method main() does not return any value after it has completed its execution. These may not make much sense to you now but do not worry, we will discuss them further later in the book. For now, just make sure that these three keywords are present in front of the method name. The sequence in which they appear is not important.

Code 1.2: A General Class Structure
```
class <classname> {

  public static void main(String[] argv) {
    // put your statements here
  }
}
```

String[] argv (alternatively, it may be written as String argv[]) is a String array parameter definition. It is through this array that inputs entered via the Windows DOS prompt are captured and conveyed to the program. You may alternatively use String[] args in place of String[] argv. No other names for this String array are allowed.

The statement "// put your statements here" is a comment. We use comment to explain to other fellow programmers our intention. A comment is not executable. Java will ignore it during program execution.

1.2 How to Run a Java Program?

To run a Java program, the program must first be compiled into byte code. The Java byte code is understood and executable by a Java Virtual Machine (JVM).

We need to prepare the environment before we can run any Java program. The Java compiler, JVM and other essential code for compiling and running a Java program is available in the form of a Software Development Kit (SDK). The Java SDK is available for free from the Oracle Java website. At the website, you will notice a number of SDKs are available. For this book, we will need the Java SE (Standard Edition) Software Development Kit.

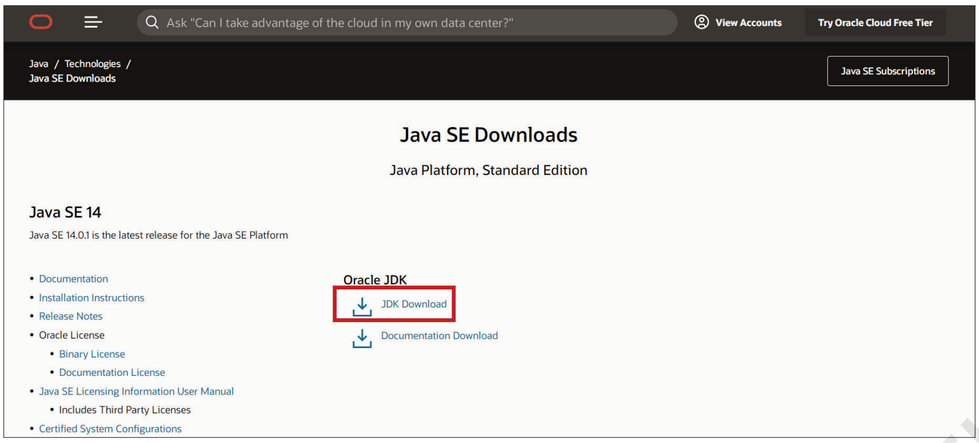
1.3 How to install the Java SE Software Development Kit (SDK)?

We need to download:

1. The Java SE (Standard Edition) Development Kit. The Java SE Development Kit (JDK) includes the Java Runtime Environment (JRE) and command-line development tools that are useful for developing Java applications.
2. The Java SE Documentation.

1.3.1 How to download the Java SE Development Kit?

Go to the Java download website:
http://www.oracle.com/technetwork/java/javase/downloads/index.html


Ask "Can I take advantage of the cloud in my own data center?"
View Accounts
Try Oracle Cloud Free Tier
Java / Technologies /
Java SE Downloads
Java SE Subscriptions

Java SE Downloads

Java Platform, Standard Edition

Java SE 14

Java SE 14.0.1 is the latest release for the Java SE Platform

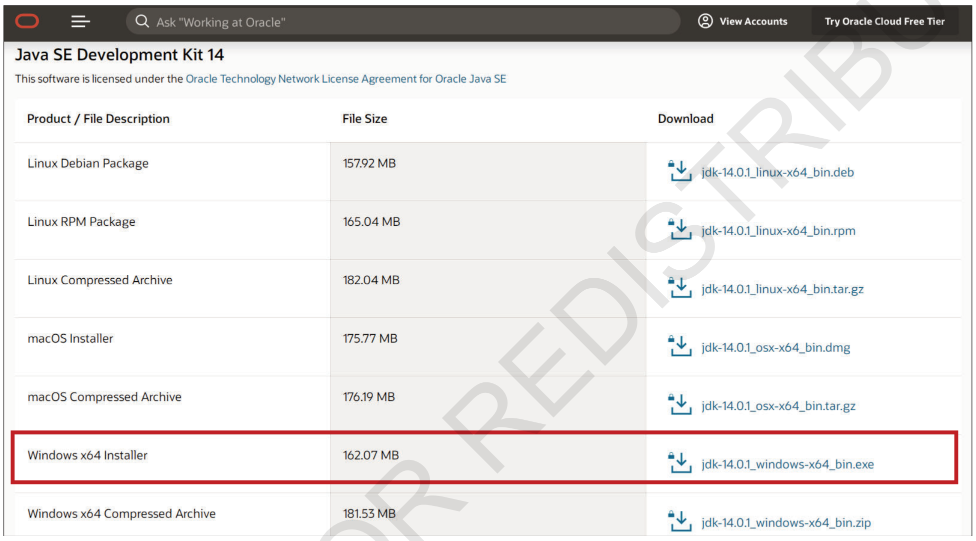
- Documentation
- Installation Instructions
- Release Notes
- Oracle License
  - Binary License
  - Documentation License
- Java SE Licensing Information User Manual
  - Includes Third Party Licenses
- Certified System Configurations

Oracle JDK

JDK Download

Documentation Download

Click “JDK Download”.

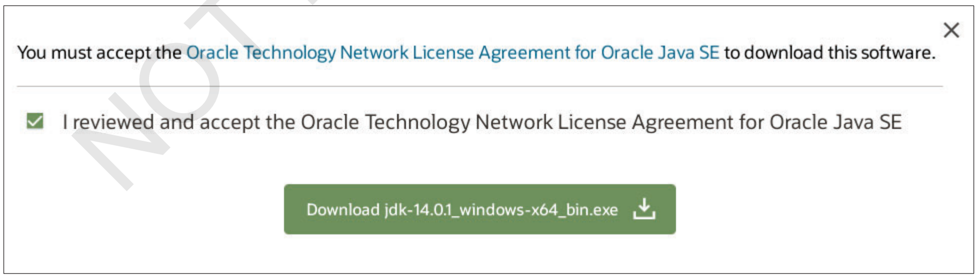


Java SE Development Kit 14

This software is licensed under the Oracle Technology Network License Agreement for Oracle Java SE

Product / File Description	File Size	Download
Linux Debian Package	157.92 MB	jdk-14.0.1_linux-x64_bin.deb
Linux RPM Package	165.04 MB	jdk-14.0.1_linux-x64_bin.rpm
Linux Compressed Archive	182.04 MB	jdk-14.0.1_linux-x64_bin.tar.gz
macOS Installer	175.77 MB	jdk-14.0.1_osx-x64_bin.dmg
macOS Compressed Archive	176.19 MB	jdk-14.0.1_osx-x64_bin.tar.gz
Windows x64 Installer	162.07 MB	jdk-14.0.1_windows-x64_bin.exe
Windows x64 Compressed Archive	181.53 MB	jdk-14.0.1_windows-x64_bin.zip

Select the appropriate executable file for your operating system. For Windows machine, click “jdk-14.0.1_windows-x64_bin.exe”.



You must accept the Oracle Technology Network License Agreement for Oracle Java SE to download this software.

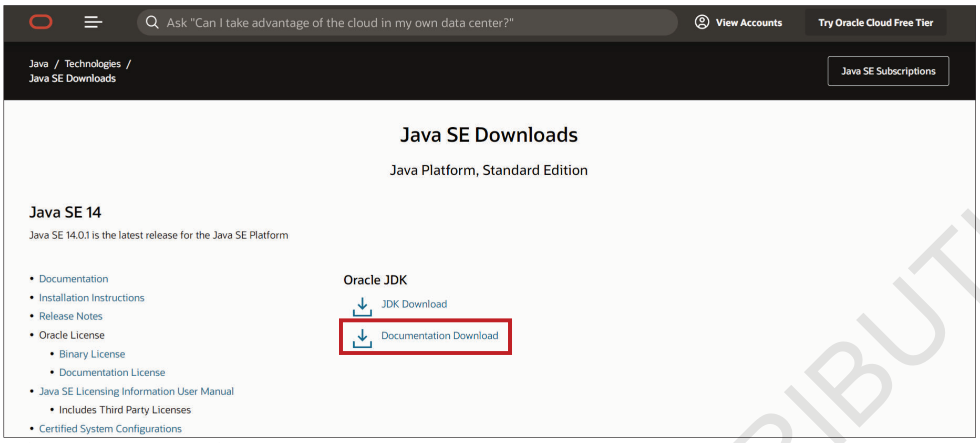
☑ I reviewed and accept the Oracle Technology Network License Agreement for Oracle Java SE

Accept the license agreement and click “Download jdk-14.0.1_windows-x64_bin.exe”.

When the download is complete, locate “jdk-14.0.1_windows-x64_bin.exe” in your computer. Double click “jdk-14.0.1_windows-x64_bin.exe” to begin the installation of Java Development Kit. Follow the instructions and select the default values for the installation by clicking “Next”. Finally, click “Close” to complete the installation.

1.3.2 Download Java SE 10 Documentation

Go to the Java download website:
http://www.oracle.com/technetwork/java/javase/downloads/index.html


Ask "Can I take advantage of the cloud in my own data center?"
View Accounts
Try Oracle Cloud Free Tier
Java / Technologies /
Java SE Downloads
Java SE Subscriptions

Java SE Downloads

Java Platform, Standard Edition

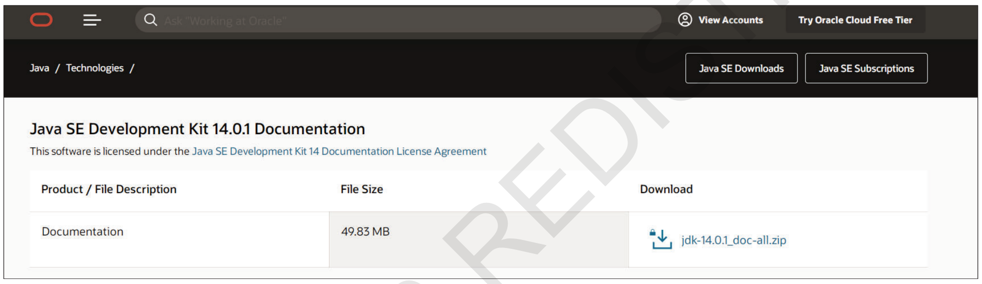
Java SE 14

Java SE 14.0.1 is the latest release for the Java SE Platform

- Documentation
- Installation Instructions
- Release Notes
- Oracle License
  - Binary License
  - Documentation License
- Java SE Licensing Information User Manual
  - Includes Third Party Licenses
- Certified System Configurations

Oracle JDK
JDK Download
Documentation Download

Click “Documentation Download”.

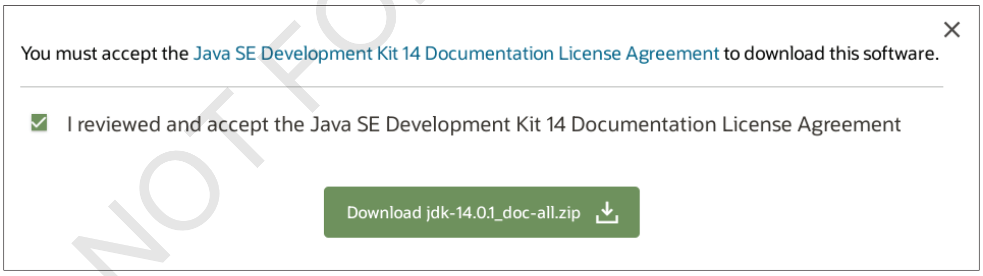

View Accounts
Try Oracle Cloud Free Tier
Java / Technologies /
Java SE Downloads
Java SE Subscriptions

Java SE Development Kit 14.0.1 Documentation

This software is licensed under the Java SE Development Kit 14 Documentation License Agreement

Product / File Description	File Size	Download
Documentation	49.83 MB	jdk-14.0.1_doc-all.zip

Click “jdk-14.0.1_doc-all.zip”.

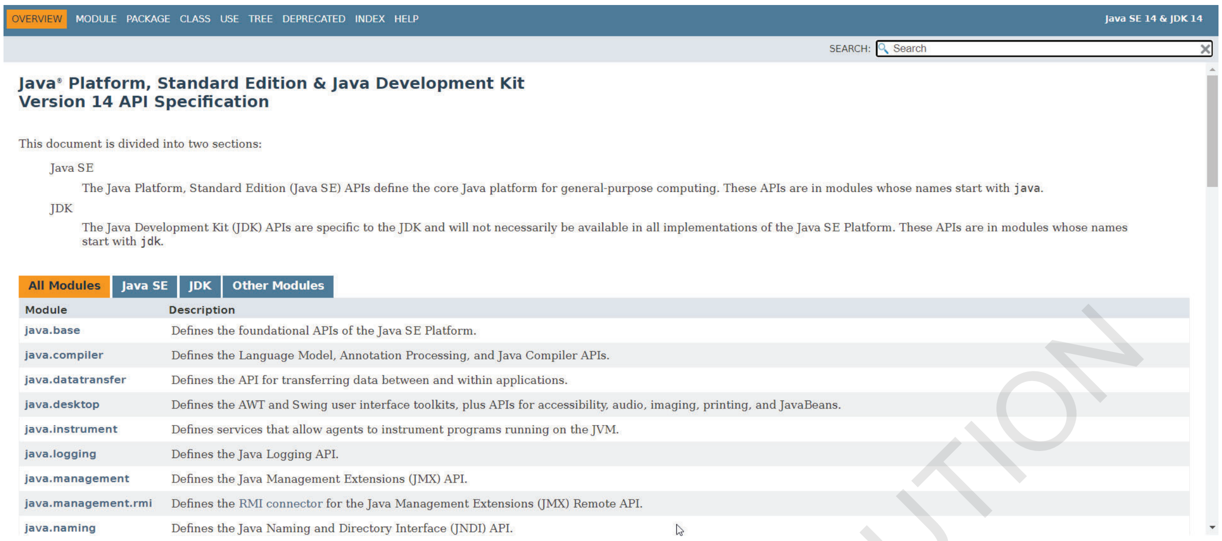

You must accept the Java SE Development Kit 14 Documentation License Agreement to download this software.

✓ I reviewed and accept the Java SE Development Kit 14 Documentation License Agreement

Download jdk-14.0.1_doc-all.zip

Accept the license agreement and click “Download jdk-14.0.1_doc-all.zip”.

When the download is complete, locate “jdk-14.0.1_doc-all.zip” in your computer. Use a Zip application (such as 7-zip) to unzip the file at where the zip file is. When the unzip is complete, a “docs” folder containing a number of other sub-folders is created. Click on “index.html” and the following is displayed.


OVERVIEW MODULE PACKAGE CLASS USE TREE DEPRECATED INDEX HELP
Java SE 14 & JDK 14
SEARCH: Search

Java® Platform, Standard Edition & Java Development Kit
Version 14 API Specification

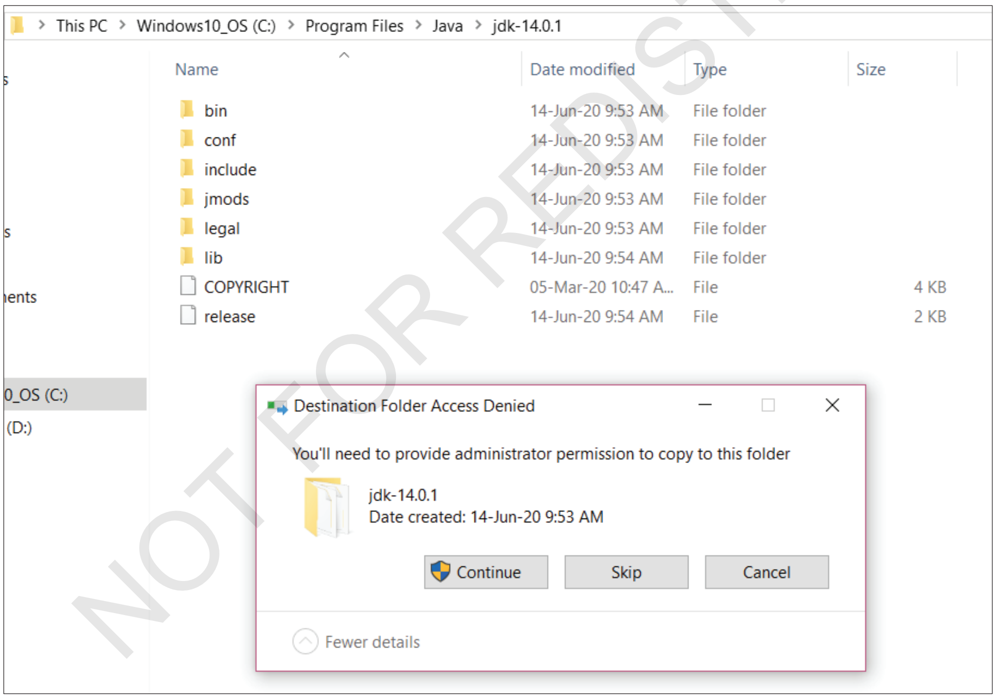
This document is divided into two sections:

Java SE
The Java Platform, Standard Edition (Java SE) APIs define the core Java platform for general-purpose computing. These APIs are in modules whose names start with java.

JDK
The Java Development Kit (JDK) APIs are specific to the JDK and will not necessarily be available in all implementations of the Java SE Platform. These APIs are in modules whose names start with jdk.

Module	Description
java.base	Defines the foundational APIs of the Java SE Platform.
java.compiler	Defines the Language Model, Annotation Processing, and Java Compiler APIs.
java.datatransfer	Defines the API for transferring data between and within applications.
java.desktop	Defines the AWT and Swing user interface toolkits, plus APIs for accessibility, audio, imaging, printing, and JavaBeans.
java.instrument	Defines services that allow agents to instrument programs running on the JVM.
java.logging	Defines the Java Logging API.
java.management	Defines the Java Management Extensions (JMX) API.
java.management.rmi	Defines the RMI connector for the Java Management Extensions (JMX) Remote API.
java.naming	Defines the Java Naming and Directory Interface (JNDI) API.

This is the documentation of the various classes found in the JDK. We will not be going through this in any details here. You may copy the "docs" folder to anywhere in your computer if you so desire. A good place to place this "docs" folder is in "C:\Program Files\Java\jdk-14.0.1". But you need to have administrator permission to copy this folder into this directory.


This PC > Windows10_OS (C:) > Program Files > Java > jdk-14.0.1
s
s
ents
0_OS (C:)
(D:)

Name	Date modified	Type	Size
bin	14-Jun-20 9:53 AM	File folder	
conf	14-Jun-20 9:53 AM	File folder	
include	14-Jun-20 9:53 AM	File folder	
jmods	14-Jun-20 9:53 AM	File folder	
legal	14-Jun-20 9:53 AM	File folder	
lib	14-Jun-20 9:54 AM	File folder	
COPYRIGHT	05-Mar-20 10:47 A...	File	4 KB
release	14-Jun-20 9:54 AM	File	2 KB

Destination Folder Access Denied

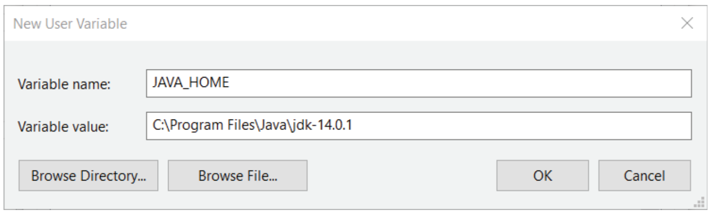
You'll need to provide administrator permission to copy to this folder

jdk-14.0.1
Date created: 14-Jun-20 9:53 AM

1.4 Setting the environment variables

Before we can start to write and execute our first Java program, there is one more thing to do – to set the environments variables. To do this, follow the steps below:

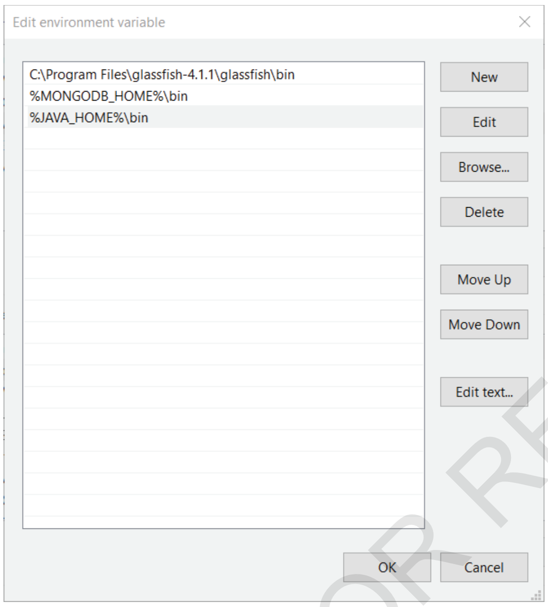
For Windows-based system: Start > Control Panel > System and Security > System > Advanced system settings > Environment Variables… > User variables. Click New… below the User variables tab. Enter “JAVA_HOME” for Variabe Name and “C:\Program Files\Java\jdk-14.0.1” for Variable Value as shown here, then click OK:


New User Variable

Variable name: JAVA_HOME

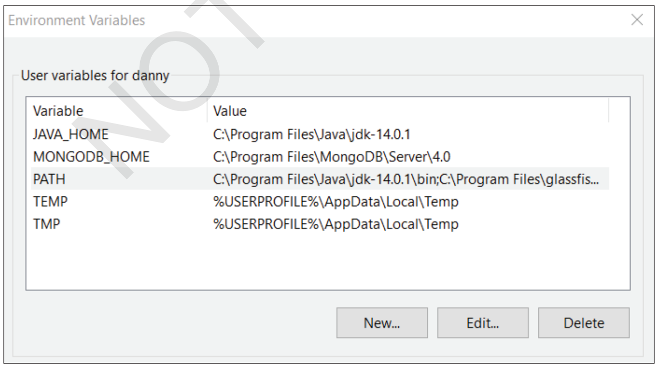
Variable value: C:\Program Files\Java\jdk-14.0.1

Next, we need to set the PATH variable. Select PATH in the User Variable tab and click “Edit…”. Click “New” then enter “%JAVA_HOME%\bin”. Click “OK”.


Edit environment variable

```
C:\Program Files\glassfish-4.1.1\glassfish\bin
%MONGODB_HOME%\bin
%JAVA_HOME%\bin
```

You should have the following environment variables set for “JAVA_HOME” and “PATH” for the User variables:


Environment Variables

User variables for danny

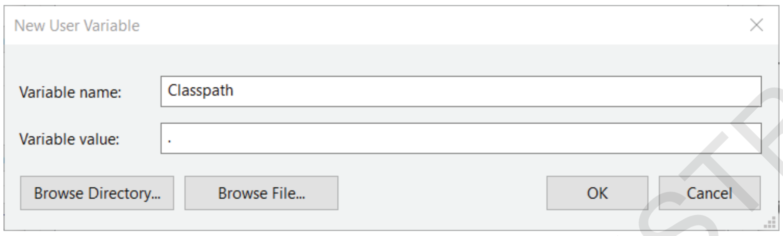
Variable	Value
JAVA_HOME	C:\Program Files\Java\jdk-14.0.1
MONGODB_HOME	C:\Program Files\MongoDB\Server\4.0
PATH	C:\Program Files\Java\jdk-14.0.1\bin;C:\Program Files\glassfis...
TEMP	%USERPROFILE%\AppData\Local\Temp
TMP	%USERPROFILE%\AppData\Local\Temp

1.5 Setting the classpath variable

The classpath variable is an environment variable that tells the JVM where to locate the appropriate classes to execute. For most beginners to Java programming, the classpath variable has been the stumbling block. Error messages that report "Class Not Found" are usually due to incorrect classpath variable setting. Using the default settings in the installation package would enable the correct setting of the classpath variable and therefore the proper execution of any Java program.

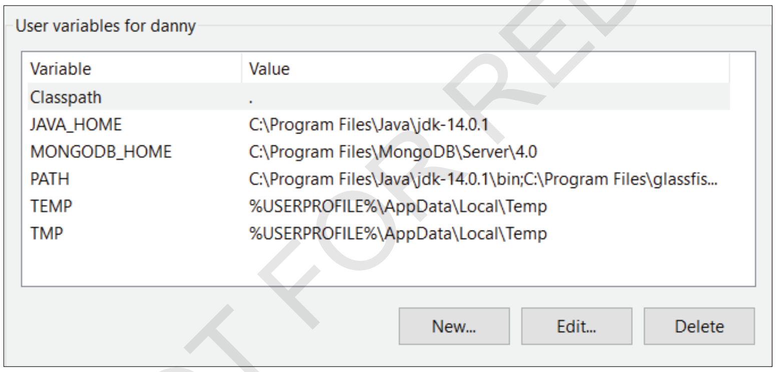
It is essential that the current directory be included in the classpath to ensure the correct compilation and execution of Java classes. To do this, follow the steps below:

For Windows-based system: Start > Control Panel > System and Security > System > Advanced system settings > Environment Variables… > User Variables > Classpath > Variable Value. Check the Variable Value of Classpath. If the current directory denoted by "." (i.e. period) is not included in the Classpath Variable Value, add it in. If there is no Classpath variable, add in a Classpath variable by clicking New… below the User Variables tab. Enter "Classpath" for Varibale Name and "." for Variable Value as shown here, then click OK:



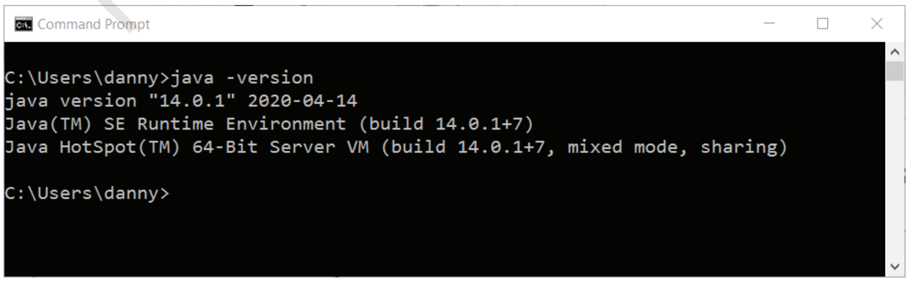
Variable name: Classpath
Variable value: .

You should get the following window for User Variables, click OK and your classpath is set:



Variable	Value
Classpath	.
JAVA_HOME	C:\Program Files\Java\jdk-14.0.1
MONGODB_HOME	C:\Program Files\MongoDB\Server\4.0
PATH	C:\Program Files\Java\jdk-14.0.1\bin;C:\Program Files\glassfis...
TEMP	%USERPROFILE%\AppData\Local\Temp
TMP	%USERPROFILE%\AppData\Local\Temp

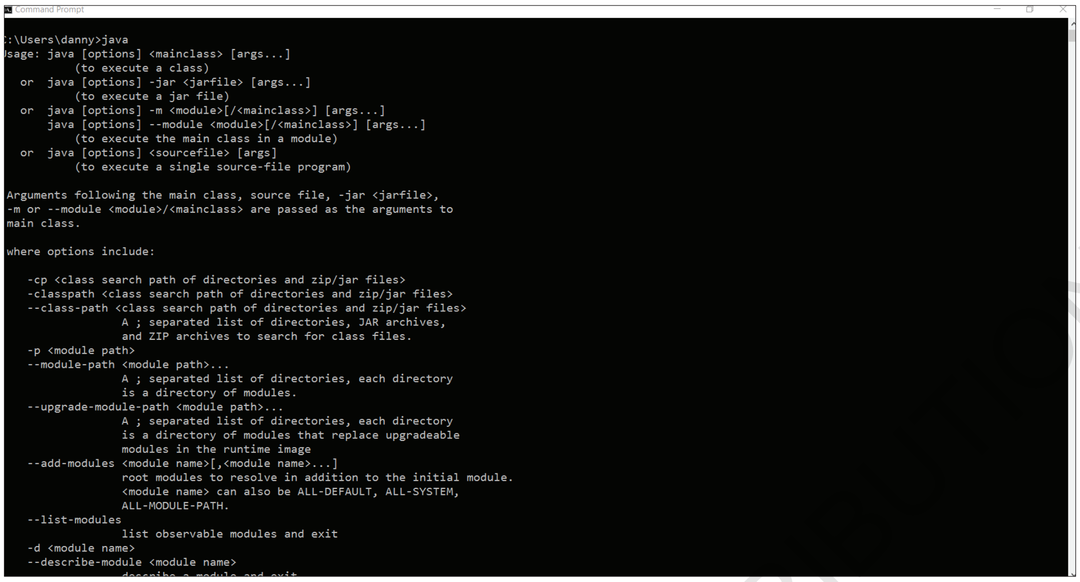
Open a Command Prompt and type in "java-version". You should get the version of JDK you have just installed.



```
C:\Users\danny>java -version
java version "14.0.1" 2020-04-14
Java(TM) SE Runtime Environment (build 14.0.1+7)
Java HotSpot(TM) 64-Bit Server VM (build 14.0.1+7, mixed mode, sharing)

C:\Users\danny>
```

Typing “java” next, you should have the following:



```
Command Prompt

:\Users\danny>java
Usage: java [options] <mainclass> [args...]
           (to execute a class)
   or  java [options] -jar <jarfile> [args...]
           (to execute a jar file)
   or  java [options] -m <module>[/<mainclass>] [args...]
       java [options] --module <module>[/<mainclass>] [args...]
           (to execute the main class in a module)
   or  java [options] <sourcefile> [args]
           (to execute a single source-file program)

Arguments following the main class, source file, -jar <jarfile>,
-m or --module <module>/<mainclass> are passed as the arguments to
main class.

where options include:

   -cp <class search path of directories and zip/jar files>
   -classpath <class search path of directories and zip/jar files>
   --class-path <class search path of directories and zip/jar files>
                 A ; separated list of directories, JAR archives,
                 and ZIP archives to search for class files.
   -p <module path>
   --module-path <module path>...
                 A ; separated list of directories, each directory
                 is a directory of modules.
   --upgrade-module-path <module path>...
                 A ; separated list of directories, each directory
                 is a directory of modules that replace upgradeable
                 modules in the runtime image
   --add-modules <module name>[,<module name>...]
                 root modules to resolve in addition to the initial module.
                 <module name> can also be ALL-DEFAULT, ALL-SYSTEM,
                 ALL-MODULE-PATH.
   --list-modules
                 list observable modules and exit
   -d <module name>
   --describe-module <module name>
                 describe a module and exit
```

If you get the above displayed, you are ready to go to compile and run your first Java program.

1.6 Examining the Java Folder

Let us examine the Java folder to have a better understanding of the programs and packages available for executing Java applications. If you had used the default directory for installing the Java packages, you should have the following file directories within the “Program Files\Java” folder in the C drive:

- jdk-<version>
- jre-<version>

Within the jdk-<version> folder are two folders of interest to us:

1. **bin**: This folder contains executable files. Of particular interest to us are two “.exe” files (javac.exe and java.exe). javac.exe is used to compile a Java program into byte code (with “.class” file) and java.exe executes a Java program in a “.class” file.
2. **lib**: This folder contains a library of classes that may be required for the execution of your programs. Classes are usually packaged into “.jar” (Java Archive) files. Sometimes, you may have to include additional classes to support your programs. Add them into the “lib\ext” folder.

The jre-<version> folder contains files or packages essential for the proper execution of Java programs.

1.7 Running a Java Program

We will use the simple Java program of Code 1.1 to illustrate the process of executing a Java program.

Produce the program using the usual word editor such as Wordpad or Notepad that comes with Windows. Do not use word editor with formatting facilities such as Microsoft Word, as Java only recognizes text and will treat non-texts (used for formatting purpose) as errors.

We will save the above program into a text-file; let us call it "HowDoYouDo.java". All Java program files have the file extension ".java". To compile the program, go to the directory where you have saved the program and type at the command prompt:

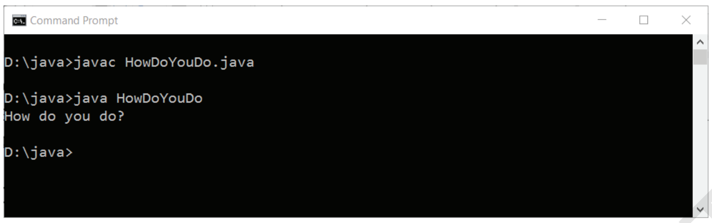
```
$javac HowDoYouDo.java
```

"$" is indicative of the command prompt and javac is the Java compiler. When javac has successfully compiled the Java program, a ".class" file is produced. This file has the name "HowDoYouDo.class" since the class is named HowDoYouDo. To execute the HowDoYouDo program, type:

```
$java HowDoYouDo
```

The command java calls the JRE (Java Runtime Environment) or JVM to execute the HowDoYouDo program. We do not include the ".class" extension in the command line. Executing this class produces the line:

```
How do you do?
```

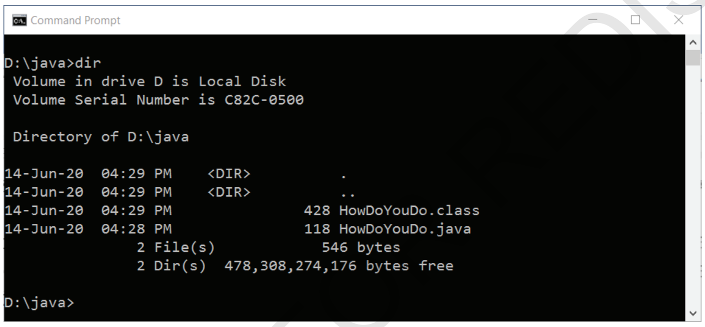


```
D:\java>javac HowDoYouDo.java

D:\java>java HowDoYouDo
How do you do?

D:\java>
```

Type in "dir" at the command prompt and a list of files in the directory will be displayed:



```
D:\java>dir
 Volume in drive D is Local Disk
 Volume Serial Number is C82C-0500

 Directory of D:\java

14-Jun-20  04:29 PM    <DIR>          .
14-Jun-20  04:29 PM    <DIR>          ..
14-Jun-20  04:29 PM               428 HowDoYouDo.class
14-Jun-20  04:28 PM               118 HowDoYouDo.java
               2 File(s)            546 bytes
               2 Dir(s)  478,308,274,176 bytes free

D:\java>
```

Note the following:

1. The name of the Java file ("HowDoYouDo.java") that contains the Java program HowDoYouDo needs not be the same as the program or class name. We could have named the file as "OurFirstProgram.java" and the program will still compile and produce the class "HowDoYouDo". However, there are exceptions to this rule.
2. The generated byte code from the compilation process is stored in a file with a ".class" extension. The name of the ".class" file is the same as the class name. You may include multiple classes in a ".java" file. There will be as many ".class" files generated as there are classes in the ".java" file.

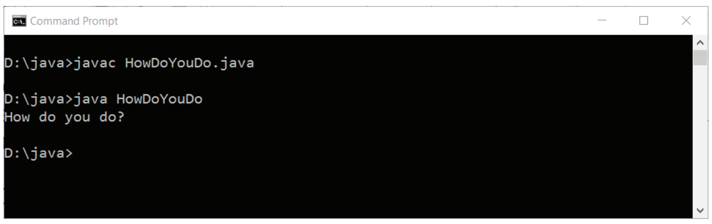
You can only execute the class (or program) that contains the main() method.

1.8 Understanding the Compilation and Execution Process

To help you in understanding the compilation and execution process, a step-by-step guide to running the HowDoYouDo class is given below. Follow the steps in Table 1.1 to run a Java program.

TABLE 1.1: Executing HowDoYouDo Class

Step	Task	How
1.	Open a command prompt	Start > run > enter "cmd"
2.	Select a drive you would like to work on e.g. Drive D	cd d:\
3.	Create a directory "java"	mkdir java
4.	Go to directory "java"	cd java
5.	Use a word editor to create the program (Code 1.1) and save it in "HowDoYouDo.java"	
6.	Compile program with javac HowDoYouDo.java	javac HowDoYouDo.java
7.	If the compilation is successful, check the directory and note that a HowDoYouDo.class file is produced	dir
8.	Execute the Java class with java HowDoYouDo	java HowDoYouDo



```
D:\java>javac HowDoYouDo.java

D:\java>java HowDoYouDo
How do you do?

D:\java>
```

1.9 Walking through the Code

Let us walk-through the steps involved in executing a Java program. The HowDoYouDo.class file contains the byte code generated by the compilation process via javac. Byte code needs to be translated (by the JVM) into object code. What the JVM does is to load the class into computer memory and executes the code starting from the first statement in the main() method. If there is no main() method in the class, an error message will be reported.

Parameters are passed into the program (or class) via the String array of arguments (String[] argv or String[] args). Since HowDoYouDo program does not require any parameter, the array is empty.

The only statement in this class is subsequently executed. As mentioned earlier, System.out.println is Java's way of producing outputs. The output is written to the output stream which by default is directed to the screen. This explains why the output "How do you do?" is produced on the screen.

1.10 Commonly Encountered Problems

Programmers new to Java programming need to be aware of the following characteristics of Java to avoid errors in coding:

1. **Java is case-sensitive.** A class named HowDoYouDo.java is different from one named howdoyoudo.java. A variable named Total is different from total. Also, the primitive type int when spelt as Int is not recognized as such.
2. **javac acts on a file of ".java" extension and not on a ".class" extension.** The ".java" extension must be included when javac is called, for example:
   $javac HowDoYouDo.java
3. **java acts on a file of ".class" extension.** Execution of a ".class" file does not require the ".class" extension, for example:
   $java HowDoYouDo
4. **Only classes with the main() method are executable.**

CHAPTER 2: BASIC BUILDING BLOCKS

Code 2.1 is a Java program that prints two lines; each line displays a number that has been defined within the program.

Code 2.1: The Print2Numbers Class
```
class Print2Numbers {

  public static void main (String args[]) {
    // Comment: prints 2 numbers
    int firstNumber   = 5;
    int second_Number = 10;

    System.out.println("The 1st number is " + firstNumber);
    System.out.println("The 2nd number is " + second_Number);
    return;
  }
}
```

2.1 Print2Numbers Java Program

You should be familiar with compiling and running a Java program by now. Do not proceed until you are sure of this step.

When Code 2.1 is executed, two lines are printed:

```
The 1st number is 5
The 2nd number is 10
```

The output once again is produced by the two "System.out.println()" lines. It is clear that System.out.println() displays outputs on the command prompt. It is an output producing command. In fact, whatever is enclosed within the two double quotes ("") are printed. The "+" sign is an operator that joins the item on the right to the item on the left of the "+" sign. That is how we get:

```
The 1st number is 5
```

when the two "System.out.println()" lines are executed.

Code 2.1 consists of a number of components: words, numbers, punctuation marks and special symbols. By combining these items together, we can form Java statements. In the English language, letters from the English alphabet are combined to form words and the words are put together to form sentences. In English, there are grammar rules governing the way sentences are formed. In Java, sentences are known as statements. The grammar rules governing the formation of statements are known as the syntax rules of Java.

Unlike spoken languages, the list of words that can be used to form statements in Java is limited and small in number. That is perhaps the beauty of the Java programming language. Smallness suggests simplicity but that does not mean Java is limited in its usage. Java is powered up with the availability of the Application Programming Interfaces (APIs), a class library.

2.2 The Java Vocabulary and Character Sets

The vocabulary of a language refers to the list of words used in the language. Java has its own set of letters, digits and special symbols (such as +, - , *, /, etc.).
In the early days of programming when English is predominantly the language used for expressing programming concepts, the 7-bit ASCII character set was popularly used. This character set can represent up to 128 characters (2^7); this is more than sufficient to represent the English alphabet (lower case and upper case) and some special symbols.

With the advent of the Internet, it has become imperative for systems to be designed to cater for language representation beyond the English language. The 7-bit ASCII character set has become inadequate and the Unicode, a 16-bit character set, was subsequently introduced. With 16 bits, Unicode can represent up to 65536 (2^{16}) characters.

Unicode has been used to represent Western European languages as well as Asian languages such as Chinese. Nevertheless, the 7-bit ASCII character set has been subsumed within the Unicode character set. In fact, the first 128 characters of the Unicode set are equivalent to the 7-bit ASCII character set and the first 256 characters of the Unicode form the Latin-1 character set.

2.3 Primitive Data Types

Primitive data types are data types that have been pre-defined in Java. Table 2.1 lists the primitive data types used in Java. Each data type has its own constant values specified using literals. (Note: A literal is a letter or symbol that stands for itself.)

The primitive data type int has been used to define two numbers in Code 2.1:

```
int firstNumber   = 5;
int second_Number = 10;
```

TABLE 2.1: Primitive Data Types

Type	Remarks
boolean	Either true or false
char	16-bit Unicode 1.1.5 character
byte	8-bit signed two's-complement integer
short	16-bit signed two's-complement integer
int	32-bit signed two's-complement integer
long	64-bit signed two's-complement integer
float	32-bit IEEE 754-1985 floating-point number
double	64-bit IEEE 754-1985 floating-point number

2.3.1 Boolean

In the 19th century, George Boole (1815-1864), a British mathematician invented the Boolean algebra/logic – a system of symbolic logic with values of "true" and "false". This system was later extended into a combinatorial system of propositions with logical operators "AND", "OR", "IF THEN", "EXCEPT" and "NOT". In Java, we can define Boolean variables which take on Boolean literals of "true" and "false".

2.3.2 Characters

Character literals are denoted using single quotes ‘’ e.g. ‘a’. They can be combined with a backslash (“\”) to define a specific function in Java. Such characters are described as escaped characters. Table 2.2 highlights some escaped characters and their pre-defined functions.

TABLE 2.2: Escaped Characters

Escaped Character	Function	Unicode
\n	newline	(\u000A)
\t	tab	(\u0009)
\b	backspace	(\u0008)
\r	return	(\u000D)
\f	form feed	(\u000C)
\\	backslash	(\u005C)
\’	single quote	(\u0027)
\”	double quote	(\u0022)
\ddd	octal value char with each d of value 0-7	
\dddd	hexadecimal value char with each d of value 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F	

2.3.3 Integers

Integer constants can be represented as octal, decimal or hexadecimal values. The base of a constant is denoted by the first character of the constant. Table 2.3 illustrates this.

TABLE 2.3: Integer Constants

Constant starts with	Value Type
0 (a single zero)	Octal (base 8)
0x	Hexadecimal (base 16)
0X	Hexadecimal (base 16)
Other than 0, 0x, 0X	Decimal (base 10)

For example, Table 2.4 shows some examples of integer constants that have the same value.

TABLE 2.4: Examples of Integer Constants

Constant	Value Type
036	Octal (base 8)
0x1E	Hexadecimal (base 16)
0X1E	Hexadecimal (base 16)
30	Decimal (base 10)

Integer constants can be of type long, int, short, or byte. int and long represents the set of whole numbers, and any value of these types is therefore a whole number. The major difference between int and long lies in the size of numbers that each can represent. Obviously, more numbers can be represented using long (64-bit signed two’s-complement integer) than int(32-bit signed two’s-complement integer).

Integer constants are long if the constants end with L or l, otherwise, they are assumed to be int. If the constant of type int is assigned to short or byte and if the value is within the range for that type, then the int literal assumes short or byte respectively.

2.3.4 Floating Point

Floating-point numbers are decimal numbers with an optional decimal point followed by an optional exponent. For example, the followings are floating-point numbers:

```
8.     90.0   6.7e3  0.18E4
```

The size of such numbers a computer can represent is limited by the implementation architecture of the machine. By default, all floating-point constants are declared double unless there is a trailing f or F at the end of the literals; in which case, they are considered as float. A trailing d or D specifies a double constant. The set of values of the type float is a subset of the set of values of type double.

A float constant can only be assigned to a float variable. To assign a double to a float, you will need to cast it to a float. A double constant cannot be assigned directly to a float variable, even if the value of the double is within the valid float range.

2.3.5 Object References

The only literal constant for object references is null. It represents an invalid or uncreated object. We will discuss the concept of objects in Chapter 6.

2.3.6 String

String literals are defined within double quotes “” e.g. “How do you do?” is a string. All escape sequences that are applicable to characters can be embedded within String literals. For example,

```
“The colour of the bag is\t= blue\n”
```

This String literal has a tab (\t) and a newline (\n) included within it.

2.4 Identifiers

Declared entities (such as variables and constants) and labels are named by identifiers. Java identifiers must begin with a letter, underscore (_) or dollar sign ($), followed by any number of letters, digits, underscore (_), and dollar sign ($). For example, the following are valid identifiers:

```
HelloWorld
$Dollar
_S$90
S$88
abc789
```

but the followings are not:

```
8Stars
Tom&Jerry
```

Java is case-sensitive and this applies to identifiers too. Hence, “a” and “A” in identifiers are different and they render different identifiers. Any differences within an identifier make the identifier unique. For example,

```
firstNumber
FirstNumber
first_Number
First_Number
```

are different identifiers. Two identifiers of int type have been declared in Code 2.1:

```
firstNumber
second_Number
```

2.5 Reserved Words

Java has a list of 46 reserved words that you cannot use them to define identifiers. Table 2.5 shows the list of reserved words.

TABLE 2.5: Reserved Words

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

goto and transient are currently not used. null, true, and false are not reserved words; they are formally known as literals and should not be used as identifiers too. Although reserved words cannot be used as identifiers they can be used as part of identifiers e.g. "value**return**ed" or "**new**Variable" are valid identifiers.

Since Java is case-sensitive, Abstract, New, and instanceOf can be considered as identifiers since abstract is not equivalent to Abstract; neither are New the same as new, and instanceOf is also not the same as instanceof. However, such use of identifier names can be confusing and may obscure the intention of the code. You are strongly advised against such practices.

2.6 Comments

It is often good practice to write some notes describing the purpose and algorithm of a piece of code. Such notes are useful to others who might have to take over the maintenance of the program. They are also useful to the original programmer who may have forgotten the reason for adopting a certain strategy when he/she first wrote the code. Most, if not all, programming languages provide a means for programmers to write such notes, commonly known as comments. The latter are ignored by the JVM at run-time.

Java provides three types of comments and they are denoted in Table 2.6.

TABLE 2.6: Comments

No.	Comment Type	Remarks
1	// comment	Any character after // to the end of the line is treated as comment and is ignored.
2	/* comment */	Any character, including line terminators \r, \n, or \r\n, between /* and the next */ is ignored. Useful in situations where there are multiple lines of comments.
3	/** comment */	Documentation comments. Similar to Comment Type 2 but is used immediately before a class declaration, class member, or constructor. Such comments are included in automatically generated documentation. We will discuss class declaration, class member, and constructor in Chapter 7.

Java comments cannot be nested. The following code does not compile:

```
class Comments {
  /*
    public static void main (String args[]) {
     /* Comment: prints 2 numbers (a nested comment) */
        int firstNumber   = 5;
        int second_Number = 10;
        System.out.println("The 1st number is " + firstNumber);
        System.out.println("The 2nd number is " + second_Number);
    }
  */
}
```

2.7 Basic Program Structure

A **program** in Java is defined by a class. A class may contain data and methods. A method is a function that executes some statements and returns a value that may be empty (or void). An example of a method is main(String args[]). This method returns void.

Code 2.1 is a simple but complete Java program. Let us look through Code 2.1 line by line from the top, reproduced here for clarity.

```
 1. class Print2Numbers {

 2.   public static void main (String args[]) {
 3.     // Comment: prints 2 numbers
 4.     int firstNumber   = 5;
 5.     int second_Number = 10;
 6.
 7.     System.out.println("The 1st number is " + firstNumber);
 8.     System.out.println("The 2nd number is " + second_Number);
 9.     return;
10.   }
11. }
```

TABLE 2.7: Code 2.1 Explained

Line Number	Explanation
1	Defines the class named Print2Numbers. By convention, the first letter of all class names is in upper-case. The file that contains this program is named as Print2Numbers.java. When this program is compiled, a Print2Numbers.class file is generated. This file contains the executable byte code of Print2Numbers class.
2	Defines the main() method. The word public suggests that this method can be called by another method outside this class. We will discuss in more detail what the word static means in Chapter 6 but for now, just take it that static defines the main() method as a *class method*[1]. void suggests that the main() method does not return any value. args[] is an array of String objects containing the values entered by users at the command line prompt (argv[] can be used as an alternative to args[]). For this example, there is no input for the String array; any attempt to print the values of the String array will produce an exception (or error).
4	Declares an integer firstNumber as a data and initializes it to the value 5.
5	Declares an integer second_Number as a data and initializes it to the value 10.
7	System.out.println is a method in the Java class library that outputs whatever is specified within the parentheses. The symbol + within the parentheses is a concatenation operator that joins two operands i.e. "The 1st number is " and the value of firstNumber. The output is thus "The 1st number is 5".
8	Outputs the value "The 2nd number is 10".
9	Causes the method to return to where it was called. Since the method was called from the command line prompt, the program terminates and exits to the command line prompt. The return; statement is actually not necessary since all programs terminate and exit after the last line is executed.

[1] Class Method as opposed to Instance Method. A class method belongs to a class while an instance method belongs to an instance or object.

CHAPTER 3: STATEMENTS AND OPERATORS

We form sentences in English to express ourselves. In Java, we form statements and statements are formed from expressions. An expression is a rule for computing value and a statement is an expression terminated with a semi-colon.

Structurally, an expression is made up of one or more operands inter-related by operators. An operand is one of these three values:

1. a constant value
2. the value of a variable
3. the result of a method call

An *operator* acts on an operand or operands to produce a value. Bringing all these together, an example of an expression looks like this:

```
sum = number + 5
```

sum, number and 5 are operands and + and = are operators.

When we add a semi-colon to the end of an expression, we get a statement. For example,

```
sum = number + 5;
```

is a statement. A statement is executable and a result is produced when it is evaluated.

A statement is executed by evaluating the right-hand side of the assignment operator (denoted by "="). The result of the evaluation is assigned to the variable on the left-hand side of the assignment operator. Thus, sum will have the result of the evaluation of number + 5.

There are two types of statement:

1. Expression Statements
2. Declaration Statements

3.1 Expression Statements

Expression statements are expressions that are terminated by a semi-colon. The previous example

```
sum = number + 5;
```

is an example of an expression statement. Another example of an expression statement is:

```
int sum = 10 + number;
```

The expression within the expression statement is:

```
int sum = 10 + number
```

More specifically, we say that the above expression statement is an *Assignment Expression Statement*.

Not all expressions can be made into a statement merely by adding a semi-colon to it. For example, adding a semi-colon to the following expression does not make it into a statement:

```
sum > number
```

In fact, it becomes an invalid statement at compilation time.

3.1.1 Types of Expressions

There are four types of expressions in Java that can be made into expression statements simply by adding a semi-colon at the end of the expression:

1. Assignment Expressions.
2. Prefix or postfix forms of "++" and "--".
3. Method calls.
4. Object creation expressions.

3.1.2 Assignment Expression Statements

Assignment expression statements are statements denoted by "=" or "op=" operators. "op" may be any of the following operators: +, -, *, /, %, >>, <<, &, ^ or |. The following four examples are assignment expression statements:

```
stringA = "The quick brown fox jumps ";
stringB = stringA + "over the lazy dog";
int sum = 0;
sum     = sum + 10;
```

The "=" symbol is an *assignment operator*. Its function is to evaluate the right-hand side of a statement and put the result of the evaluation into the left-hand side variable of the statement. stringA and stringB are of type String in the first two statements above. stringA is assigned a string "The quick brown fox jumps ". This string is concatenated with string "over the lazy dog"in the second statement resulting in a longer string "The quick brown fox jumps over the lazy dog" which is assigned to stringB. The "+" operator in the second statement acts like a *string concatenating operator*.

The "+" operator in the fourth statement takes two integer operands (sum and 10) and behaves as an integer add operator rather than a string concatenating operator. The behaviour change of an operator is dictated by the type of the operands. Such change in behaviour is permissible in Java and is known as *operator overloading*.

3.1.3 Prefix or Postfix Forms of "++" and "--" Statements

Prefix or postfix forms of "++" and "--" expressions can be transformed into expression statements by adding a semi-colon to the end of the expressions. Examples of such statements include:

- Prefix: e.g. ++sum;
- Postfix: e.g. sum++;
- Prefix: e.g. --total;
- Postfix: e.g. total--;

3.1.4 Method Call Statements

Method calls that return or do not return values can be transformed into expression statements. Examples of such statements include:

```
myCounter.add();
parser.parse();
```

3.1.5 Object Creation Statements

Objects can be created using the new operator (see Chapter 6). Expressions for creating objects can be transformed into expression statements by adding a semi-colon to the expressions. Examples include:

```
new Counter();
new Analyser();
```

3.2 Declaration Statements

Declaration statements are used to declare variables and initialize variables. Unlike other programming languages, declaration statements can exist anywhere inside a block. A *block* is a group of statements enclosed within braces ({ and }). Examples of declaration statements include:

```
int number  = 1;
String name = “John”;
```

The first statement declares number to be a variable of primitive data type int and initializes it to the value 1. The second statement declares name to be a variable of type String and initializes it to the value “John”.

3.3 Operators

An *operator*, as its name suggests, operates on one or more values to produce a result. The values operated on are known as *operands.*

A *unary* operator is one that takes in a single operand while a *binary* operator requires two operands to produce a result.

Java supports the following types of operators:

1. Arithmetic operators
2. Auto-Increment and Auto-Decrement operators
3. Logical operators
4. Relational operators
5. Bitwise operators
6. The Conditional operator “?:”
7. Assignment operators
8. “+”operator
9. “.” operator

3.3.1 Arithmetic Operators

Arithmetic operators act on numeric operands. Java supports unary and binary arithmetic operators. The unary arithmetic operator “-” is used to denote negation, for example, -5. The sign of a number can also be inverted like this:

```
number = -number;
```

Binary arithmetic operators include:

+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Remainder

The easiest way to appreciate arithmetic operators is through the use of examples:

```
int revenue   = 0;
int cost      = 0;
int profit    = 0;
int price     = 45;
int quantity  = 10;
int unitCost  = 30;
int tax       = 5;
int remainder = 0;

cost      = unitCost + tax   // returns cost with value 35
profit    = price - cost     // returns profit with value 10
revenue   = quantity * price // returns revenue with value 450
monthly   = revenue/12       // returns monthly with value 37
remainder = revenue%12       // returns remainder with value 6
```

These operators apply to numeric operands and return a value consistent with the type of its operands. However, when there is a mix of operand types, the widest is used to prevent truncation. For example,

```
float floatNumber = 17;
int   intNumber   = 8;

floatNumber + intNumber  // returns result 25.0
floatNumber -  intNumber // returns result 9.0
floatNumber * intNumber  // returns result 136.0
floatNumber / intNumber  // returns result 2.125
```

Dividing by zero is invalid for integer arithmetic and this throws an exception (ArithmeticException). An exception is an indication of the occurrence of an error. The use of exception provides a gracious way to exit from an execution when an error is encountered.

3.3.2 Auto-Increment and Auto-Decrement Operators

Auto-increment and auto-decrement operators are denoted by "++" and "--" respectively; they are applied on integer operands. For example,

```
a++;   // this is equivalent to a = a + 1;
b--;   // this is equivalent to b = b - 1;
```

These operators can be either *prefix* or *postfix* operators. When the operator appears before the variable, it is said to be a *prefix operator* (e.g. ++a). When the operator appears after the variable, the operator is said to be a *postfix operator* (e.g. a++).

For prefix operator, the operation is carried out *before* the value of the expression is returned. For postfix operator, the operation is applied *after* the original value is used. We will illustrate the effect of prefix and postfix auto-increment and auto-decrement operators using 4 similar examples in Table 3.1.

TABLE 3.1: Prefix and Postfix Auto-Increment and Auto-Decrement Operators

	Auto-increment	Auto-decrement

Prefix	`int a = 0;`
	`int b = 0;`
	`b = 7;`
	`a = ++b; // a is 8, b is 8`
		`int a = 0;`
		`int b = 0;`
		`b = 7;`
		`a = --b; // a is 6, b is 6`

Postfix	`int a = 0;`
	`int b = 0;`
	`b = 7;`
	`a = b++; // a is 7, b is 8`
		`int a = 0;`
		`int b = 0;`
		`b = 7;`
		`a = b--; // a is 7, b is 6`

3.3.3 Logical Operators

Java supports the standard set of logical operators that return Boolean values of true or false (see Table 3.2).

TABLE 3.2: Logical Operators

Logical Operators	Represented in Java by	Number of Operands	Returns true if
and	&&	Binary	Both operands are true
or	||	Binary	At least one operand is true
not	!	Unary	Single operand is false

The truth table for each of the logical operators is given below:

TABLE 3.3: Truth Table for &&

a	b	a && b
false	false	false
false	true	false
true	false	false
true	true	true

TABLE 3.4: Truth Table for | |

a	b	a || b
false	false	false
false	true	true
true	false	true
true	true	true

TABLE 3.5: Truth Table for !

a	!a
false	true
true	false

Boolean values are tested directly. Hence, we write:

```
if (x && y) {
  // do something
}
```

rather than

```
if (x == true && y == true) {
  // do something
}
```

If x is false, Java does not evaluate y and the body of the if statement is not executed. Java does this for efficiency reason. This rule also ensures program correctness. Consider the following code:

```
if ( index >= 0 &&
     index < array.length &&
     array[index] != 0) {
  // do something
}
```

The range check on the index is done first. If the index is negative or not within the bounds of the array, the test is aborted and the body of the code is not executed. The above code ensures safe execution.

3.3.4 Relational Operators

Java supports the standard set of relational and equality operators:

TABLE 3.6: Relational and Equality Operators

Relational Operator	Meaning	Operate on
>	Greater than	Numeric values
>=	Greater than or equal to	Numeric values
<	Less than	Numeric values
<=	Less than or equal to	Numeric values
==	Equal to	All values
!=	Not equal to	All values

The following example illustrates the condition of each test:

```
int a = 5;
int b = 5;
int c = 6;

a < b         // returns false
a <= b        // returns true
a < c         // returns true
a >= c        // returns false
a == b        // returns true
```

Only the equality (==) and inequality (!=) operators are allowed to operate on Boolean values. They operate on all values including objects to return a Boolean result. Let's consider an example:

Code 3.1: The EqualityOp Class

```
class EqualityOp {

  public static void main(String argv[]) {
    int a = 5;
    int b = 5;
    int c = 6;
    String s1 = new String("John");
    String s2 = new String("John");

    if (a == b) System.out.println("a == b is true");
    if (a == c) System.out.println("a == c is true");
    if (b != c) System.out.println("b != c is true");
    if (s1 == s2) System.out.println("s1 == s2 is true");
    if (s1 == s1) System.out.println("s1 == s1 is true");
    if (s1.equals(s2)) System.out.println("s1, s2 contents are the same");
  }
}
```

The output of Code 3.1 is:

```
a == b is true
b != c is true
s1 == s1 is true
s1, s2 contents are the same
```

Equality operators on String objects do not work as you might have expected. s1 == s2 tests whether s1 and s2 refers to the same object; it does not test whether the two String objects have the same contents. To test on contents, use the String method String.equals() as shown above.

A combination of relational operators can be used to perform a test. Since equality operators can be applied on Boolean values, the following example is valid:

```
if (a < 0 == b < 0)
  doA();
else
  doB();
```

a < 0 and b < 0 are first evaluated to return Boolean values. The Boolean values are then evaluated and if it is true, doA() is executed, otherwise, doB() is executed.

3.3.5 Bitwise Operators

Bitwise operators operate on the corresponding bits of byte, short, int, and long.

TABLE 3.7: Bitwise Operators

Bitwise Operator	Meaning
&	Bitwise AND
|	Bitwise inclusive OR
^	Bitwise exclusive OR (XOR)
~	Bitwise one's complement
<<	Shift bits left, filling with zero bits on the right-hand side
>>	Shift bits right, filling with the highest (sign) bit on the left-hand side
>>>	Shift bits right, filling with zero bits on the left-hand side

The following example illustrates the use of bitwise operators:

Code 3.2: The BitwiseOp Class
```
class BitwiseOp {

  public static void main(String argv[]) {

    int a = 5; // binary 000000101
    int b = 14;       // binary 000001110
    int c = 0;

    c = a & b;        // returns decimal 4   or binary 000000100
    System.out.println("Decimal c = " + c + "\t\tBinary c = " +
                       Integer.toBinaryString(c));

    c = a ^ b;        // returns decimal 11 or binary 000001011
    System.out.println("Decimal c = " + c + "\t\tBinary c = " +
                       Integer.toBinaryString(c));

    c = a | b;        // returns decimal 15 or binary 000001111
    System.out.println("Decimal c = " + c + "\t\tBinary c = " +
                       Integer.toBinaryString(c));
    c = ~a;            // returns decimal -6 or
                       // binary 11111111111111111111111111110010
    System.out.println("Decimal c = " + c + "\t\tBinary c = " +
                        Integer.toBinaryString(c));

    c = b >> 2;        // returns decimal 3 or binary 000000011
    System.out.println("Decimal c = " + c + "\t\tBinary c = " +
                        Integer.toBinaryString(c));

    c = b << 2;        // returns decimal 46 or binary 000111000
    System.out.println("Decimal c = " + c + "\t\tBinary c = " +
                        Integer.toBinaryString(c));

    c = ~a >> 2;       // returns decimal -2 or
                      // binary 11111111111111111111111111111110
    System.out.println("Decimal c = " + c + "\t\tBinary c = " +
                        Integer.toBinaryString(c));

    c = ~a >>> 2;      // returns decimal 1073741822 or
                       // binary 111111111111111111111111111110
    System.out.println("Decimal c = " + c + "\tBinary c = " +
                        Integer.toBinaryString(c));
  }
}
```

The output from Code 3.2 is:

```
Decimal c = 4           Binary c = 100
Decimal c = 11          Binary c = 1011
Decimal c = 15          Binary c = 1111
Decimal c = -6          Binary c = 11111111111111111111111111111010
Decimal c = 3           Binary c = 11
Decimal c = 56          Binary c = 111000
Decimal c = -2          Binary c = 11111111111111111111111111111110
Decimal c = 1073741822  Binary c = 111111111111111111111111111110
```

3.3.6 The Conditional Operator "?:"

The conditional operator "?:" provides a shortcut to the if…else statement. The operator yields one of two values based on a Boolean expression. For example,

Code 3.3: The CondOp Class

```
class CondOp {

  public static void main(String argv[]) {

    String colour = new String("");
    boolean depressed = true;

    colour = (depressed ? "red" : "black");
    System.out.println("Colour is now " + colour);

    depressed = false;
    colour = (depressed ? "red" : "black");
    System.out.println("Colour is now " + colour);

  }
}
```

The output of Code 3.3 is:

```
Colour is now red
Colour is now black
```

Consider the following use of conditional operator “?:”

```
(a = b ? c : d)
```

The result expressions (i.e. c and d) must have assignment compatible types. For example,

```
double price = (offer ? 10 : 20.50);
```

The result expressions are int (10) and double (20.50). Since int is assignable to double, the int 10 becomes 10.0 and the result returned is a double. If neither side is assignable to the other, the operation is invalid.

The “?:” operator is also known as the *ternary operator* since the operator operates on 3 operands. The operator is also the only ternary operator in Java.

3.3.7 Assignment Operators

An assignment is an operation that assigns the right-hand-side value of an expression to the variable on the left-hand-side of the expression. In addition, the assigned value is returned as a final result of the assignment. For example,

```
a = b = 3; is equivalent to a = (b = 3);
```

3 is first assigned to b; the assignment returns a value (3) which is subsequently assigned to a.

Unlike arithmetic operations in which values are associated from left to right, assignment operators associate from right to left. We will discuss operator associativity later in this chapter.

The simplest form of assignment operator is the single “=”. However, any arithmetic or binary bitwise operator can be combined with “=” to form another assignment operator. Thus, there is a special form “op=” assignment operator.

```
a op= b;
```

is equivalent to

```
a = a op b;
```

where op may be operators such as +, -, *, /, %, >>, <<, &, ^ or |. Thus, we have the following valid assignment operators as shown in Table 3.8.

TABLE 3.8: Assignment Operators

Operator	op=
+	+=
-	-=
*	*=
/	/=
%	%=
>>	>>=
<<	<<=
&	&=
^	^=
|	|=

Assume that op is + in

```
a op= b;
```

then we have:

```
a += b;
```

which is equivalent to:

```
a = a + b;
```

Let us consider Code 3.4 as an example. When the code is evaluated, the result yields

```
b = -6
```

Why is this so?

```
b -= 10 + 2;
```

is equivalent to

```
b = b - (10 + 2);
```

and not

```
b = b - 10 + 2;
```

which yields

```
b = -2
```

Code 3.4: The AssignmentOp Class

```
class AssignmentOp {

  public static void main(String argv[]) {

    int b = 6;

    b -= 10 + 2;
    System.out.println("b = " + b);
  }
}
```

3.3.8 "+" Operator

The "+" operator is overloaded in Java i.e. it can be used in *integer arithmetic* or *string concatenation* as we have seen in previous examples.

When "+" operator is used in integer arithmetic, it takes in two operands and returns an integer or float value depending on the type of operands involved in the addition.

For example, in the following case, the "+" operator is used as an addition operator:

```
int a = 5;
int b = 6;
int c = a + b; // returns c with value 11
```

We can also use "+" to concatenate strings together.

For example, in Code 3.5, the output produced is:

```
The title of this book is So You Want To Learn Java
```

Code 3.5: The PlusOp Class

```
class PlusOp {

  public static void main(String argv[]) {
    String part1 = "So You Want ";
    String part2 = "To Learn ";
    String part3 = "Java";
    String title = part1 + part2;
    title += part3;
    System.out.println("The title of this book is " + title);
  }
}
```

3.3.9 "." Operator

Last but not least, there is the "." operator. This operator is used to denote membership in objects. Members of objects (variables and methods) are accessed using the "." operator. For example, the add() method in object myCounter is accessed via myCounter.add(). Also, to access a variable count in the object myCounter, we write myCounter.count.

3.3.10 Precedence and Associativity

Precedence and Associativity are two rules on operators that ensure there is no ambiguity in interpreting how an expression of operators and operands are evaluated. To appreciate these concepts, let us consider the following expression:

```
a - b * c + d / e
```

Operators and operands are nested in this expression. How should we interpret it? Let's assume the following values for the operands: a=7, b=2, c=4, d=8 and e=2. The expression is now:

```
7 - 2 * 4 + 8 / 2
```

We can evaluate this expression in three ways:

1. Evaluating from left to right we get: (((7 - 2) * 4) + 8) / 2 or 14

2. Evaluating from right to left we get: 7 - (2 * (4 + (8 / 2))) or -9
3. We may also evaluate it as (7 - (2 * 4)) + (8 / 2) or 3

Which of these evaluations is correct?

Java manages this issue with *operator precedence*. Some operators have higher precedence over the others. Operators of higher precedence are evaluated before operators with lower precedence.

Table 3.9 shows the list of operator precedence. The precedence level decreases as we go down the table. All assignments are therefore evaluated last with respect to the other operators present in an expression.

We will use Table 3.9 to evaluate the expression:

```
7 - 2 * 4 + 8 / 2
```

TABLE 3.9: Operator Precedence

Operator Type	Operator
Unary	[] . (*params*) E++ E--
Unary	*unary operators:* -E !E ~E ++E --E
Object creation	new (*type*)E
Arithmetic	* / %
Arithmetic	+ -
Bitwise	>> << >>>
Relational	< > <= >=
Relational	== !=
Bitwise	&
Bitwise	^
Bitwise	|
Logical	&&
Logical	||
Conditional	? :
Assignment	= += -= *= /= %= >>= <<= &= ^= |=

The code to evaluate the above expression is given in Code 3.6.

Code 3.6: The Precedence Class

```
class Precedence {

  public static void main(String argv[]) {
    int a = 7;
    int b = 2;
    int c = 4;
    int d = 8;
    int e = 2;

    int result = a - b * c + d / e;

    System.out.println("Value of a - b * c + d / e = " + result);
  }
}
```

"*" and "/" are of higher precedence than "+" and "-". Therefore, the third evaluation approach is used. The expression evaluates into a value 3.

Since * and / are of equal precedence why do we evaluate "*" first before "/"? Evaluating "*" first rather than "/" is based on another rule: *Operator Associativity*. This rule specifies the direction of evaluation i.e. either from left to right or from right to left.

TABLE 3.10: Operator Precedence and Associativity

Operator Type	Operator	Associativity
Unary	[] . (*params*) E++ E--	Right to left
Unary	*unary operators:* -E !E ~E ++E --E	Right to left
Object creation	new (*type*)E	Right to left
Arithmetic	* / %	Left to right
Arithmetic	+ -	Left to right
Bitwise	>> << >>>	Left to right
Relational	< > <= >=	Left to right
Relational	== !=	Left to right
Bitwise	&	Left to right
Bitwise	^	Left to right
Bitwise	|	Left to right
Logical	&&	Left to right
Logical	||	Left to right
Conditional	? :	Left to right
Assignment	= += -= *= /= %= >>= <<= &= ^= |=	Right to left

In Table 3.10, we have added another column indicating operator associativity. According to the table, expressions with "*" and "/" are evaluated from left to right.

Since "*" appears first before "/" in the evaluation of the expression 7 - 2 * 4 + 8 / 2, multiplication is carried out first before the division.

In summary, we note that the rule of operator precedence helps us to evaluate unambiguously a << b + c * d as (a << (b + (c * d))) and the rule of operator associativity is applied when the precedence rule is unable to determine which operator takes precedence (because the operators are of equal level of precedence).

Since the operators in a / b * c % d are of equal level of precedence, we apply operator associativity rule and evaluate a / b * c % d from left to right i.e. ((a / b) * c) % d.

Similarly, with right to left associativity, ~a++ is interpreted as ~(a++) rather than (~a)++.

CHAPTER 4: PROGRAM FLOW CONTROLS

Must statements be executed one after another in a sequential manner in Java? Well, that seems to be the case in the examples so far. You do not always have to code your statements such that they run sequentially.

Java allows you to control how the program is run. There are three types of program flow control mechanisms available in Java: *Sequence*, *Selection*, and *Iteration*.

4.1 Sequence

Statements in a program are executed sequentially in Java. Consider the following code fragment:

```
int a = 5;
int b = 6;
int c = a + b;
System.out.println("Value of c is " + c);
```

Java begins its execution with the first statement int a = 5;. The variable a is assigned the value 5. The next statement int b = 6; is then executed. The variable b is assigned the value 6. Following this, c is assigned the evaluated value of a + b. Finally, the line "Value of c is 11" is output by the System.out.println() method. The program flow is carried out sequentially statement by statement. This form of program flow control is known as *Sequence*.

4.2 Selection

Suppose we want to indicate if c is greater than 10 or less than 10. We can achieve this by modifying the code fragment to:

```
int a = 5;
int b = 6;
int c = a + b;
if (c > 10)
  System.out.println("c is greater than 10");
else
  System.out.println("c is less than 10");
```

The program flow begins as before, statement by statement. But when execution arrives at if (c > 10), the value of c is evaluated. If it is greater than 10, the next statement

```
System.out.println("c is greater than 10");
```

is executed and the output is:

```
c is greater than 10
```

The program terminates. Otherwise, the statement in the else part

```
System.out.println("c is less than 10");
```

is executed and the output is:

```
c is less than 10
```

The program terminates.

The program flow is altered at the if (c > 10) part. Java has to select which of the two parts: ‘if’ part or ‘else’ part to execute depending on the evaluated value of c. This form of program flow control is known as *Selection*.

4.2.1 Block

Suppose we want to initialize the value of c to 0 when it is greater than 10, then an appropriate statement like c = 0; could be inserted at the if part of the if…else statement as follows:

```
int a = 5;
int b = 6;
int c = a + b;

if (c > 10)
  System.out.println(“c is greater than 10”);
  c = 0;
  System.out.println(“c is now “ + c);
else
  System.out.println(“c is less than 10”);
```

When this code is compiled with javac, an error is reported. The ‘else’ part is considered to be without a corresponding ‘if’ part. The statements c = 0; and System.out.println(“c is now ” + c); cause a break in the if…else statement rendering the ‘else’ without an ‘if’. To resolve this, we use a *block* (denoted by a set of braces “{..}”) to contain the three statements as shown in Code 4.1.

Code 4.1: The Block Class
```
class Block {

  public static void main(String argv[]) {
    int a = 5;
    int b = 6;
    int c = a + b;

    if (c > 10) {
      System.out.println("c is greater than 10");
        c = 0;
      System.out.println("c is now " + c);
    }
    else
      System.out.println("c is less than 10");
  }
}
```

The two braces ({..}) indicates the existence of a block. A *block* groups statements together as one statement. It may occur anywhere in the program where statements are valid. Statements within a block are executed sequentially, one at a time. With the use of block, the following is produced as output instead:

```
c is greater than 10
c is now 0
```

Blocks enable statements to be nested within bigger constructs. Through the use of blocks, more complex nested groups of statements are possible.

4.2.2 Types of Selection Statements

We saw earlier that *Selection* allows for conditional statements to be added. Also, the 'if' statement is used to implement Selection in Java.

4.2.2.1 The Simplest 'if' Statement

The simplest form of Selection is an 'if' statement denoted by:

```
if (Boolean-expression)
  statement
```

The statement is executed if the Boolean expression evaluates to true. For example, consider the following code fragment:

```
int a = 5;
int b = 6;
if (a < b)
  c = c +2;
```

Since a is less than b, the 'if' expression evaluates to true and the statement c = c + 2; is executed. It is common for beginners to omit the parenthesis in the 'if' expression:

```
int a = 5;
int b = 6;
if a < b          <- error: parenthesis missing
  c = c +2;
```

4.2.2.2 The 'if…else' Statement

A variation of Selection is the if..else statement. The if…else statement tests on a condition and alters the flow of control by branching it to statements enclosed within the 'if' part (if the condition evaluates to true) or the else part (if the condition evaluates to false). The format of the 'if…else' statement is:

```
if (Boolean-expression)
  statement 1
else
  statement 2
```

The Boolean-expression is first evaluated. If the evaluation is true, statement 1 will be executed otherwise statement 2 will be executed. 'if' statements can be connected with one another via the 'else' part of the 'if' statement to provide for multiple test conditions. For example, the following code determines how the light should change at a traffic junction:

```
if (colour.equals("red") && (change))
  toGreen(); // change from red to green
else
  if (colour.equals("amber") && (change))
    toRed(); // change from amber to red
  else
    if (change) // must be green
      toAmber() // change from green to amber only when time to change
```

4.2.2.3 Binding ‘else’ to ‘if’

Each ‘else’ clause must be bound to an ‘if’ clause. Consider the following code fragment:

```
if (a < 1)
  b = b + 2;
  c = b++;
else c = 5;
```

At first glance, this code may seem valid with 2 statements to be executed if a is less than 1. However, Java reports that the ‘else’ clause does not have a corresponding ‘if’ clause. The reason for the error lies with the statement

```
c = b++;
```

Java treats this statement as a new statement that does not belong to the ‘if’ part of the if…else statement. In fact, the ‘if’ statement is complete without an ‘else’ part as follows:

```
if (a < 1)
  b = b + 2;
```

Since the ‘if’ statement is complete, Java treats the next occurrence of ‘else’ as an unmatched else clause resulting in the error. A solution to this problem is to use block (via braces {…}) to enclose the two statements as follows:

```
if (a < 1) {
  b = b + 2;
  c = b++;
}
else c = 5;
```

In this way, the two statements are considered as a block of two statements belonging to the ‘if’ clause. The ‘else’ then binds with the ‘if’ clause. We discussed *block* earlier in Section 4.2.1.

An ‘else’ clause is always bound to the most recent ‘if’ clause that does not have an ‘else’ clause bound to it. Consider for example, the following code fragment:

```
1. if (a < 1)
2.   b = b + 1;
3.   if (a > 1)
4.     b = b + 2;
5. else b = b + 3;
6.   else b = b + 4;
```

The else b = b + 3; clause (Line 5) in the above code is matched to:

```
if (a > 1) // Line 3
```

and not:

```
if (a < 1) // Line 1
```

Visually, it seems like (because of the indentation) else b = b + 3; is matched to:

```
if (a < 1) // Line 1
```

but this is not the case. Java ignores indentation in its attempt to bind an ‘if’ clause to an ‘else’ clause. In fact,

```
else b = b + 4;
```

is matched to the first ‘if’ clause

```
if (a < 1) // Line 1
```

For clarity, braces should be used and indentation should be done properly as follows:

```
if (a < 1) {
  b = b + 1;
  if (a > 1)
    b = b + 2;
  else b = b + 3;
}
else b = b + 4;
```

It is clear from above that while ‘if’ statements can be nested, multiple nesting levels can make code difficult to read. In addition, poor indentation and lack of clarity make the code looks complex and prone to error. Developers are therefore advised to put in more effort (such as using braces and comments) to make ‘if’ statements as simple and clear as possible. You are also advised to check through ‘if’ statements to ensure if the statements have been formed correctly according to the specification.

4.2.2.4 The switch Statement

Multi-way tests are possible and can be simplified using the switch statement. A *switch statement* can be used in situations where there are many possible cases and corresponding actions for a given expression. The format of a switch statement is as follows:

```
switch (expression-E) {
  case constant-expression-1: statement-1; break;
  case constant-expression-2: statement-2;
  case constant-expression-3: statement-3; return;
  default: throw something;
}
```

There are a few points to note about the above switch statement:

1. The switch statement evaluates an expression (expression-E) and determines which of the constant expression in the case labels matches the expression-E value. The statements in the case branch that matches expression-E are executed. If none of the case expression matches expression-E value, the statements in the default branch are executed.
2. The constant expression, as its name indicates, contains literals or constant values.
3. Values of constant expressions in a single switch statement must be unique.
4. There is at most one default label in a switch statement.
5. There is a fall through in execution from one case branch to another. However, fall through can be prevented using break, return or throw statement. The break statement causes control flow to exit the switch statement and onto the next statement. The return statement causes control flow to exit the switch statement as well as the method containing the switch statement. The throw statement is often used in exception management.

A more concrete example will provide a clearer picture of the switch statement. Let us return to our traffic light change example. We will use a switch statement to determine the action for each colour change and assume a value of 0, 1, 2 for red, amber, green respectively. Any other numeric value will be considered an error. Code 4.2 illustrates our example.

currentValue represents the current value of the traffic light. Given a value for currentValue, Code 4.2 displays the colour of the light.

When Java encounters the return function, control exits and returns to where this code was called i.e. the command prompt.

Code 4.2: Using switch Statement

```
class Switch {

  public static void main(String argv[]) {
        int currentValue = Integer.parseInt(argv[0]);
        int colour = currentValue;
        switch (colour) {
          case 0:  System.out.println("red");
                    break;
          case 1:  System.out.println("amber");
                    return;
          case 2:  System.out.println("green");
          default: System.out.println("error");
        }
        System.out.println("out of switch but still in main()");
  }
}
```

Let us number the Switch class with line numbers:

```
 1. class Switch {
 2.   public static void main(String argv[]) {
 3.     int currentValue = Integer.parseInt(argv[0]);
 4.     int colour = currentValue;
 5.     switch (colour) {
 6.       case 0:  System.out.println("red");
 7.           break;
 8.       case 1:  System.out.println("amber");
 9.           return;
10.       case 2:  System.out.println("green");
11.       default: System.out.println("error");
12.     }
13.     System.out.println("out of switch but still in main()");
14.   }
15. }
```

Table 4.1 illustrates the outputs produced for each value of currentValue. Note how the control flows within the switch statement. A common mistake made by beginners is in the omission of break, return or throw in the multiple case branches.

As shown in Table 4.1, the omission will lead to control flow falling through to the next case label (see the case of green when the default value is also selected).

TABLE 4.1: Outputs from Traffic Light Change Example

When currentValue is	Output	Line
0	red out of switch but still in main()	6, 7, 12, 13, 14, 15
1	amber	8, 9, 14, 15
2	green error out of switch but still in main()	10, 11, 12, 13, 14, 15
others	error out of switch but still in main()	11, 12, 13, 14, 15

4.3 Iteration

The *Sequence* construct allows for statements to be executed as a group and one at a time sequentially. Adding test conditions to select a group of statements to execute is what the *Selection* construct allows for. In both cases, the group of statements is executed at most once. The *Iteration* construct differs from these two constructs. It allows for statements to be executed *repeatedly*.
There are three types of Iteration statement in Java:

1. The while statement
2. The do-while statement
3. The for statement

4.3.1 The while Statement

The most basic Iteration statement is the while statement. It has the following format:

```
while (Boolean expression)
  constituent statement
```

or (for a group of statements using block)

```
while (Boolean expression) {
  constituent statements
}
```

The while statement begins by evaluating the Boolean expression. If the expression evaluates to true, the constituent statement is (or statements are) executed. Control then returns to evaluate the Boolean expression again. The constituent statement is executed so long as the Boolean expression evaluates to true. This process repeats itself until the Boolean expression evaluates to false; control then exits from the while statement. The next statement after the while statement is subsequently executed.

Let's consider an application of the while statement:

Code 4.3: Using while Statement
```
class While {

  public static void main(String argv[]) {
    int number = 0;
    while (number < 10) {
      System.out.println(number);
      number++;
    }
    System.out.println("Out of while statement");
  }
}
```

Code 4.3 prints a series of numbers from 0 to 9 while the while statement is true. The value of number is tested each time the Boolean expression

```
(number < 10)
```

is evaluated. Since number is initialized to 0, the first test on the Boolean expression returns true. The constituent statements execute and a number is printed. The variable number is incremented via number++;. Control returns to evaluate the Boolean expression again. Since number is now 1 which is less than 10, the constituent statements execute once again and incrementing number (via number++;) in the process. This process repeats until number reaches 10. When this happens, the Boolean expression evaluates to false. The next statement

```
System.out.println(“Out of while statement”);
```

then prints the output:

```
Out of while statement
```

The while statement is often referred to as the *pre-test iteration* since the constituent statements are only executed if the Boolean expression evaluates to true. This means that the constituent statements are never executed if the Boolean expression returns false in the first expression evaluation.

4.3.2 The do-while Statement

The do-while statement is slightly different from the while statement. It has the following format:

```
do {
  constituent statements
} while (Boolean expression);
```

The constituent statements are first executed followed by a test on the Boolean expression. If the Boolean expression evaluates to true, control returns to execute the constituent statements again. This process of executing constituent statements and then evaluating the Boolean expression is repeated until the Boolean expression evaluates to false. The iteration halts and the next statement after the do-while statement is executed. Note that the constituent statements are executed at least once, even when the Boolean expression is tested to be false subsequently. Let us consider an application of the do-while statement.

Code 4.4 prints a series of numbers from 0 to 9, similar to Code 4.3. However, for this case, the constituent statements are executed before the Boolean expression number < 10 is tested. Since the test evaluates to true, the do-while statement repeats itself and only exits when the Boolean expression evaluates to false (i.e. when number equals 10).

Code 4.4: Using do-while Statement
```
class DoWhile {

  public static void main(String argv[]) {
    int number = 0;
    do {
      System.out.println(number);
      number++;
    } while (number < 10);
    System.out.println("Out of while statement");
  }
}
```

To illustrate the effect of do-while statement, let us initialize number to 10 as shown in Code 4.5.

Code 4.5: number initialized to 10
```
class DoWhile {

  public static void main(String argv[]) {
    int number = 10;
    do {
      System.out.println(number);
      number++;
    } while (number < 10);
    System.out.println("Out of while statement");
  }
}
```

Notice the body of the do-while statement is executed at least once with number being equal to 10 even though the Boolean expression is evaluated to be false subsequently. The output from Code 4.5 is:

```
10
Out of while statement
```

Since the constituent statements are executed before the condition test is evaluated, the do-while statement has been known as the *post-test iteration*.

4.3.3 The for Statement

The functionality of the while statement in Code 4.3 can also be achieved using the for statement. It has the following format:

```
for (initialization expression; Boolean expression; increment expression)
  constituent statement
```

or (for a group of statements using block)

```
for (initialization expression; Boolean expression; increment expression) {
  constituent statements
}
```

The for statement is used to execute statements over a range of values. It behaves in the following manner:

1. The initialization expression sets an initial value to the range counter.
2. A test is carried out using the Boolean expression each time the for loop is executed. If the Boolean expression evaluates to true, the constituent statements in the for statement are executed.
3. The increment expression is automatically incremented when the constituent statements complete. The Boolean expression is then tested to determine if the loop should repeat.
4. The for statement exits when the Boolean expression evaluates to false.
5. The next statement after the for statement is then executed.

Typically the for statement is used to iterate a variable *over a range of values* until some logical end to that range is reached. Let us consider an example in Code 4.6.

Code 4.6: Using the for Statement

```
class For {

  public static void main(String argv[]) {
    for (int number=0; number<10; number++)
      System.out.println(number);
    System.out.println("Out of for statement");
  }
}
```

In executing the for statement, number is first set to 0. The number<10 expression is evaluated. Since 0 is less than 10, the statement System.out.println(number); in the for loop is executed. The value 0 is printed.

Next, the update expression number++ is evaluated. This increments number to 1. The variable number is once again evaluated in the Boolean expression number<10. Since 1 is less than 10, the statement in the for loop is executed and the value 1 is printed. This process repeats until number equals 10 and the loop terminates resulting in the following output (Notice the output is similar to Code 4.3 and 4.4):

```
0
1
2
3
4
5
6
7
8
9
Out of for statement
```

4.3.3.1 Infinite Loop

The three expressions in a for statement: Initialization Expression, Boolean Expression and Update Expression are optional. If initialization expression and update expression are left out, their parts in the loop are simply omitted. If the Boolean expression is left out, the expression is said to evaluate to true. Assuming we have the three expressions left out, we get:

```
for ( ; ; )
  statement
```

Since the for statement is unable to test the Boolean expression, it assumes the value true. Furthermore, without a terminating condition, the for statement simply repeats itself indefinitely. In other words, we get an infinite loop with no ending at all. For example, the following code prints "Forever Friends" repeatedly forever.

```
for ( ; ; )
  System.out.println("Forever Friends");
```

4.3.4 The Enhanced 'for' Statement

Beginning Java 5 (i.e. JDK 1.5 or J2SE 5), a new form of enhanced 'for' statement is available. This new form provides a for-each capability designed to cycle through a collection of objects, such as an array, in strictly sequential fashion, from start to finish. It has the following format:

```
for (type iteration-variable : iterableObject)
  constituent statement
```

or (for a group of statements using block)

```
for (type iteration-variable : iterableObject) {
  constituent statements
}
```

type specifies the type and this type must be the same as the type of the elements retrieved from iterableObject, iteration-variable specifies the name of an iteration variable that will receive the elements contained in iterableObject, one at a time, from beginning to end. iterableObject must be an array or an object that implements the new Iterable interface.

This is how it works. With each iteration of the loop, the next element in iterableObject is retrieved and stored in iterable-variable. The loop repeats until all elements have been obtained.

We will discuss with an example how enhanced 'for' statement works in Chapter 5 after we have discussed the concept of arrays in Java.

4.3.5 Labels

Consider a situation where you want to force control to a certain statement in a program. How do you do it? Of course, you need to know the statement in the first place so that you can inform Java which statement to direct control to. Java provides a facility for statements to be labeled so that we can force control to "jump" to the statement when required. The facility is in the form of *label* and it has the following format:

```
label: statement
```

For example, the following statement is labeled as again,

```
again: a = a + b; // again labels the statement a = a + b;
```

Typically, labels are used on blocks and loops (or iterations), and in conjunction with the break statement.

4.4 The break Statement

A break statement is used to force an exit from a block.

4.5 The continue Statement

The continue statement is used inside loops – while, do-while, and for – to allow flow of control to skip to the end of a loop without executing any of the statements within the loop. We will illustrate the use of the break and continue statements, and label within the examples of arrays when we discuss them in Chapter 5.

CHAPTER 5: ARRAYS

An *array* is a data structure for storing and manipulating data of the same type as a collection. For each array, there exists an *array variable* that is used as a reference variable (or index) on the array.

5.1 Declaring and Creating an Array

An array must be declared before it can be used in a program. There are three steps involved in array creation:

1. Declare an array variable
2. Create an array via the new keyword
3. Assign the reference of the newly created array to the array variable

There are two ways to declare an array variable:

```
dataType[] arrayName;
```

or

```
dataType arrayName[];
```

The former approach (using dataType[]) is preferred. For example, to declare an array of module marks of int type:

```
int[]  marks;
```

or

```
int    marks[];
```

An array variable keeps a reference to an array. The declaration of an array variable does not take any memory space and the array variable has a value null if it does not reference any array.

No elements can be assigned to an array if it has not been created. To create an array, use the new keyword:

```
arrayName = new dataType[size];
```

For example,

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

creates an array referenced by (or simply, named) marks, with five elements of int type. Alternatively, the two statements can be combined into one:

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

Technically, the above combined statement declares an array variable marks, create an array of five elements and assign the reference of the array to the array variable as illustrated in Figure 5.1.

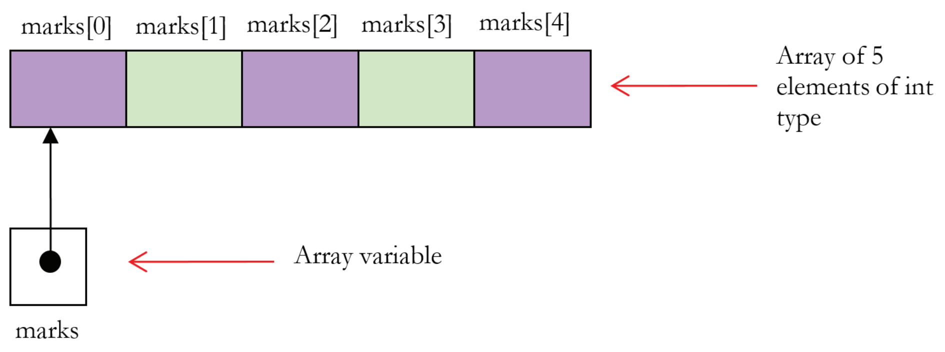


FIGURE 5.1: Creating an Array

Java indexes elements beginning from 0; therefore, the first element is referenced as int[0] followed by int[1], and so on; the index of the last element is arraySize-1. Each element in an array occupies memory space. The larger the array, the larger the amount of memory space is required to store the elements.

The size of an array is fixed and determined when the array is created. Java provides a variable, arrayName.length, to obtain the size of an array. For example, the value of marks.length is 5.

5.2 Initializing an Array

When an array is created, the elements are assigned default values for its primitive types: 0 for int, 0.0 for double, '\u0000' for char, and false for Boolean type. These values may be changed by assigning new values to the elements. For example,

```
marks[0] = 89;
```

assigns a new value 89 to the first element of the marks array. The other 4 elements may similarly be assigned new values as follows:

```
marks[1] = 56;
marks[2] = 88;
marks[3] = 35;
marks[4] = 65;
```

Java provides a short cut to assigning values at array creation time. Instead of using the new keyword, we use the braces {..} :

```
int[] marks = {89, 56, 88, 35, 65};
```

The first element of the marks array, marks[0], is assigned the value 89 and the last element, marks[4], has the value 65. The length of the array (or marks.length) is 5.

5.3 Using Arrays

Let us consider an application of arrays. This application involves the creation of two arrays: one array is of String type containing students' names while the other array is of int type containing the marks obtained by each student. The application prints the marks awarded to the students.

Code 5.1 is an implementation of the requirement. Both approaches of creating and initializing arrays have been implemented in Code 5.1.

```
String students[] = {"Benny", "Robin", "Sally", "Aaron", "Simon"};
```

uses the braces approach to creating and initializing the students array. The size of the students array is determined by the number of elements within the braces {..}.

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

creates a marks array of 5 elements.

```
marks[0] = 89;
marks[1] = 56;
marks[2] = 88;
marks[3] = 35;
marks[4] = 65;
```

initializes the marks array with values. Note that the elements have to be individually assigned. As we will see later, the for statement is a good construct for manipulating arrays.

Code 5.1: Students and Marks Array

```
class StudentsMarksArray {

  public static void main(String args[]) {
    String students[] = {"Benny", "Robin", "Sally", "Aaron", "Simon"};
    int marks[]       = new int[5];
    marks[0]          = 89;
    marks[1]          = 56;
    marks[2]          = 88;
    marks[3]          = 35;
    marks[4]          = 65;
    int i = 0;

    System.out.println("\nMarks for students:");

    for (i=0; i<students.length; i++) {
      System.out.println(students[i] + " -> " + marks[i]);
    }
  }
}
```

The following output is produced by the System.out.println() statement via the for loop:

```
Marks for students:
Benny -> 89
Robin -> 56
Sally -> 88
Aaron -> 35
Simon -> 65
```

The for loop is often used in conjunction with arrays. The reasons are simple:

1. The size of arrays is fixed, this is congruent with the nature of for loop
2. Elements in arrays are of a uniform type and can be evenly processed in the same fashion using a for loop

Note the use of students.length which is the number of elements in the array in the above example.

5.4 Two-dimensional Arrays

In our previous example of students and marks, we used two arrays: students (to keep the names of students) and marks (to keep the mark of the corresponding student in students array). That solution works well since there is only 1 mark for each student. Consider a situation where there are 3 marks for each student (e.g. one for each subject taken).

There are 2 approaches we can consider:

1. Create 3 one-dimensional marks array, one for each subject marks
2. Use a matrix to contain all subject marks. A matrix can be represented with a two-dimensional array.

5.4.1 One-dimensional Array Approach

Code 5.2 illustrates the one-dimensional array approach.

Code 5.2: One-dimensional Array Approach
```
class MarksArray {

  public static void main(String args[]) {
    String students[]  = {"Benny", "Robin", "Sally", "Aaron", "Simon"};
    int mathsMarks[]   = {89, 56, 88, 35, 65};
    int englishMarks[] = {45, 86, 81, 75, 62};
    int scienceMarks[] = {55, 23, 73, 39, 77};

    int i = 0;

    System.out.println("\nMarks for students:");

    for (i=0; i<students.length; i++) {
      System.out.print(students[i] + " -> " +
                       "maths=" + mathsMarks[i] + "  " +
                       "english=" + englishMarks[i] + "  " +
                       "science=" + scienceMarks[i]);
      System.out.println();
    }
  }
}
```

Four arrays are declared to store the names and marks of students. The solution is very simple – it prints the marks obtained by each student. How the code is processed is left as an exercise to the reader. The output is as follows:

```
Marks for students:
Benny -> maths=89  english=45  science=55
Robin -> maths=56  english=86  science=23
Sally -> maths=88  english=81  science=73
Aaron -> maths=35  english=75  science=39
Simon -> maths=65  english=62  science=77
```

5.4.2 Two-dimensional Array Approach

The second approach uses a matrix to store the marks obtained by the students for the various subjects. The code is given in Code 5.3 which illustrates the two-dimensional array approach.

The output is the same as the one-dimensional array approach:

```
Marks for students:
Benny -> maths=89  english=45  science=55
Robin -> maths=56  english=86  science=23
Sally -> maths=88  english=81  science=73
Aaron -> maths=35  english=75  science=39
Simon -> maths=65  english=62  science=77
```

Code 5.3: Two-dimensional Array Approach

```
class MarksMatrix {

  public static void main(String args[]) {
    String students[] = {"Benny", "Robin", "Sally", "Aaron", "Simon"};
    int marks[][]     = {{89, 45, 55},
                         {56, 86, 23},
                         {88, 81, 73},
                         {35, 75, 39},
                         {65, 62, 77}
                        };
    int i = 0;
    int j = 0;

    System.out.println("\nMarks for students:");

    for (i=0; i<marks.length; i++) { // number of students
      System.out.print(students[i] + " -> " +
                       "maths=" + marks[i][j] + "  " +
                       "english=" + marks[i][j+1] + "  " +
                       "science=" + marks[i][j+2]);
      System.out.println();
    }
  }
}
```

Let us review Code 5.3. A students array is declared to maintain the names of the students. A marks matrix is next declared. A matrix is a two-dimensional table and is implemented in Java as an array of arrays (two-dimensional array structure) as follows:

```
int[][] marks;
```

or

```
int marks[][];
```

To create a two-dimensional array, we use the usual new keyword:

```
marks = new int[5][3];
```

Alternatively, we may use the shorthand notation to declare, create and initialize the matrix:

```
int[][] marks = {{89, 45, 55},
                 {56, 86, 23},
                 {88, 81, 73},
                 {35, 75, 39},
                 {65, 62, 77}
                };
```

The marks matrix is an array containing 5 elements that are arrays themselves. The inner arrays contain 3 elements each. To help you remember the representation of the elements in a matrix, consider the representation as:

```
marks[row][column];
```

Therefore, marks[2][1] refers to the third row (since row 1 starts from 0) and second column element. marks[2][1] has the value 81 and marks[1][2] has the value 23. Figure 5.2 shows the marks matrix.

	[0]	[1]	[2]
[0]	89	45	55
[1]	56	86	**23**
[2]	88	**81**	73
[3]	35	75	39
[4]	65	62	77

int[][] marks = new int[5][3];

marks[2][1] = 81;

marks[1][2] = 23;

FIGURE 5.2: A Two-dimensional Array

In the for loop, the variable marks.length refers to the number of arrays in the single array declared in marks:

```
int[][] marks = {{89, 45, 55},
                 {56, 86, 23},
                 {88, 81, 73},
                 {35, 75, 39},
                 {65, 62, 77}
                };
```

Since there are 5 arrays, marks.length is equal to 5. Within the for loop, the elements in the matrix are visited one at a time and printed.

5.4.3 Populating Two-dimensional Arrays

The for statement is a suitable construct for populating a two-dimensional array. Code 5.4 illustrates an application of for loop for populating a two-dimensional array.

Code 5.4: Populating a Two-dimensional Array

```
class PopulatingTwoDimensionalArray {

  public static void main(String argv[]) {
    int[][] matrix = new int[10][10];
    for (int i=1; i<10; i++) {
      for (int j=0; j<10; j++) {
        matrix[i][j] = (i*10) + j;
        System.out.print(matrix[i][j]);
      }
      System.out.println();
    }
    System.out.println("Out of for statement");
  }
}
```

A matrix (containing two arrays) is first created. The first for statement loops through the matrix 9 times while the second, inner for statement loops through 10 times. The cell value of the matrix is printed in the inner for statement each time the inner for statement is called. The two for loops repeat themselves until all the elements of the matrix are initialized and printed.

Code 5.4 produces the following output:

```
10111213141516171819
20212223242526272829
30313233343536373839
40414243444546474849
50515253545556575859
60616263646566676869
70717273747576777879
80818283848586878889
90919293949596979899
Out of for statement
```

5.5 Applying the Enhanced 'for' Statement

We are now ready to explain how the enhanced 'for' statement can be used. Let us consider an example in Code 5.5.

Code 5.5: Using the Enhanced 'for' Statement
```
class ForEach {

  public static void main (String args[]) {
    char[] chars = {'A', 'B', 'C', 'D', 'E'};

    for (char c : chars)
      System.out.print(c + " ");
    System.out.println("\nArray chars has " + chars.length + " characters");
  }
}
```

We define an array of characters. Using the enhanced 'for' statement, we print each character from the beginning to the end of the array. char defines the type of the iteration variable c. With each iteration of the loop, the next element in chars array is retrieved and stored in iteration variable c. The loop repeats until all elements in chars have been obtained. The output of the code is given below:

```
A B C D E
Array chars has 5 characters
```

5.6 An Application: Printing Numbers Divisible by 3

Code 5.6 prints all numbers in the range 10 to 99 that are divisible by 3.

The code begins with the initialization of a matrix with numbers starting from 10 and ending with 99. The second segment of the code tests all numbers and prints them if the remainder of dividing a number by 3 is 0.

The output from the code is:

```
10111213141516171819
20212223242526272829
30313233343536373839
40414243444546474849
50515253545556575859
60616263646566676869
70717273747576777879
80818283848586878889
90919293949596979899

Numbers divisible by 3 are:
```

```
12 15 18
21 24 27
30 33 36 39
42 45 48
51 54 57
60 63 66 69
72 75 78
81 84 87
90 93 96 99
```

Code 5.6: Divisible by 3

```
class DivisibleBy3 {

  public static void main(String argv[]) {
    // initialize matrix with values
    int[][] matrix = new int[10][10];
    for (int i=1; i<10; i++) {
      for (int j=0; j<10; j++) {
        matrix[i][j] = (i*10) + j;
        System.out.print(matrix[i][j]);
      }
      System.out.println();
    }

    System.out.println("\nNumbers divisible by 3 are:");

    for (int i=1; i<10; i++) {
      for (int j=0; j<10; j++) {
        if (matrix[i][j]%3 == 0) {
          System.out.print(matrix[i][j] + " ");
        }
      }
      System.out.println();
    }
  }
}
```

5.6.1 Using Label and break Statement

Suppose we want to print only the first occurrence of a number divisible by 3 for each row of the numbers instead of for every number divisible by 3 as we did earlier. The desired output is:

```
10111213141516171819
20212223242526272829
30313233343536373839
40414243444546474849
50515253545556575859
60616263646566676869
70717273747576777879
80818283848586878889
90919293949596979899

Numbers divisible by 3 are:

12
21
30
42
51
60
72
81
```

```
90
```

To solve this problem, we will use the label facility (see Section 4.4) and the break statement – see Code 5.7.

Code 5.7: Label and break Statement
```
class Break {

  public static void main(String argv[]) {
    // initialize matrix with values
    int[][] matrix = new int[10][10];
    for (int i=1; i<10; i++) {
      for (int j=0; j<10; j++) {
        matrix[i][j] = (i*10) + j;
        System.out.print(matrix[i][j]);
      }
       System.out.println();
    }

    System.out.println("\nNumbers divisible by 3 are:");

    for (int i=1; i<10; i++) {
      inner:
        for (int j=0; j<10; j++) {
          if (matrix[i][j]%3 == 0) {
            System.out.print(matrix[i][j] + " ");
            break inner;
          }
        }
      System.out.println();
      System.out.println("Out of inner for loop");
    }
    System.out.println("Out of outer for loop");
  }
}
```

We have modified the previous code slightly to include a label inner, a break statement and two println statements "Out of inner for loop" and "Out of outer for loop". The addition of these two println statements is to help us in the understanding of the flow of execution.

The label inner identifies the inner for loop. When the break statement is encountered, control exits from the if block and the inner for loop since the break statement is identified with the inner label. The output from this code:

```
10111213141516171819
20212223242526272829
30313233343536373839
40414243444546474849
50515253545556575859
60616263646566676869
70717273747576777879
80818283848586878889
90919293949596979899

Numbers divisible by 3 are:

12
Out of inner for loop
21
Out of inner for loop
30
Out of inner for loop
42
Out of inner for loop
```

```
51
Out of inner for loop
60
Out of inner for loop
72
Out of inner for loop
81
Out of inner for loop
90
Out of inner for loop
Out of outer for loop
```

Readers might be interested to experiment with the flow of control by shifting the inner label out of the inner for loop and identify it with the outer for loop. In this case, the break statement will exit from the outer for loop, bypassing all statements within the inner and outer for loops.

5.6.2 Using continue Statement

Suppose we want to print all numbers that are *not* divisible by 3 for each row of numbers from 10 to 99. The expected output looks like this:

```
10111213141516171819
20212223242526272829
30313233343536373839
40414243444546474849
50515253545556575859
60616263646566676869
70717273747576777879
80818283848586878889
90919293949596979899

Numbers NOT divisible by 3 are:

10 11 13 14 16 17 19
20 22 23 25 26 28 29
31 32 34 35 37 38
40 41 43 44 46 47 49
50 52 53 55 56 58 59
61 62 64 65 67 68
70 71 73 74 76 77 79
80 82 83 85 86 88 89
91 92 94 95 97 98
```

To achieve this, we have to test each number, and if the number is not divisible by 3, print it; otherwise, exit the loop without executing the print statement. The continue statement is used in Code 5.8 to achieve this.

Code 5.8: Using the continue Statement

```
class Continue {

  public static void main(String argv[]) {
    // initialize matrix with values
    int[][] matrix = new int[10][10];
    for (int i=1; i<10; i++) {
      for (int j=0; j<10; j++) {
        matrix[i][j] = (i*10) + j;
        System.out.print(matrix[i][j]);
      }
      System.out.println();
    }

    System.out.println("\nNumbers NOT divisible by 3 are:");
    for (int i=1; i<10; i++) {
      inner:
        for (int j=0; j<10; j++) {
          if (matrix[i][j]%3 == 0) {
            continue inner;
          }
          System.out.print(matrix[i][j] + " ");
        }
      System.out.println();
      System.out.println("Out of inner for loop");
    }
    System.out.println("Out of outer for loop");
  }
}
```

The output from Code 5.8 is as follows:

```
10111213141516171819
20212223242526272829
30313233343536373839
40414243444546474849
50515253545556575859
60616263646566676869
70717273747576777879
80818283848586878889
90919293949596979899

Numbers NOT divisible by 3 are:

10 11 13 14 16 17 19
Out of inner for loop
20 22 23 25 26 28 29
Out of inner for loop
31 32 34 35 37 38
Out of inner for loop
40 41 43 44 46 47 49
Out of inner for loop
50 52 53 55 56 58 59
Out of inner for loop
61 62 64 65 67 68
Out of inner for loop
70 71 73 74 76 77 79
Out of inner for loop
80 82 83 85 86 88 89
Out of inner for loop
91 92 94 95 97 98
Out of inner for loop
Out of outer for loop
```

When the number tested is divisible by 3, the continue statement in the if statement is executed. Upon encountering continue, control is transferred to reevaluate the Boolean condition of the for loop for the next iteration of j. In the process, the print statement System.out.print(matrix[i][j] + " "); is ignored. In other words, a labeled continue will break out of any inner loops and carry on with the next iteration of the named loop. **Unlike break, continue does not exit a loop – it continues with the next iteration.**

Again, the two additional println statements on "Out of inner for loop" and "Out of outer for loop" are meant to highlight the flow of execution.

CHAPTER 6: METHODS

All the programs we have written so far have been executed within the main() method. Although there is nothing wrong with this approach of programming applications, this solution is limited and does not scale up as the application becomes more and more complex. For real-life applications where the length of code can go into thousands or even millions lines of code, writing every line of code within a single main() method is certainly not the way to producing good and maintainable code.

An approach commonly used to manage program complexity is divide and conquer. This phrase (in Latin divide et impera) reflects an approach practised by the ancient Romans in their conquest of other nations – divide and isolate and then conquer them one by one. In software program development, this approach is translated into "divide a larger problem into few smaller sub-problems; solve the sub-problems individually and the larger problem is said to be solved when all the sub-problems are solved".

We will apply the divide and conquer strategy in our approach to reducing program complexity in Java programming. Let us define a problem and discuss how we can apply this approach to make Java programs more structured.

6.1 Defining a Problem

A class consists of 20 students and each student takes 3 modules. A program is required to produce the equivalent grade for each mark obtained based on the following table:

Grade	From	To
A	80	100
B	65	79
C	50	64
D	40	49
F	0	39

An average score based on three module marks is calculated for each student. The program is to generate a printout of modules and their respective marks and grades for each student. A sample output of two students' marks and grades for each of the module taken is illustrated below:

```
Mark Chua
        Chemistry       77      B
        Mathematics     80      A
        Physics         60      C
        Average 72      B
Simon Goh
        Chemistry       65      B
        Mathematics     45      D
        Physics         89      A
        Average 66      B
```

Inputs to the program are captured as a sequence of 4 items: name, module 1 mark, module 2 mark, module 3 mark.

6.2 A Problem Solving Approach

We will solve this problem using a step-wise refinement approach (a divide and conquer approach). The larger problem of "processing students' examination grades" is divided into the following sub-problems or steps:

1. Create data structures for storing data values
2. Read in data
3. Convert module marks into grades for one student
4. Print student's name, modules, marks and grades
5. Repeat step 2, 3 and 4 until all students are processed

The above steps can be translated into pseudo code (i.e. non-executable, representative code) as follows:

```
1.  initialize internal data values
2.  for each student {
3.    read module marks
4.    convert marks into grades
5.    print student's name, modules, marks and grades
6.  }
```

The pseudo code is then translated into actual Java implementation as given in Code 6.1.

Code 6.1: Students' Examination Grades

```
class GradesA {

  static String[] students      = new String[20];
  static int[] chemMarks        = new int[20];
  static int[] mathsMarks       = new int[20];
  static int[] physicsMarks     = new int[20];
  static int[] averageMarks     = new int[20];
  static String[] chemGrades    = new String[20];
  static String[] mathsGrades   = new String[20];
  static String[] physicsGrades = new String[20];
  static String[] averageGrades = new String[20];
  static int      k             = 0;

  public static void main(String argv[]) {

    // initialize students array
    for (int j=0; j<20; j++) {
      students[j]      =  new String();
      chemGrades[j]    =  new String();
      mathsGrades[j]   =  new String();
      physicsGrades[j] =  new String();
      averageGrades[j] =  new String();
    }

    // produce students' examination grades
    for (int i=0; i<(argv.length/4); i++) {
      // read input values
      students[i]     = argv[k];
      chemMarks[i]    = Integer.parseInt(argv[k+1]);
      mathsMarks[i]   = Integer.parseInt(argv[k+2]);
      physicsMarks[i] = Integer.parseInt(argv[k+3]);
      averageMarks[i] = (chemMarks[i] + mathsMarks[i] + physicsMarks[i])/3;
      k               = k + 4;

      // convert marks to grades
      if (chemMarks[i] >= 80 && chemMarks[i] <= 100)
        chemGrades[i] = "A";
      if (chemMarks[i] >= 65 && chemMarks[i] <= 79)
        chemGrades[i] = "B";
      if (chemMarks[i] >= 50 && chemMarks[i] <= 64)
        chemGrades[i] = "C";
      if (chemMarks[i] >= 40 && chemMarks[i] <= 49)
        chemGrades[i] = "D";
      if (chemMarks[i] >= 0 && chemMarks[i] <= 39)
        chemGrades[i] = "F";
      if (mathsMarks[i] >= 80 && mathsMarks[i] <= 100)
        mathsGrades[i] = "A";
      if (mathsMarks[i] >= 65 && mathsMarks[i] <= 79)
        mathsGrades[i] = "B";
      if (mathsMarks[i] >= 50 && mathsMarks[i] <= 64)
        mathsGrades[i] = "C";
      if (mathsMarks[i] >= 40 && mathsMarks[i] <= 49)
        mathsGrades[i] = "D";
      if (mathsMarks[i] >= 0 && mathsMarks[i] <= 39)
        mathsGrades[i] = "F";
      if (physicsMarks[i] >= 80 && physicsMarks[i] <= 100)
        physicsGrades[i] = "A";
      if (physicsMarks[i] >= 65 && physicsMarks[i] <= 79)
        physicsGrades[i] = "B";
      if (physicsMarks[i] >= 50 && physicsMarks[i] <= 64)
        physicsGrades[i] = "C";
      if (physicsMarks[i] >= 40 && physicsMarks[i] <= 49)
        physicsGrades[i] = "D";
      if (physicsMarks[i] >= 0 && physicsMarks[i] <= 39)
        physicsGrades[i] = "F";
      if (averageMarks[i] >= 80 && averageMarks[i] <= 100)
        averageGrades[i] = "A";
      if (averageMarks[i] >= 65 && averageMarks[i] <= 79)
        averageGrades[i] = "B";
      if (averageMarks[i] >= 50 && averageMarks[i] <= 64)
        averageGrades[i] = "C";
      if (averageMarks[i] >= 40 && averageMarks[i] <= 49)
        averageGrades[i] = "D";
      if (averageMarks[i] >= 0 && averageMarks[i] <= 39)
        averageGrades[i] = "F";

      // print student's name, modules, marks and grades
      System.out.println(students[i]);
      System.out.println("\tChemistry \t" + chemMarks[i] +
                         "\t" + chemGrades[i]);
      System.out.println("\tMaths \t\t" + mathsMarks[i] +
                         "\t" + mathsGrades[i]);
      System.out.println("\tPhysics \t" + physicsMarks[i] +
                         "\t" + physicsGrades[i]);
      System.out.println("\tAverage \t" + averageMarks[i] +
                         "\t" + averageGrades[i]);
    }
  }
}
```

Data structures in the form of arrays are declared to contain the various values/grades for module marks and students. Note the use of static in the data declaration. Declaring data as static ensures that the data are available to any part of the program (or class as in the case of Java).

Code 6.1 contains two for loops. The first for loop initializes 5 internal arrays for storing information about students and their respective module marks and grades. The second for loop processes the marks and grades for all students. Each iteration of the second for loop prints the marks and grades of the modules taken by each student. The program exits when all the students have been processed. Specifically, the followings are carried out in the second for loop:

1. *Input values are read into the various arrays*. The input values are entered via the command line prompt in the following format: *student's name, Chemistry module mark, Maths module mark, Physics module mark*. These values are stored in the argv array. As mentioned previously, the argv array is the conduit for channeling inputs into Java programs via the command line prompt. Since argv array is declared as a String array, there is a need to convert String values into int values. Integer.parseInt() is

a method in the Java class library for converting String values into int values. This method is part of the java.lang package which comes with the Java Software Development Kit (SDK). The average mark is calculated and stored in the respective array.
2. Marks are converted into grades based on the given grade table.
3. The required output is produced and printed.
4. The process is repeated for every student.

6.3 Improving the Problem-Solving Approach

Although the above solution works, it is tedious and is error prone. A better solution makes use of methods. **A *method* is a collection of data and statements for performing a task.** It executes when called from other parts of a program. It may take in parameters (or inputs) to vary its behaviour. The main() block is an example of a method that takes in a String array (argv) as its parameter. The main() method does not return any value to its caller.

Code 6.1 is revised to take into consideration the use of methods producing Code 6.2.

Code 6.2: Students' Examination Grades (Using Methods 1st Attempt)
```
class GradesB {

  static String[] students      = new String[20];
  static int[] chemMarks        = new int[20];
  static int[] mathsMarks       = new int[20];
  static int[] physicsMarks     = new int[20];
  static int[] averageMarks     = new int[20];
  static String[] chemGrades    = new String[20];
  static String[] mathsGrades   = new String[20];
  static String[] physicsGrades = new String[20];
  static String[] averageGrades = new String[20];
  static int      k             = 0;


  static void initArrays() {
    // initialize students array
    for (int j=0; j<20; j++) {
      students[j]      =  new String();
      chemGrades[j]    =  new String();
      mathsGrades[j]   =  new String();
      physicsGrades[j] =  new String();
      averageGrades[j] =  new String();
    }
  }


  static void readInputValues(int i, String argv[]) {
    // read input values
    students[i]     = argv[k];
    chemMarks[i]    = Integer.parseInt(argv[k+1]);
    mathsMarks[i]   = Integer.parseInt(argv[k+2]);
    physicsMarks[i] = Integer.parseInt(argv[k+3]);
    averageMarks[i] = (chemMarks[i] + mathsMarks[i] + physicsMarks[i])/3;
    k               = k + 4;
  }

  static void convertMarksToGrades(int i) {
    // convert marks to grades
    if (chemMarks[i] >= 80 && chemMarks[i] <= 100)
      chemGrades[i] = "A";
    if (chemMarks[i] >= 65 && chemMarks[i] <= 79)
      chemGrades[i] = "B";
    if (chemMarks[i] >= 50 && chemMarks[i] <= 64)
```

```
        chemGrades[i] = "C";
    if (chemMarks[i] >= 40 && chemMarks[i] <= 49)
      chemGrades[i] = "D";
    if (chemMarks[i] >= 0 && chemMarks[i] <= 39)
      chemGrades[i] = "F";
    if (mathsMarks[i] >= 80 && mathsMarks[i] <= 100)
      mathsGrades[i] = "A";
    if (mathsMarks[i] >= 65 && mathsMarks[i] <= 79)
      mathsGrades[i] = "B";
    if (mathsMarks[i] >= 50 && mathsMarks[i] <= 64)
      mathsGrades[i] = "C";
    if (mathsMarks[i] >= 40 && mathsMarks[i] <= 49)
      mathsGrades[i] = "D";
    if (mathsMarks[i] >= 0 && mathsMarks[i] <= 39)
      mathsGrades[i] = "F";
    if (physicsMarks[i] >= 80 && physicsMarks[i] <= 100)
      physicsGrades[i] = "A";
    if (physicsMarks[i] >= 65 && physicsMarks[i] <= 79)
      physicsGrades[i] = "B";
    if (physicsMarks[i] >= 50 && physicsMarks[i] <= 64)
      physicsGrades[i] = "C";
    if (physicsMarks[i] >= 40 && physicsMarks[i] <= 49)
      physicsGrades[i] = "D";
    if (physicsMarks[i] >= 0 && physicsMarks[i] <= 39)
      physicsGrades[i] = "F";
    if (averageMarks[i] >= 80 && averageMarks[i] <= 100)
      averageGrades[i] = "A";
    if (averageMarks[i] >= 65 && averageMarks[i] <= 79)
      averageGrades[i] = "B";
    if (averageMarks[i] >= 50 && averageMarks[i] <= 64)
      averageGrades[i] = "C";
    if (averageMarks[i] >= 40 && averageMarks[i] <= 49)
      averageGrades[i] = "D";
    if (averageMarks[i] >= 0 && averageMarks[i] <= 39)
      averageGrades[i] = "F";
  }

  static void printDetails(int i) {
    // print student's name, modules, marks and grades
    System.out.println(students[i]);
    System.out.println("\tChemistry \t" + chemMarks[i] +
                       "\t" + chemGrades[i]);
    System.out.println("\tMaths \t\t" + mathsMarks[i] +
                       "\t" + mathsGrades[i]);
    System.out.println("\tPhysics \t" + physicsMarks[i] +
                       "\t" + physicsGrades[i]);
    System.out.println("\tAverage \t" + averageMarks[i] +
                       "\t" + averageGrades[i]);
  }

  public static void main(String argv[]) {

    // produce students' examination grades
    initArrays();
    for (int i=0; i<(argv.length/4); i++) {
      readInputValues(i, argv);
      convertMarksToGrades(i);
      printDetails(i);
    }
  }
}
```

We begin our discussion of Code 6.2 with the main()method:

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

Contrast the main() in Code 6.2 with the main() in Code 6.1. The former is much shorter and clearer in its description of the processing logic than the latter. Details of the methods are now captured in separate sections of the program. Let us walk through the code with the use of three sets of marks as inputs:

```
D:\java>java GradesB "Mark Chua" 77 80 60 "Simon Goh" 45 90 23 "Kevin Lee" 66 92 56
```

The use of double quote for enclosing a student's name in the command prompt is to ensure that "Mark Chua" (for example) is treated as a single name rather than 2 separate names ("Mark" and "Chua").

Execution of the program begins with main(). The initArrays() method is first called to initialize the arrays for containing the values of each student. The for loop in main() iterates through all the available values in the argv array. This is indicated by argv.length/4. The numerator argv.length is the size of the array. Since there are 4 elements for each student, the number of items in the argv array is thus divided by 4. We check the value of argv.length to ensure that we do not process more than is provided in the argv array, otherwise, an exception (or error) will be reported by the JVM.

Given that there are only 3 sets of students' input data, the for loop will step through three times. How each student's examination grades are processed is described in the body of the for loop: read the input values (readInputValues()), convert the marks to grades (convertMarksToGrades()), and print the details (printDetails()).

6.3.1 Advantage of Using Methods

It is always good programming practice to provide meaningful and descriptive names for methods. From the name of the methods, readers will be able to fully comprehend how the program works without going into the details of the code. *The advantage of using methods therefore lies in its ability to contain code complexity by hiding details.*

6.3.2 Walking Through readInputValues() Method

Upon entry into the for loop, control is passed to the readInputValues() method. Two parameters are passed into the readInputValues() method – i and argv. The parameter i keeps count of the number of students processed and parameter argv provides a means for transferring the input String array into the readInputValues() method.

Execution of the readInputValues() method takes place one statement at a time with the various array elements updated with the values in the argv array. Note the use of the Integer.parseInt() method to convert a String value into an int value.

The variable k is incremented with the value 4 to move the index to the next student. Control is returned to main() after the last statement in readInputValues() method completes. convertMarksToGrades() is subsequently called from main().

6.3.3 Walking Through convertMarksToGrades() Method

At main(), the method convertMarksToGrades() is called and control is then transferred to convertMarksToGrades() with i as its input parameter. The method convertMarksToGrades() translates an entered mark into grade A, B, C, D, or F depending on the value of the mark. Control is returned to main() after the last statement in convertMarksToGrades() method completes. printDetails() is subsequently called from main().

6.3.4 Walking Through printDetails() Method

printDetails() prints the marks and grades for each of the module of a student. This process of reading (via readInputValues()), converting marks to grades (via convertMarksToGrades()) and printing the details (via printDetails()) is repeated for each student's set of marks. The program exits when all the entered data are processed.

6.4 Block Structure and Scope

You may have noticed that the body of all methods is defined as a block enclosed by braces ({…}). For example, the method initArrays() encloses a for loop that initializes the various array structures. A block allows us to declare data within a method; the data include constants, types and variables.

6.4.1 Local Variables

Any identifiers defined within a block are said to be *local* to the block i.e. the identifiers are only applicable within the block and are not recognized beyond the block. The area in which an identifier may be referenced is known as the *scope* of the identifier. For example, the index j as defined within the for loop of method initArrays() is local to initArrays(). To be exact, index j is local to the for loop and can only be referenced within the for loop. Beyond that, the identifier j does not exist and an error message is produced when j is referenced. For example, in Code 6.3, we have added an additional println() statement in initArrays()method. This println() statement references j beyond the scope with which j is defined. This form of referencing is not permissible in Java and the code therefore does not compile and an error message will be generated.

Code 6.3: Scope of Variables
```
static void initArrays() {
  // initialize students array
  for (int j=0; j<20; j++) {
    students[j]       =  new String();
    chemGrades[j]     =  new String();
    mathsGrades[j]    =  new String();
    physicsGrades[j]  =  new String();
    averageGrades[j]  =  new String();
  }
  System.out.println("Value of index j is " + j);
}
```

6.4.2 Global Variables

Let us consider another example (Code 6.4).

Code 6.4: Global Variables
```
class GlobalVar {
  static int i = 5;
  static int j = 8;

  public static void main(String argv[]) {
    for (int j=0; j<3; j++) {
      System.out.println("Value of i in for loop = " + i);
      System.out.println("Value of j in for loop = " + j);
    }
    System.out.println("Value of i out of for loop = " + i);
    System.out.println("Value of j out of for loop = " + j);
  }
}
```

There are two separate declarations of j. It is clear from the output of the code which j has been used to produce the values:

```
Value of i in for loop = 5
Value of j in for loop = 0
Value of i in for loop = 5
Value of j in for loop = 1
Value of i in for loop = 5
Value of j in for loop = 2
Value of i out of for loop = 5
Value of j out of for loop = 8
```

The scope of the j variable in the for loop is limited to the block of the for loop and only that j is used to produce the value in the println() statement within the for loop. However, when the execution exits the for loop, the j variable defined outside the for loop is used to print the j value (i.e. 8). The j variable defined within the for loop does not exist beyond the for loop and is said to be *out of scope*.

From the output of the code, it is clear that identifier i can be referenced within the for loop as well as without the for loop. This suggests that *the scope of an identifier covers both the block the identifier has been declared as well as any enclosing blocks*. **Identifiers declared outside a block are said to be *non-local*, or *global*, to the enclosed block.**

6.4.3 Determining Scope of Variables across Methods

How is the scope of variables determined in a multi-methods situation? We will use Code 6.5 to illustrate this situation.

Code 6.5: Local Variables in Methods
```
class LocalVariables {
  static int x = 5;
  static int y = 5;

  static void methodA() {
    int x = 1;
    System.out.println("x in methodA = " + x);
    System.out.println("y in methodA = " + y);
  }

  static void methodB() {
    int y = 1;
    System.out.println("x in methodB = " + x);
    System.out.println("y in methodB = " + y);
  }

  static void methodC() {
    System.out.println("x in methodC = " + x);
    System.out.println("y in methodC = " + y);
  }

  public static void main(String argv[]) {
    int x = 3;
    int y = 3;

    methodA();
    System.out.println("x in main = " + x);

    methodB();
    System.out.println("y in main = " + y);

    methodC();
    }
}
```

Declared within the methods are some local variables. Again, the code begins its execution from the method main() by calling methodA(). When control is returned to main(), an x value is printed followed by a call to methodB(). Subsequently, a y value is printed. Finally, methodC() is called. Let us examine the output from Code 6.5:

```
x in methodA = 1
y in methodA = 5
x in main = 3
x in methodB = 5
y in methodB = 1
y in main = 3
x in methodC = 5
y in methodC = 5
```

There are three declarations of variables x and y. When methodA() is executed, the variable x in methodA() is in scope and is thus printed. Within main(), the variable x in main() is printed since the local variable of methodA() is no longer in scope in main(). The same situation applies to the variable y in methodB() and main(). Finally, in methodC(), the variables x and y as defined in the program are printed since there is no x or y declared within methodC(). The variables x and y in the program are known as *global variables* and they apply to all methods, including main(). However, since methodA(), methodB() and main() have their own declarations of x and y, the local variables within these methods are printed instead.

6.4.4 Distinguishing Local Variables from Global Variables

Distinguishing local variables from global variables has the following advantages:

1. Since local variables are applicable within a method in which they have been declared, it is clear where the local variables are significant. In this way, it simplifies the reading of the program.
2. Inadvertent use of local variables by other parts of a program can be detected by the compiler – this ensures errors can be trapped and reported.
3. Local variables are created only on entry to the method and cease to exist before control returns to the calling sequence. Storage for local variables is only allocated when the method in which the local variables are declared is active. It can be de-allocated once control exits the method.

6.4.5 Scope of Identifier Declaration

From the above discussion, it is clear that identifier declaration is limited to a certain area of a program. The area in which an identifier declaration is applicable is known as the *scope of the identifier declaration*.

6.4.5.1 Scope Rules

The rules governing the scope of identifier declaration in Java are as follow:

1. The scope of an identifier declaration is the block (enclosed within two braces {...}) in which the declaration occurs, and all blocks enclosed by that block subject to rule 2 below.
2. When an identifier declared in a block A is re-declared in some block B enclosed by A, then block B and all blocks enclosed by it are excluded from the scope of the identifier's declaration in A.

We will illustrate the scoping rules using the example in Code 6.6. The output from Code 6.6 is:

```
x in methodA = 1
y in methodA = 5
x in main = 5
```

Block A encloses block B1 and B2. The variables x and y are applicable to block B1 and B2 (note the output from the code). Rule 1 applies here. Since block B2 has its own declaration of identifier x, the x that is used in block B2 is that declared in methodA(). This conforms to rule 2 above.

Code 6.6: Scoping Rules for Identifier Declaration

```
class IdentifierDeclaration {

  // block A
  static int x = 5;
  static int y = 5;

  static void methodA() {
    // block B2 (enclosed within block A)
    int x = 1;
    System.out.println("x in methodA = " + x);
    System.out.println("y in methodA = " + y);
  }

  public static void main(String argv[]) {
    // block B1 (enclosed within block A)
    methodA();
    System.out.println("x in main = " + x);
  }
}
```

6.4.5.2 Existence of Local Variables

Prior to the method entry, local variables do not exist and are undefined. They are created and initialized only when the method is invoked by other statements in the program. Local variables are destroyed and cease to exist just before control returns to the calling sequence. There is thus no relationship between the values of local variables created by successive invocation of the same method.

6.5 Parameters

In our discussion so far, we have mentioned the use of argv as a means for allowing inputs to be passed to a program from the command line prompt. argv is an example of a parameter. **A *parameter* is a variable of a certain type that allows values external to a method to be passed on to the method.** A parameter is also known as an *argument*.

6.5.1 Actual and Formal Parameters

Code 6.7 is a program that orders two numbers, a and b. If a is greater than b then the value of a and b are swapped; otherwise, nothing is done.

Code 6.7: Number Ordering 1

```
class NumberOrdering {

  // This program orders 2 numbers
  static int a = 5;
  static int b = 10;
  static int x = 0;
  static void order() {
    // swap values if a > b
    System.out.println("Before...");
    System.out.println("a is " + a);
    System.out.println("b is " + b);
    if (a>b) {
      System.out.println("a and b NOT in order. Swap a and b");
      x = a;
      a = b;
      b = x;
      System.out.println("After...");
      System.out.println("a is now " + a);
      System.out.println("b is now " + b);
    }
    else {
      System.out.println("a and b are in order");
      System.out.println("No further action");
    }
  }


  public static void main(String argv[]) {
    order();
  }
}
```

For the given values in Code 6.7, the following output is produced:

```
Before...
a is 5
b is 10
a and b are in order
No further action
```

Let us vary the values of a and b to 15 and 10 respectively. Code 6.8 records the new program with the change in values.

Code 6.8: Number Ordering 2

```
class NumberOrdering {

  // This program orders 2 numbers
  static int a = 15;
  static int b = 10;
  static int x = 0;

  static void order() {
    // swap values if a > b
    System.out.println("Before...");
    System.out.println("a is " + a);
    System.out.println("b is " + b);
    if (a>b) {
      System.out.println("a and b NOT in order. Swap a and b");
      x = a;
      a = b;
      b = x;
      System.out.println("After...");
      System.out.println("a is now " + a);
      System.out.println("b is now " + b);
    }
    else {
      System.out.println("a and b are in order");
      System.out.println("No further action");
    }
  }
  public static void main(String argv[]) {
    order();
  }
}
```

This revised program produces the following output:

```
Before...
a is 15
b is 10
a and b NOT in order. Swap a and b
After...
a is now 10
b is now 15
```

Note that the value of a and b are swapped. Although the above program works, it is inefficient since every change in the values of a and b requires a re-compilation of the code.

Consider Code 6.9. We begin our explanation from main() – the place where all Java programs start their execution. As mentioned earlier, Integer.parseInt() converts String inputs into int values. The value of variable m and n are passed into the method order() via parameters a and b as follows:

```
static void order(int a, int b) {
```

a and b are known as the *formal parameters* of order while m and n are *actual parameters* that convey values for the order() method. Actual parameters must match formal parameters i.e. the number and type of actual parameters must match that of the formal parameters. With Code 6.9, we can call:

```
D:\java>java Parameters 5 10
```

at the command line prompt, producing the following output:

```
Before...
a is 5
b is 10
a and b are in order
No further action
```

Subsequently, when we call

```
D:\java>java Parameters 15 10
```

the following output is produced:

```
Before...
a is 15
b is 10
a and b NOT in order. Swap a and b
After...
a is now 10
b is now 15
```

Code 6.9: Parameters

```
class Parameters {

  // This program orders 2 numbers
  static int x = 0;

  static void order(int a, int b) {
    // swap values if a > b
    System.out.println("Before...");
    System.out.println("a is " + a);
    System.out.println("b is " + b);
    if (a>b) {
      System.out.println("a and b NOT in order. Swap a and b");
      x = a;
      a = b;
      b = x;
      System.out.println("After...");
      System.out.println("a is now " + a);
      System.out.println("b is now " + b);
    }
    else {
      System.out.println("a and b are in order");
      System.out.println("No further action");
    }
  }


  public static void main(String argv[]) {
    int m = Integer.parseInt(argv[0]);
    int n = Integer.parseInt(argv[1]);
    order(m,n);
  }
}
```

Unlike Code 6.8, Code 6.9 does not require any re-coding or re-compilation even though the values of a and b are changed. **The use of parameters has allowed code to be reused. It has also enhanced the readability of program code.**

6.5.2 Value Parameters

Consider Code 6.10. The major difference between Code 6.9 and 6.10 is minor – the value of variables m and n are output before and after the order() method is executed.

Code 6.10: Value Parameters
```
class ValueParameters {

  // This program orders 2 numbers
  static int x = 0;

  static void order(int a, int b) {
    // swap values if a > b
    System.out.println("Before...");
    System.out.println("a is " + a);
    System.out.println("b is " + b);
    if (a>b) {
      System.out.println("a and b NOT in order. Swap a and b");
      x = a;
      a = b;
      b = x;
      System.out.println("After...");
      System.out.println("a is now " + a);
      System.out.println("b is now " + b);
    }
    else {
      System.out.println("a and b are in order");
      System.out.println("No further action");
    }
  }

  public static void main(String argv[]) {
    int m = Integer.parseInt(argv[0]);
    int n = Integer.parseInt(argv[1]);
    System.out.println("Value of m before order() = " + m);
    System.out.println("Value of n before order() = " + n);

    order(m,n);

    System.out.println("Value of m after order() = " + m);
    System.out.println("Value of n after order() = " + n);
  }
}
```

Executing Code 6.10 with the following values:

```
D:\java>java ValueParameters 15 10
```

produce the following output:

```
Value of m before order() = 15
Value of n before order() = 10
Before...
a is 15
b is 10
a and b NOT in order. Swap a and b
After...
a is now 10
b is now 15
Value of m after order() = 15
Value of n after order() = 10
```

Note that the values of m and n remain the same although they have been matched with a and b and the values of a and b were changed within the order() method. In other words, the values of actual parameters are not affected by any change in the corresponding value of formal parameters. Value parameters do not introduce any inadvertent side effects.

6.6 Methods that Return Values

You may have noticed the use of the term void in the methods you introduced in previous sections. The term void indicates that nothing is returned from a method. What if we want a method to return a value to the caller? Is it possible? If so, how do we do it in Java?

6.6.1 Returning Values

Let us revisit the Number Ordering problem first introduced in Section 6.5. The order() method does not return any value, so we do not know if a and b have been ordered successfully. To improve on this solution, we require order() to return a Boolean value to main(). If the value returned is true, main() can report “Values ordered!”, otherwise, it can report “Values NOT ordered!”. The order() method is thus changed from:

```
static void order(int a, int b)
```

to

```
static boolean order(int a, int b)
```

Code 6.9 is improved resulting in Code 6.11.

Code 6.11: Methods that Return Values

```
class MethodReturnValue {
  // This program demonstrates function
  static int x = 0;

  static boolean order(int a, int b) {
    // swap values if a > b
    System.out.println("Before...");
    System.out.println("a is " + a);
    System.out.println("b is " + b);
    if (a>b) {
      System.out.println("a and b NOT in order. Swap a and b");
      x = a;
      a = b;
      b = x;
      System.out.println("After...");
      System.out.println("a is now " + a);
      System.out.println("b is now " + b);
      return true;
    }
    else {
      System.out.println("a and b are in order");
      System.out.println("No further action");
      return false;
    }
  }


  public static void main(String argv[]) {
    int m = Integer.parseInt(argv[0]);
    int n = Integer.parseInt(argv[1]);

    boolean ordered = order(m,n);

    if (ordered)
      System.out.println("Values ordered!");
    else
      System.out.println("Values NOT ordered!");
  }
}
```

We add two return statements into order(). In main(), a new Boolean variable ordered has been defined. A test on the ordered variable determines which of the two statements in 'if' will be executed. The value of ordered is derived from the order() method and is *returned* from the evaluation of order(). Executing Code 6.11 using:

```
D:\java>java MethodReturnValue 15 10
```

yields

```
Before...
a is 15
b is 10
a and b NOT in order. Swap a and b
After...
a is now 10
b is now 15
Values ordered!
```

and executing it with

```
D:\java>java MethodReturnValue 5 10
```

yields

```
Before...
a is 5
b is 10
a and b are in order
No further action
Values NOT ordered!
```

From the outputs, it is clear that both solutions (one using method and the other using function) are similar.

6.6.2 The return Statement

A return statement terminates the execution of a method and returns flow control to the invoking method. A simple return statement such as return; returns no value. This form of return is commonly used to exit a method.

If a method has a return type, the return statement must include an expression of a type that could be assigned to the return type. For example, if a method returns double, a return could have an expression that is a double, float or int.

CHAPTER 7: CLASS AND OBJECTS

Yes! Java is object-oriented. You have noticed that all Java programs begin with the word 'class' and this is suggestive of object-oriented programming.

All data variables are based on a type that defines their structure. For example, a data variable number1 may be defined with an int type and initialized with a value 10 as follows:

```
int number1 = 10;
```

Similarly, we may define another data variable number2 with a different value, say 88 as follows:

```
int number2 = 88;
```

Both number1 and number2 are allocated their own memory space to store their values. They are of the same type int but their values differ. Figure 7.1 illustrates this.

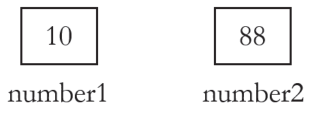


FIGURE 7.1: Data Variables and Type

We can create as many data variables of the same type as we like, each data variable taking on its own memory space. So, we can create another data variable, number3, of int type, for example.

A *type* therefore defines the form or structure of data variables. Data variables from the same type have the same characteristics. The only difference among them is their value. We say that data variables are *instantiations* of type.

7.1 Class and Objects

The concept of class is similar to type. A class defines the structure of objects, just as a type defines the structure of a data variable. Let us consider an example of class and objects. The following code fragment defines a String object named title and has the value "JAVA Programming":

```
String title = new String("JAVA Programming");
```

The class that defines this object is String. Let us create another string object:

```
String author = new String("Danny Poo");
```

author is an object from the String class and it has the value "Danny Poo".

We use the new keyword to create an object. What new does is to allocate memory space for the object and associate the name (such as title) to the created object. Figure 7.2 illustrates the creation of the two objects of String type referenced by title and author. Notice the similarity between type and class. Both of them are definitional and specify the structure of something. Data variables and objects can be instantiated from type and class respectively.

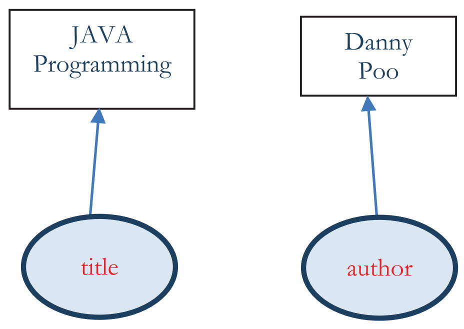


FIGURE 7.2: Class and Objects

7.2 Objects in Java

Look around you and you see objects. Tables, chairs, buses, trees, and telephones are all examples of real-world objects. Every object in the real world can be modeled as a programming *object* in Java. Even abstract things like schedules or diary entries can be modeled as objects.

Let us consider the example of a bus. To model a bus using the Object-Oriented Programming approach, we need to consider what aspects of a bus are of interest in the solution. Are we interested in the colour of the bus? The driver's name? The number of passengers in the bus? Or even the capacity of the bus? In Code 7.1, we have limited our consideration of a bus to two properties: colour and number of passengers.

Code 7.1: The Bus Class

```
class Bus {
  String colour;
  int numPassengers = 0;
}
```

To be able to instantiate objects, we need to define a class. In Code 7.1, the Bus class has been defined with two data variables colour and numPassengers. Data variables in classes are also known as *attributes* (Note: Attributes, properties and fields are used inter-changeably to mean the same thing).

Attributes define the *static properties* of a class of objects. A class is basically a mould or template for describing objects i.e. all objects from a class are similar in properties and they behave in the same way as any other objects of the same class. Objects are instances of a class, just as data variables are instances of a type.

7.2.1 Constructors

Objects must be created (or instantiated) so that their behaviour can be manifested. To facilitate the creation of objects, a class has to define a special method called the *constructor method* of the class.

A constructor method provides the means for creating objects. It includes statements for initializing attribute values. Code 7.2 extends Code 7.1 with the addition of a constructor method. *The name of a constructor method is similar to the name of the class it represents.*

In Code 7.2, we note that the constructor method for the class Bus is the method Bus(). In executing a constructor method, Java allocates memory and creates an instance of Bus before initializing the values of the object's attributes. In this case, the colour attribute is initialized to "Red". Note that a constructor method does not have a return type (not even the ubiquitous "void").

Code 7.2: The Constructor Method
```
class Bus {
  String colour;
  int numPassengers = 0;

  // Default constructor
  Bus() {
    colour = "Red";
  }
}
```

While a constructor method enables an object to be created, it must be explicitly called. To call a constructor method, we use the new keyword. To illustrate, we will add an additional user class (ObjectCreation) to Code 7.2. The modified code is now given in Code 7.3.

Code 7.3: Instantiating a Bus Object
```
class ObjectCreation {

  // This program demonstrates object creation

  public static void main(String argv[]) {
    Bus b = new Bus();
    System.out.println("Color of bus is " + b.color);
    System.out.println("There are " + b.numPassengers + " passengers");
  }
}


class Bus {
  String color;
  int numPassengers = 0;

  // Default constructor.
  Bus() {
    color = "Red";
  }
}
```

As usual, execution begins with main(). The first statement in main():

```
Bus b = new Bus();
```

defines a variable b of class Bus. The new keyword creates an object of class Bus. What new does is to call the constructor method to allocate memory for representing the newly created object before initializing the attributes of the object of class Bus. Reference to the object is assigned to the variable b. Henceforth, any processing on the Bus object will be done via the variable b. The next two statements:

```
System.out.println("Colour of bus is " + b.colour);
System.out.println("There are " + b.numPassengers + " passengers");
```

produce the following output:

```
Colour of bus is Red
There are 0 passengers
```

Note the use of the *dot notation* in b.colour. The dot is used to refer to an attribute or method of an object. In this case, the colour attribute of the object b is referenced. Similarly, the numPassengers of object b is referenced and produced in the println() statement.

7.2.2 Multiple Constructor Method Definition

Java allows for multiple constructor method definition, that is, a class can have more than one constructor methods defined. For example, in Code 7.4, there are three constructor methods for the class Bus. Which of these constructors is used depends on what is called by the user class.

Code 7.4: Multiple Constructors
```
class MultipleConstructors {

  // This program demonstrates multiple constructors

  public static void main(String argv[]) {
    Bus b = new Bus();
    System.out.println("Color of bus is " + b.color);
    System.out.println("There are " + b.numPassengers + " passengers");
    Bus b1 = new Bus("Blue");
    System.out.println("Color of bus is " + b1.color);
    System.out.println("There are " + b1.numPassengers + " passengers");
    Bus b2 = new Bus("White", 10);
    System.out.println("Color of bus is " + b2.color);
    System.out.println("There are " + b2.numPassengers + " passengers");
  }
}

class Bus {
  String color;
  int numPassengers = 0;

  // Default constructor.
  Bus() {
    color = "Red";
  }

  // Constructor 1.
  Bus(String c) {
    color = c;
  }

  // Constructor 2.
  Bus(String color, int numPassengers) {
    this.color = color;
    this.numPassengers = numPassengers;
  }
}
```

In Code 7.4, there are three instances of Bus object created – b, b1, and b2. For object b, the default constructor is called, for object b1, constructor 1 is called and finally for object b2, constructor 2 is called. Multiple constructor method definition therefore allows us to initialize object attributes with different values upon creation time.

Java is able to know which constructor has been called by matching the type and number of parameters with those defined for the constructor methods. If the type and number of parameters provided by the calling method match a constructor method's signature then that constructor is invoked. If none of the available constructor methods' signature matches the calling method's provided parameters then an error is reported by Java.

All constructors share the same method name and they are distinguished by the types and number of parameters. Constructor methods are therefore *overloaded* in definition.

In Code 7.4, the two constructor methods are overloaded:

1. Bus(String c) allows us to specify the initial colour of the Bus object. An example of how this constructor is used is:
   ```
   Bus b1 = new Bus("Blue");
   ```
   This creates a blue Bus object with zero passengers on board.
2. Bus(String colour, int numPassengers) allows us to specify both the initial colour of the bus and the number of passengers in the bus. Notice the formal parameter and object attribute define two variables of the same name: colour. To resolve which variable is referenced, Java provides the this keyword to distinguish an object's attribute from the formal parameter definition. We use this.colour to refer to the object attribute and colour to refer to the formal parameter colour. An example of how this constructor is used is:
   ```
   Bus b2 = new Bus("White", 10);
   ```
   This creates a white Bus object with 10 passengers.

Method overloading increases code flexibility. For this example, a Bus object needs not always be "Red" and have no passenger when it is created. Colour may be changed, so can the number of passengers when the object is first created. The above code illustrates this flexibility as evident in the output of the code:

```
Colour of bus is Red
There are 0 passengers
Colour of bus is Blue
There are 0 passengers
Colour of bus is White
There are 10 passengers
```

7.2.3 Constructor Method Invocation

A constructor method can also be used to invoke another constructor method to assign values to object attributes. This is done using the this keyword which refers to the current referenced object. When you see this("Red"), think of it as invoking the constructor Bus("Red"). Code 7.5 illustrates.

Code 7.5: Constructors Using this Keyword
```
class Bus {
  String colour;
  int numPassengers = 0;

  // Default constructor.
  Bus() {
    this("Red");
  }

  // Overloaded constructor 1.
  Bus(String c) {
    this(c, 0);
  }

  // Overloaded constructor 2.
  Bus(String colour, int numPassengers) {
    this.colour = colour;
    this.numPassengers = numPassengers;
  }
}
```

When the statement Bus b = new Bus(); is called, the default constructor is executed. This default constructor method in turn invokes Bus("Red"). The overloaded constructor 1 invokes Bus("Red", 0) that finally creates a red bus with zero passengers.

7.2.4 Instance and Class Variables

The variables colour and numPassengers of a Bus object are known as *instance variables* or *non-static variables*. Each occurrence of these two variables belongs to one and only one Bus object. Should there be three different Bus objects created, there will be three copies of colour and numPassengers variables, each copy belonging to one Bus object. For example, one green bus may contain 0 passenger, another green bus may contain 5 passengers, and a third red bus may contain 20 passengers.

Besides instance variables, there are also *class variables* or *static variables*. Class variables are prefixed with the keyword static as shown in Code 7.6.

Code 7.6: Class or Static Variable
```
class Bus {
  String colour;
  int numPassengers = 0;
  static int numBuses;
}
```

In Code 7.6, the variable numBuses is defined as a *class variable* using the static keyword. Think of numBuses as being a common variable that is *shared by all the Bus objects*. This variable can be used to keep a count of the number of Bus objects created. Class variables are also referenced using the dot notation.

7.2.5 Instance and Class Methods

Class and objects are distinct in concept. They have their own definition of data types and methods. By this, we mean that there are instance (or object) methods and class methods, just as there are instance variables and class variables.

7.2.5.1 Instance Methods

Methods that belong to objects are known as *instance methods* or *non-static methods.* Let us consider how instance methods are defined and used.

We will add two methods to the class Bus: board() and disembark(). The method board() increases and the method disembark() decreases the number of passengers on a bus respectively. Consider the code in Code 7.7. The method board() accepts an int parameter n which specifies the number of passengers boarding the bus. The instance variable numPassengers is increased by this number n. We invoke an instance method, board(), on a Bus object b as follows: b.board(7). This method increases the number of passengers on the Bus object b by 7.

Code 7.7: Instance Methods
```
class Bus {
  String colour;
  int numPassengers = 0;

  // n passengers board the bus.
  public void board(int n) {
    numPassengers += n;
  }

  // n passengers leave the bus.
  public void disembark(int n) {
    numPassengers -= n;
  }
}
```

The method disembark() is similar to the method board() except that it decreases the number of passengers on the bus.

As this example shows, instance methods allow us to change the *instance variables* (object attributes) of an object.

7.2.5.2 Class Methods

Class methods or *static methods* allow us to change only the *class variables* of an object. Suppose we want a method to increase the number of buses in the class variable numBuses (see Code 7.8), we define a class method incNumBuses(). A class method is specified with the static keyword.

Code 7.8: Class Method
```
class Bus {
  String colour;
  int numPassengers = 0;
  static int numBuses;

  public static void incNumBuses() {
    numBuses++;
  }
}
```

There are two ways of referencing class variables:
1. Using the class name Bus or
2. Using the object's variable name b

Both Bus.numBuses and b.numBuses are acceptable, but we prefer the former notation for clarity. Code 7.9 is a complete example that illustrates this point.

Code 7.9: Referencing Static Variables and Methods
```
class StaticVariablesMethods {

  // This program demonstrates referencing static variables and methods

  public static void main(String argv[]) {
    Bus b = new Bus("Blue ", 12);
    Bus.numBuses++;
    Bus b1 = new Bus("Red  ", 16);
    b1.numBuses++;
    Bus b2 = new Bus("White", 10);
    b2.incNumBuses();
    Bus b3 = new Bus("Orange", 19);
    b2.incNumBuses();
    Bus b4 = new Bus("Green", 7);
    Bus.incNumBuses();
    System.out.println("Using object b.  " + " Number of buses = " + b.numBuses);
    System.out.println("Using object b1. " + " Number of buses = " + b1.numBuses);
    System.out.println("Using object b2. " + " Number of buses = " + b2.numBuses);
    System.out.println("Using object b3. " + " Number of buses = " + b3.numBuses);
    System.out.println("Using object b4. " + " Number of buses = " + b4.numBuses);
    System.out.println("Using class name." + " Number of buses = " + Bus.numBuses);
  }
}

class Bus {
  String color;
  int numPassengers = 0;
  static int numBuses;
  // Constructor
  Bus(String color, int numPassengers) {
    this.color = color;
    this.numPassengers = numPassengers;
  }

  // n passengers board the bus.
  public void board(int n) {
    numPassengers += n;
  }

  // n passengers leave the bus.
  public void disembark(int n) {
    numPassengers -= n;
  }

  public static void incNumBuses() {
    numBuses++;
  }
}
```

Five Bus objects referenced by b, b1, b2, b3, and b4 are created. The static variable numBuses is incremented each time a Bus object is created. Increment on numBuses is done in two ways:

1. Via the class name Bus
   ```
   Bus.numBuses++;
   Bus.incNumBuses();
   ```
2. Via the object reference
   ```
   b1.numBuses++;
   b2.incNumBuses();
   ```

Note that it is also possible to call a static method via an object reference e.g.

```
b2.incNumBuses();
```

Code 7.9 produces the following outputs:

```
Using object b.   Number of buses = 5
Using object b1.  Number of buses = 5
Using object b2.  Number of buses = 5
Using object b3.  Number of buses = 5
Using object b4.  Number of buses = 5
Using class name. Number of buses = 5
```

7.3 Class Hierarchy

So far we have been looking at classes independently. Classes can also be related to one another hierarchically. For example, the class Bus we introduced earlier can also be considered as a Vehicle. Similarly, class Car, Truck, and Motorcycle are Vehicles too.

Figure 7.3 shows the hierarchical relationship between Vehicle and Bus, Car, Truck, and Motorcycle.

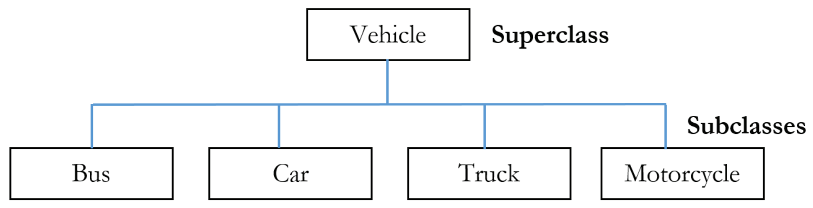


FIGURE 7.3: Class Hierarchy for Vehicle

7.3.1 Superclass and Subclass

Hierarchically, Vehicle is drawn higher than the other classes and is said to be more general than those classes lower in the hierarchy. A more general class (such as Vehicle) is known as a *superclass* of a class that is more specific (e.g. Bus, Car, Truck, or Motorcycle).

A superclass, being more general, has properties that are common to classes that are more specific. For example, a Vehicle class has properties like "colour" and "number of passengers" that are common to the more specific classes Bus, Car, Truck, and Motorcycle.

Class that are more specific are known as *subclasses* (of Vehicle in this case). Subclasses, being more specific, would have some properties that are *not* common to the other classes in the hierarchy. The uncommon properties distinguish a subclass from the other subclasses. For example, a truck may be fitted with a tailgate that can be lowered or raised. Hence, a truck may have methods such as lowerTailGate() and raiseTailGate(); it may also have a Boolean variable that indicates if the tailgate is raised. These methods and variable are not common in the other classes of vehicles and are thus not defined in the other classes.

The hierarchical relationship of superclasses and subclasses are represented in Java using the extends keyword. Code 7.10 is a Java representation of the Vehicle class hierarchy.

Code 7.10: Class Hierarchy for Vehicle
```
class SuperSubclass {

  // This program illustrates Superclass and Subclass

  public static void main(String argv[]) {
    Bus b = new Bus("White", 10);
    System.out.println("Color of bus is " + b.color);
    System.out.println("There are " + b.numPassengers + " passengers");
    Car c = new Car("Gold", 4);
    System.out.println("Color of car is " + c.color);
    System.out.println("There are " + c.numPassengers + " passengers");
    Truck t = new Truck("Black", 1);
    System.out.println("Color of truck is " + t.color);
    System.out.println("There are " + t.numPassengers + " passengers");
    Motorcycle m = new Motorcycle("Red", 1);
    System.out.println("Color of motorcycle is " + m.color);
    System.out.println("There are " + m.numPassengers + " passengers");
  }
}

class Vehicle {

  // Constructor
  Vehicle() {}
}

class Bus extends Vehicle{
  String color;
  int numPassengers = 0;

  // Constructor
  Bus(String color, int numPassengers) {
    this.color = color;
    this.numPassengers = numPassengers;
  }
}

class Car extends Vehicle{
  String color;
  int numPassengers = 0;

  // Constructor
  Car(String color, int numPassengers) {
    this.color = color;
    this.numPassengers = numPassengers;
  }
}

class Truck extends Vehicle{
  String color;
  int numPassengers = 0;

  // Constructor
  Truck(String color, int numPassengers) {
    this.color = color;
    this.numPassengers = numPassengers;
  }
}

class Motorcycle extends Vehicle{
  String color;
  int numPassengers = 0;

  // Constructor
  Motorcycle(String color, int numPassengers) {
    this.color = color;
    this.numPassengers = numPassengers;
  }
}
```

The output from Code 7.10 is given below:

```
Colour of bus is White
There are 10 passengers
Colour of car is Gold
There are 4 passengers
Colour of truck is Black
There are 1 passengers
Colour of motorcycle is Red
There are 1 passengers
```

Note that the attributes colour and numPassengers appear in the definition of each subclass of Vehicle.

7.3.2 Inheritance

The attributes colour and numPassengers are common in all subclasses in Code 7.10 and have been repeatedly defined in each of the subclasses of Vehicle. Instead of repeating the definition of common properties in subclasses, Java allows for them to be defined in the superclass once and *implicitly* propagate their definition to the lower subclasses. This is shown in Code 7.11.

Note that properties colour and numPassengers are now defined only in the class Vehicle and not in the subclasses. The net outcome of Code 7.11 is similar to Code 7.10.

By taking on the definition of the properties in the superclass, the subclasses are said to *inherit* those properties from their superclass(es). This mechanism of propagating property definition from superclass to subclasses is known as *inheritance* in Object-Oriented Programming.

Code 7.11: Inheritance
```
class Inheritance {

  // This program illustrates Inheritance

  public static void main(String argv[]) {
    Bus b = new Bus("White", 10);
    System.out.println("Color of bus is " + b.color);
    System.out.println("There are " + b.numPassengers + " passengers");
    Car c = new Car("Gold", 4);
    System.out.println("Color of car is " + c.color);
    System.out.println("There are " + c.numPassengers + " passengers");
    Truck t = new Truck("Black", 1);
    System.out.println("Color of truck is " + t.color);
    System.out.println("There are " + t.numPassengers + " passengers");
    Motorcycle m = new Motorcycle("Red", 1);
    System.out.println("Color of motorcycle is " + m.color);
    System.out.println("There are " + m.numPassengers + " passengers");
  }
}


class Vehicle {
  String color;
  int numPassengers = 0;

  // Constructor
  Vehicle() {}
}

class Bus extends Vehicle{

  // Constructor
  Bus(String color, int numPassengers) {
    this.color = color;
    this.numPassengers = numPassengers;
  }
}

class Car extends Vehicle{

  // Constructor
  Car(String color, int numPassengers) {
    this.color = color;
    this.numPassengers = numPassengers;
  }
}

class Truck extends Vehicle{

  // Constructor
  Truck(String color, int numPassengers) {
    this.color = color;
    this.numPassengers = numPassengers;
  }
}

class Motorcycle extends Vehicle{
  // Constructor
  Motorcycle(String color, int numPassengers) {
    this.color = color;
    this.numPassengers = numPassengers;
  }
}
```

CHAPTER 8: INPUTS AND OUTPUTS

So far, we have been using System.out.print() and System.out.println() to display texts on screens but we did not mention why we need to use such complicated syntax to output texts. We have also not mentioned anything on the use of Graphical User Interface components for outputs. (Note: Graphical User Interface (GUI) components in Java is beyond the scope of this book.)

In this chapter, we will discuss keyboard inputs and screen outputs. We will also discuss how to handle exceptions when they occur during the input and output process.

In addition, we will examine a class called the Scanner class which has been included in the Java API beginning Java 5. This class is a simple text scanner that can parse primitive types and strings using regular expressions.

8.1 Input and Output Streams

Inputs and outputs in Java function in terms of streams. Think of how a stream of water flows. It flows in a continuous manner and always in a single direction.

8.1.1 Screen Outputs

The way screen outputs function is similar to the way water flows in a stream. When a text is printed via the System.out.println() statement, the output is channeled in a uni-directional manner to the screen for display. The stream with which a screen output statement is associated with is known as an *output stream* and it is represented by the variable System.out. The screen output stream is the default output stream and is automatically created by Java. You as a Java programmer simply use it.

8.1.2 Keyboard Inputs

Keyboard input stream is treated differently from screen output stream. The keyboard input stream has to be explicitly created before it can be used. More specifically, you will need to create a *keyboard input stream object*.

To simplify programming, Java provides a set of APIs for entering texts via the keyboard. These APIs are included in the java.io package. Of great interest in text inputs and outputs are the following classes:

1. BufferedReader
2. BufferedWriter
3. InputStreamReader
4. PrintWriter

To create a keyboard input stream object, we write:

```
BufferedReader br = new BufferedReader(
                        new InputStreamReader(System.in)
                    );
```

Let us unpack this statement. System.in is a System class variable that refers to the standard input stream object created automatically by Java. System.in typically refers to the keyboard and data are read from the keyboard in terms of bytes. Inputs in terms of bytes are not easily comprehensible. To convert bytes into characters, an InputStreamReader object is created to serve as a bridge from byte streams to character streams. This object reads bytes and decodes them into characters.

In the statement, a BufferedReader object is created based on the InputStreamReader object. This is to facilitate more efficient reading of data. Once we have a BufferedReader object, it is used to read in texts from the keyboard.

Since all input and output classes reside in java.io package, they must be imported. We include import java.io.*; at the beginning of each source-code file to facilitate the import of the required input and output classes.

8.1.3 Reading and Displaying Texts

Screen output and keyboard input are usually done together to provide users with a user-friendly interface.

8.1.3.1 Text Input

Information from the keyboard is usually read in as strings using the readLine() instance method in the BufferedReader class. An example of the use of the readLine() instance method is shown in Code 8.1.

Code 8.1: Reading from a Keyboard
```
BufferedReader br = new BufferedReader(
                       new InputStreamReader(System.in)
                     );
String name;
System.out.print("Please enter your name: ");
name = br.readLine();
System.out.println("Your name is: " + name);
```

Code 8.1 displays a user-friendly prompt, waits for the user to type in his name, and displays his name on the screen. The name is entered from the keyboard and only when the user presses the "return" key. The readLine() statement reads in whatever is entered up to the "return" key at the keyboard. If you intend to read in character by character, use the read() instance method instead.

8.1.3.2 Numeric Input

Java provides a mechanism for converting a String object into an int or double primitive type. Integer.parseInt() converts a String object into an int value and Double.parseDouble() converts a String object into a double value. Both methods are class methods (of the Integer and Double class respectively). An example of the use of the parseInt() class method is shown in Code 8.2.

Code 8.2: Converting String input into int Type
```
BufferedReader br = new BufferedReader(
                      new InputStreamReader(System.in)
                     );
int age;
System.out.print("Please enter your age: ");
age = Integer.parseInt(br.readLine());
System.out.println("Your age is: " + age);
```

Integer.parseInt() converts the String object returned by br.readLine() into its equivalent int primitive type.

8.2 Exception Handling

Errors may happen when br.readLine() is called. For example, a technical problem occurs at the input device. Errors are known as *exceptions*. Input errors must be dealt with and there are two ways of handling them:

1. Programmer provides code to deal with the errors explicitly
2. Let Java handles the errors automatically

For the example below, we will let Java handles the errors automatically. We will discuss the first approach in Section 8.4.

8.2.1 Java Exception Handling

To let Java handles the errors, a method has to throw the exception. In Code 8.3, the exception (IOException) is thrown as indicated by the statement "throws IOException" at the end of the method header. Note that input and output classes are imported via the import statement.

Code 8.3: Numeric Input with Exception Handling by Java

```
import java.io.*;

class NumericInput {

  public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    int age;
    System.out.print("Please enter your age: ");
    age = Integer.parseInt(br.readLine());
    System.out.println("Your age is: " + age);
  }
}
```

A sample of output from Code 8.3 might be:

```
Please enter your age: 25
Your age is: 25
```

Entering an invalid age such as "twenty-five" will result in an exception being reported by Java:

```
Please enter your age: twenty-five
Exception in thread "main" java.lang.NumberFormatException: For input string: "twenty-five"
        at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
        at java.lang.Integer.parseInt(Integer.java:468)
        at java.lang.Integer.parseInt(Integer.java:518)
        at Chap8NumericInput.main(Chap8NumericInput.java:9)
```

8.2.2 Explicit Exception Handling

By now, you should be familiar with how to execute a Java program at the command prompt:

```
D:\java>java <className>
```

where <className> is the name of the Java class to execute.

To execute a Java program with user supplied arguments, however, we need to enter at the command prompt some arguments:

```
D:\java>java <className> arg0 arg1 arg2 …
```

where arg0, arg1, arg2 are the arguments.

The entered command line arguments are mapped to the String array elements of the main() method:

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

The formal parameter args store the command line arguments. Suppose the following is entered at the command line:

```
D:\java>java <className> arg0 arg1 arg2
```

then args[0] will contain the String "arg0", args[1] will contain the String "arg1", and args[2] will contain the String "arg2". args.length (the variable indicating the length of the array) will contain the value 3 since there are exactly three command line arguments entered. Let us examine an example as given in Code 8.4.

Code 8.4: Command Line Arguments

```
import java.io.*;

class CommandLineArguments {

  public static void main(String[] args) {
    String s = args[0];
    int i    = Integer.parseInt(args[1]);
    double d = Double.parseDouble(args[2]);

    System.out.println("s is: " + s);
    System.out.println("i is: " + i);
    System.out.println("d is: " + d);
  }
}
```

Code 8.4 expects three command line arguments; the first must be a String, the second must be an integer, and the third must be a floating point number. Since the command line arguments are all stored as strings in the String array, they must be converted to the equivalent int and double type where required. So, the first argument stored in args[0] can simply be assigned to the String s; the second argument stored in args[1] must be converted to an int first using Integer.parseInt() before it can be stored in the int variable i; and the third argument args[2] must be converted to a floating point number first using Double.parseDouble() before it can be stored in the double variable d. Let us execute this program with the following arguments:

```
D:\java>java CommandLineArguments Hello 123 4.56
```

No error is reported and the following output is produced:

```
s is: Hello
i is: 123
d is: 4.56
```

Try entering values in the command line that do not conform to the required data type and observe the exceptions reported by Java.

Alternatively, we can provide code to deal with the errors explicitly. Let us consider the example in Code 8.5.

Code 8.5: Numeric Input with Explicit Exception Handling

```
import java.io.*;

public class NumericInput2 {

  public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(
    new InputStreamReader(System.in));
    String input;
    int i = 0;
    boolean ok;

    do {
      ok = true;
      System.out.print("Please enter your age: " );
      input = br.readLine();
      try {
        i = Integer.parseInt(input);
      } catch (NumberFormatException e) {
          System.out.println(e);
          System.out.println("---");
          e.printStackTrace();
          ok = false;
          System.out.println("Invalid input: " + input);
          System.out.println("Please try again.");
      }
    } while (!ok);
    System.out.println("Your age is: " + i);
  }
}
```

In Code 8.5, any NumberFormatException that may occur when invoking Integer.parseInt() (Line 15) is now managed by the use of the

```
try {
  try_block
} catch ( exception_to_catch ) {
    catch_block
}
```

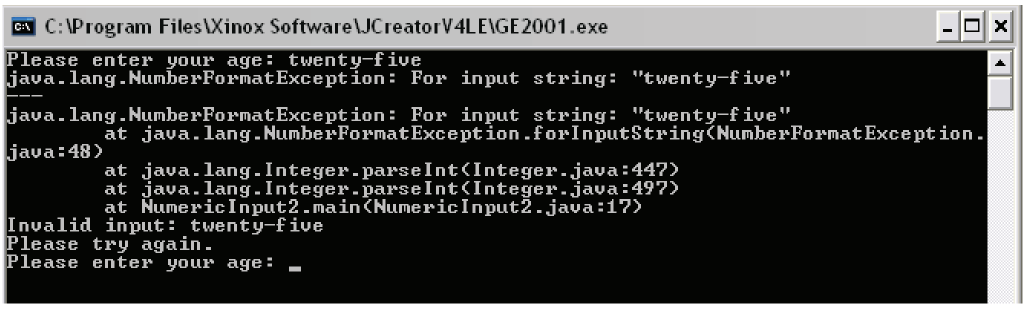
structure. The words “try” and “catch” are Java reserved keywords.

Within the block marked try_block, we try to catch the exception specified within the parenthesis and denoted by exception_to_catch. If such an exception occurs, the block marked catch_block is executed.

For the above example, a NumberFormatException exception occurs when non-numeric characters are entered. When such an exception happens, the statement

```
i = Integer.parseInt(input);
```

throws the NumberFormatException exception and execution is diverted to the catch_block. Statements within the catch_block will be executed. What these statements do is to ask the user to re-enter his age again (see Figure 8.1).


C:\Program Files\Xinox Software\JCreatorV4LE\GE2001.exe

```
Please enter your age: twenty-five
java.lang.NumberFormatException: For input string: "twenty-five"
---
java.lang.NumberFormatException: For input string: "twenty-five"
        at java.lang.NumberFormatException.forInputString(NumberFormatException.
java:48)
        at java.lang.Integer.parseInt(Integer.java:447)
        at java.lang.Integer.parseInt(Integer.java:497)
        at NumericInput2.main(NumericInput2.java:17)
Invalid input: twenty-five
Please try again.
Please enter your age: _
```

FIGURE 8.1: Handling Exception

This approach is certainly more user-friendly than the previous approach of abruptly ending the program with an unfriendly error message. Once an appropriate age value is entered, the program displays the age and terminates gracefully.

8.3 The Scanner Class

You were earlier introduced to the standard input and standard output via System.in and System.out. You can use System.in to read in inputs into your application but you can only do so one byte at a time. Since integers or strings are structured using multiple bytes (e.g. four bytes are required for an int value), it would be cumbersome to use the read() method in System.in for this purpose.

An alternate approach is to use the BufferedReader class to read in texts and with the parseInt() method in the Integer class, we can transform string values captured via BufferedReader into integer values.

Beginning Java 5, the Java API includes a class that allows us to read primitive data type values (such as int, float, double, byte, long, and short) and strings directly into our applications. With this class, much of the problems associated with reading in multi-byte values are removed. The new class is known as the Scanner class and it is located within the java.util package.

8.3.1 Creating a Scanner Object

To use the facilities in the Scanner class, we need to create a Scanner object. The most common way of creating a Scanner object is to associate a Scanner class with System.in:

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

8.3.2 Handling Numerical Data Types

The Scanner class has six methods to input numerical data types: int, float, double, byte, long, and short. Table 8.1 shows how these methods are used.

TABLE 8.1: Six Methods to Input Numerical Data Types

Method	Usage
nextInt()	`int    i = scanner.nextInt();`
nextFloat()	`float  f = scanner.nextFloat();`
nextDouble()	`double d = scanner.nextDouble();`
nextByte()	`byte   b = scanner.nextByte();`
nextLong()	`long   l = scanner.nextLong();`
nextShort()	`short  s = scanner.nextShort();`

8.3.3 Handling String Values

To input a string value, use the next() method, for example:

```
String s = scanner.next();
```

8.3.4 Handling Boolean Values

To input a boolean value, use the nextBoolean() method, for example:

```
boolean b = scanner.nextBoolean();
```

8.3.5 Exceptions and Delimiters

In Code 8.6, we show you how to apply the Scanner class. To use the Scanner class, we need to import the java.util package or java.util.Scanner. A Scanner object associated with System.in is instantiated. The user is prompted to enter 3 numbers. The numbers entered are printed. The following is a possible interaction on the command prompt:

```
Enter 3 numbers: 1 2 3
1
2
3
```

In the above interaction, the three numbers entered have been separated by a space which is default delimiter used for distinguishing input tokens. When the tokens are entered, they are placed in the input buffer. Each call on the nextInt() method gets a token from the input buffer. No exception will be reported so long as the token is a number (or integer in this case).

Code 8.6: Scanner1
```
import java.util.Scanner;

public class Scanner1 {

  public static void main(String[] args) {
    boolean done = false;

    while (!done) {
      try {
        Scanner scanner = new Scanner(System.in);
        System.out.print("\nEnter 3 numbers: ");
        for (int i=0; i<3; i++)
          System.out.println(scanner.nextInt());
        done = true;
      } catch (Exception e) {
        System.out.println("Exception: " + e.toString());
      }
    }
  }
}
```

Consider a situation when a floating-point number is entered instead. An InputMismatchException will be thrown:

```
Enter 3 numbers: 1 2.3 4
1
Exception: java.util.InputMismatchException
```

In the above interaction, three tokens “1”, “2.3” and “4” are kept in the input buffer. When the second token “2.3” is read, nextInt() throws an InputMismatchException exception.

We can override the default delimiter by using the useDelimiter() method with the appropriate argument for the delimiter. A common delimiter is the line separator. Since each computer platform may define different sequence of control characters as its line separator, we use the System.getProperty() method to retrieve the sequence of control characters defining the line separator for the particular computer platform. For example,

```
String lineSeparator = System.getProperty(“line.separator”);
Scanner scanner = new Scanner(System.in);
scanner.useDelimiter(lineSeparator);
```

If we had use the lineSeparator as the delimiter in the above Scanner1 application, then the three numbers entered would be treated as a single token made up of a sequence of three numbers “1 2 3” and this would result in an InputMismatchException when the first nextInt() is called.

8.3.6 A Scanner Class Application

In the next example, we will consider a larger application showing how the Scanner class can be used to capture the various data input types. Code 8.7 is the application.

Code 8.7: ScannerInputs

```
import java.util.Scanner;

public class ScannerInputs {

  public static void main(String[] args) {
    boolean done    = false;
    String  name    = "";
    int     age     = 0;
    boolean citizen = false;
    double  fee     = 0.0;
    String  ls = System.getProperty("line.separator");

    while (!done) {
      try {
        Scanner scanner = new Scanner(System.in);
        scanner.useDelimiter(ls);
        System.out.print("\nEnter name: ");
        name = scanner.next();
        System.out.print("Enter age: ");
        age = scanner.nextInt();
        System.out.print("Citizen?(true/false): ");
        citizen = scanner.nextBoolean();
        System.out.print("Enter fee: ");
        fee = scanner.nextDouble();
        done = true;
      } catch (Exception e) {
        System.out.println("Exception: " + e.toString());
      }
    }
    System.out.println("\nYou have entered: ");
    System.out.println("Name:    " + name);
    System.out.println("Age:     " + age);
    System.out.println("Citizen: " + citizen);
    System.out.println("Fee:     " + fee);
  }
}
```

The Scanner object is set to use the line separator as the delimiter. The next() method is used to get a string. In this case, a name can be made up of a sequence of words e.g. "Harry Potter". An integer value is read in. Next, a boolean value which can be "true" or "false" is read in. A boolean value is recognized by the nextBoolean() method. We use nextDouble() to read in a double value from the input buffer. Finally, the values entered are printed. Any input value that does not conform to the expected data type will lead to InputMismatchException exception. An interaction from Code 8.7 is shown below:

```
Enter name: Harry Potter
Enter age: 23
Citizen?(true/false): false
Enter fee: 50.00

You have entered:
Name:    Harry Potter
Age:     23
Citizen: false
Fee:     50.0
```

The following shows a situation when a wrong value for Citizen is entered:

```
Enter name: Harry Potter
Enter age: 23
Citizen?(true/false): no
Exception: java.util.InputMismatchException
```

CHAPTER 9: FILE HANDLING

In this chapter we will extend our discussion on inputs and outputs with a special focus on file handling. In particular, we will discuss:

1. Text Files
2. Binary Files

9.1 Text Files

Files are constructs that reside on secondary storage such as hard disks. They can contain data that persist beyond the execution cycle of a program i.e. the data are still available even when the computer machine in which the Java programs execute is turned off.

9.1.1 Writing to a File

File input and output work very much in the same way as keyboard input and screen output respectively. They are all streams.

A file has a filename. To read or write to a file, we need to create file input and output streams that are associated with the filename.

A text-based file output stream object is created by the line:

```
PrintWriter pw = new PrintWriter(new FileWriter(“numbers.txt”));
```

Let us unpack the statement. “numbers.txt” is a String that represents the filename of the file we want to write data to. A FileWriter object allows us to write streams of characters. A PrintWriter object makes use of the FileWriter object, allowing us to use the familiar println() methods. Instead of using System.out.println() statements that print data onto the screen, we use pw.println() methods that store data into the output file. When we have finished writing output to the PrintWriter object using the println() methods, we need to close the output stream using the close() method. This also ensures that any output data that may have been buffered in memory is written into the output file. Once again, *exceptions* can occur. To turn the error handling over to Java, we must append the statements “throws IOException” at the end of the method header.

Code 9.1: File Output (Writing to a File)

```
import java.io.*;

public class WriteFile {
  public static void main(String[] args) throws IOException {
    PrintWriter pw;
    int random;

    pw = new PrintWriter(new FileWriter("numbers.txt"));

    for (int i = 1; i <= 10; i++) {
      random = (int)(Math.random() * 100) + 1;
      pw.println("Random number #" + i + ": " + random);
    }

    pw.close();
  }
}
```

Code 9.1 creates an output file named “numbers.txt”. You can check this by displaying the directory of the folder in which Code 9.1 resides. Next the program stores ten random numbers into the file. Each random number ranges from 1 to 100. Finally, the file is closed. Note that random numbers are generated by the class method Math.random(). Do you know which package does the Math class belong to?

The output file can be viewed using any text editor such as Microsoft Wordpad. A sample content of this output file is shown below:

```
Random number #1: 24
Random number #2: 21
Random number #3: 88
Random number #4: 99
Random number #5: 34
Random number #6: 16
Random number #7: 11
Random number #8: 41
Random number #9: 52
Random number #10: 39
```

9.1.2 Appending Texts to a File

When Code 9.1 is run, a file named “numbers.txt” is created. When we run this program again, the previously created random numbers are now replaced by a new set of random numbers. That is, the file “numbers.txt” is overwritten by the second run of the program.

Suppose we want to keep the previous set of random numbers in the file when we run the program the second time. Java needs to know that it should not overwrite the file but to append any new data to the end of the file. To achieve this, we create a FileWriter object with a different constructor method:

```
PrintWriter pw = new PrintWriter(new FileWriter(“numbers.txt”, true));
```

new FileWriter(“numbers.txt”, true) constructs a FileWriter object with file name “numbers.txt”. The second parameter “true” indicates to the FileWriter object to append rather than overwrite any data written to it.

9.1.3 Reading from a File

File input is similar to keyboard input. Instead of an InputStreamReader object, we replace it with a FileReader object:

```
BufferedReader br = new BufferedReader(new FileReader(“numbers.txt”));
```

Let us unpack the statement. “numbers.txt” is a String that represents the filename of the file we want to read data from. A FileReader object allows us to read streams of characters from a file. A BufferedReader object makes use of the FileReader object, providing for the efficient reading of data.

Consider Code 9.2. Lines are read from the input file line by line using br.readLine(). The latter returns a String object. Note that br has earlier been assigned to read from a file.

When the end of file is reached, br.readLine() returns null. This test for end-of-file is implemented in the while statement.

The usual cleanup proceeds with the calling of br.close() to close the file input stream. Exception handling is managed by Java.

Code 9.2: File Input (Reading from a File)
```
import java.io.*;

public class ReadFile {

  public static void main(String[] args) throws IOException {
    BufferedReader br;
    br = new BufferedReader(new FileReader("numbers.txt"));
    String s;

    while ((s = br.readLine()) != null)
      System.out.println(s);

    br.close();
  }
}
```

9.2 Binary Files

Text files work well when data are stored as semi-structured lines of text. However, we may wish to work with files in terms of bits and bytes at a very low level. Java provides the RandomAccessFile class (in java.io package) for this purpose.

In a binary file, the most basic data unit is usually a byte. Each byte is made up of eight bits and it stores an integral value between -128 to 127 inclusive.

A binary file has the added advantage that it can be opened for both reading and writing at the same time. Also, each opened binary file has a file pointer that can be moved forward and backward. We can choose to write a byte at the end of the file, jump to somewhere in the middle of the file to read a byte, then jump to the beginning of the file to write a byte, and etc.

While binary files offer flexibility, it is the programmer's responsibility to ensure correctness of operation.

Code 9.3: Binary File
```
import java.io.*;

public class BinaryFile {
  public static void main(String[] args) throws IOException {
    RandomAccessFile raf = new RandomAccessFile("rafile.dat", "rw");
    int b;
    long pointer;

    for (int i = 0; i < 100; i++) {
      raf.writeByte(65);
    }
    raf.seek(10); // Jump to pos 10, the 11th byte in the file.
    raf.writeByte(66); // Write character 'B' at this position.
    // Note that the pointer has moved by one byte due to the
    // writeByte(...) call.
    pointer = raf.getFilePointer();
    b = raf.readByte(); // Read the byte at position 11.
    System.out.println("Byte read at file pointer " + pointer +
                       " has ASCII value: " + b);
    raf.seek(10); // Jump to pos 10, the 11th byte in the file.
    pointer = raf.getFilePointer();
    b = raf.readByte(); // Read the byte at position 10.
    System.out.println("Byte read at file pointer " + pointer +
                       " has ASCII value: " + b);
    raf.close();
  }
}
```

We will illustrate the use of a binary file in Code 9.3. We will create a binary file named “rafile.dat”, write 100 consecutive bytes of the value 65 (which correspond to the ASCII character ‘A’), then jump to the 11th byte and change that byte to the value 66 (the ASCII character ‘B’). Next, we display the location of the pointer and read the value of the 12th byte. Finally, we return to file pointer position 10 (11th byte) and display the value of the byte (which is the character ‘B’). The comments in Code 9.3 are self-explanatory.

The random access file “rafile.dat” has the following values before the file is processed:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAA
```

Running Code 9.3 produces the following output:

```
Byte read at file pointer 11 has ASCII value: 65
Byte read at file pointer 10 has ASCII value: 66
```

The resulting binary file “rafile.dat” contains exactly 100 bytes of data, with each byte having the value “65” except for the 11th byte (file pointer 10) which has value “66” i.e.

```
AAAAAAAAAABAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAA
```

9.3 String Tokenizers

Suppose we have the following line of texts:

```
Tan Ah Meng;12345:;98.7
```

This line contains three items separated by “;”: “Tan Ah Meng”, “12345:”, and “98.7”. How do we extract these three components from the texts?

9.3.1 The java.util.StringTokenizer Class

There is a class in Java that allows you to do this very easily. It is called StringTokenizer. This class can be found in the java.util package.

Given a string and a list of delimiters, a StringTokenizer object is able to extract components of the string as tokens. Code 9.4 illustrates.

Code 9.4: The StringTokenizerTest Class

```
import java.util.*;

class StringTokenizerTest {

  public static void main(String[] args) {
    String s     = args[0]; // string for extraction
    String delim = args[1]; // delimiter
    StringTokenizer st = new StringTokenizer(s, delim);
    while (st.hasMoreTokens())
      System.out.println(st.nextToken());
  }
}
```

To execute Code 9.4, we enter the following at the command line prompt:

```
java StringTokenizerTest "Tan Ah Meng;12345:;98.7" ";"
```

The first argument contains the string to be processed and the second argument defines the delimiter characters. A StringTokenizer object is instantiated with two parameters: a String s and the delimiter characters. The code iterates through the tokens and display them one by one, producing the following outputs:

```
Tan Ah Meng
12345:
98.7
```

Notice the delimiter character ";" is not be treated as a token.

The StringTokenizer class has a method called hasMoreTokens(). This method returns a Boolean value true, if there are still more tokens to be retrieved, and false otherwise. The nextToken() method in the StringTokenizer class retrieves the next token.

Suppose we include ":" as one of the delimiter characters:

```
java StringTokenizerTest "Tan Ah Meng;12345:;98.7" ";:"
```

The following output is produced instead:

```
Tan Ah Meng
12345
98.7
```

Suppose we add a space (" ") as a delimiter characters:

```
java StringTokenizerTest "Tan Ah Meng;12345:;98.7" "; :"
```

we get:

```
Tan
Ah
Meng
12345
98.7
```

To use the StringTokenizer class, we need to import the java.util package. This class has three constructor methods and one of them takes in only one argument:

```
StringTokenizer(String s)
```

This method constructs a string tokenizer for the specified string s with the default delimiter set (Note: The default delimiter set includes the space character, the tab character, the newline character, the carriage-return character, and the form-feed character. We can specify them as " \t\n\r\f"). Using this constructor method, we will get the following output:

```
Tan
Ah
Meng;12345:;98.7
```

9.3.2 Delimiter Characters

Various delimiter characters can be used to break up the tokens in a string. The following example shows that a double quote commonly used to encapsulate a string can also be treated as a delimiter character. For example, given the following string:

```
A simple "String Tokenizer" example!
```

extract the words in the sentence. We will use the same Code 9.4 to illustrate this example. Let us begin with space as the delimiter character:

```
java StringTokenizerTest "A simple \"String Tokenizer\" example!" " "
```

Note the use of "\" to inform Java to treat the following character as a literal. The following output is produced:

```
A
simple
"String
Tokenizer"
example!
```

For the above requirement, we add two more delimiter characters, a double quote and an exclamation mark:

```
java StringTokenizerTest "A simple \"String Tokenizer\" example!" " \"!"
```

The output is now:

```
A
simple
String
Tokenizer
example
```